HDL Coder™

User's Guide

MATLAB® & SIMULINK®

R2017a

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

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

HDL Coder™ User's Guide

Revision History

March 2012	Online only	New for Version 3.0 (R2012a)
September 2012	Online only	Revised for Version 3.1 (R2012b)
March 2013	Online only	Revised for Version 3.2 (R2013a)
September 2013	Online only	Revised for Version 3.3 (R2013b)
March 2014	Online only	Revised for Version 3.4 (R2014a)
October 2014	Online only	Revised for Version 3.5 (R2014b)
March 2015	Online only	Revised for Version 3.6 (R2015a)
September 2015	Online only	Revised for Version 3.7 (R2015b)
October 2015	Online only	Rereleased for Version 3.6.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 3.8 (R2016a)
September 2016	Online only	Revised for Version 3.9 (R2016b)
March 2017	Online only	Revised for Version 3.10 (Release 2017a)

HDL Code Generation from MATLAB

4 Fixed-Point Conversion

5 Code Generation

HDL Code Generation from Simulink

10 Model Design for HDL Code Generation

11 Code Generation Options in the HDL Coder Dialog Boxes

13 Generating HDL Code for Multirate Models

15 Optimization

16 Code Generation Reports, HDL Compatibility Checker, Block Support Library, and Code Annotation

17 HDL Coding Standards

20 Generating HDL Code with the MATLAB Function Block

22 Using the HDL Workflow Advisor

Hardware-Software Codesign

26 Hardware-Software Co-Design Basics

27 Target SoC Platforms and Speedgoat Boards

HDL Code Generation from MATLAB

Functions Supported for HDL Code Generation

Functions Supported for HDL Code Generation — Alphabetical List

You can generate efficient HDL code for a subset of MATLAB built-in functions and toolbox functions that you call from MATLAB code. These functions appear in alphabetical order in the following table.

To find supported functions by MATLAB category or toolbox, see “Functions Supported for HDL Code Generation — Categorical List” on page 1-10.

Name	Product	Remarks and Limitations
`abs`	Fixed-Point Designer™	Double and complex data types not supported.
`add`	Fixed-Point Designer	—
`all`	Fixed-Point Designer	Double data type not supported.
`and`	MATLAB	—
`any`	Fixed-Point Designer	Double data type not supported.
`bitand`	MATLAB	—
`bitand`	Fixed-Point Designer	—
`bitandreduce`	Fixed-Point Designer	—
`bitcmp`	MATLAB	—
`bitcmp`	Fixed-Point Designer	—
`bitconcat`	Fixed-Point Designer	—
`bitget`	MATLAB	—
`bitget`	Fixed-Point Designer	—
`bitor`	MATLAB	—
`bitor`	Fixed-Point Designer	—
`bitorreduce`	Fixed-Point Designer	—
`bitreplicate`	Fixed-Point Designer	—
`bitrol`	Fixed-Point Designer	—
`bitror`	Fixed-Point Designer	—
`bitset`	MATLAB	—
`bitset`	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
`bitshift`	MATLAB	For efficient HDL code generation, use the Fixed-Point Designer functions `bitsll`, `bitsrl`, or `bitsra` instead of `bitshift`.
`bitshift`	Fixed-Point Designer	—
`bitsliceget`	Fixed-Point Designer	—
`bitsll`	Fixed-Point Designer	—
`bitsra`	Fixed-Point Designer	—
`bitsrl`	Fixed-Point Designer	—
`bitxor`	MATLAB	—
`bitxor`	Fixed-Point Designer	—
`bitxorreduce`	Fixed-Point Designer	—
`ceil`	Fixed-Point Designer	—
`complex`	MATLAB	—
`complex`	Fixed-Point Designer	—
`conj`	Fixed-Point Designer	—
`convergent`	Fixed-Point Designer	—
`ctranspose`	MATLAB	—
`ctranspose`	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
`divide`	Fixed-Point Designer	• For HDL Code generation, the divisor must be a constant and a power of two. • Non-`fi` inputs must be constant; that is, their values must be known at compile time so that they can be cast to `fi` objects. • Complex and imaginary divisors are not supported. • Code generation in MATLAB does not support the syntax `T.divide(a,b)`.
`end`	Fixed-Point Designer	—
`eps`	Fixed-Point Designer	• Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix, `fi` single and `fi` double signals.
`eq`	MATLAB	—
`eq`	Fixed-Point Designer	—
`fi`	Fixed-Point Designer	—
`fimath`	Fixed-Point Designer	—
`fix`	Fixed-Point Designer	—
`floor`	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
`for`	MATLAB	Do not use `for` loops without static bounds. Do not use the `&` and `|` operators within conditions of a `for` statement. Instead, use the `&&` and `||` operators. HDL Coder™ does not support nonscalar expressions in the conditions of `for` statements. Instead, use the `all` or `any` functions to collapse logical vectors into scalars.
`ge`	MATLAB	—
`ge`	Fixed-Point Designer	—
`getlsb`	Fixed-Point Designer	—
`getmsb`	Fixed-Point Designer	—
`gt`	MATLAB	—
`gt`	Fixed-Point Designer	—
`horzcat`	Fixed-Point Designer	—
`if`	MATLAB	Do not use the `&` and `|` operators within conditions of an `if` statement. Instead, use the `&&` and `||` operators. HDL Coder does not support nonscalar expressions in the conditions of `if` statements. Instead, use the `all` or `any` functions to collapse logical vectors into scalars.
`imag`	MATLAB	—
`imag`	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
int8, int16, int32	Fixed-Point Designer	—
iscolumn	Fixed-Point Designer	—
isempty	Fixed-Point Designer	—
isequal	Fixed-Point Designer	—
isfi	Fixed-Point Designer	—
isfimath	Fixed-Point Designer	—
isfimathlocal	Fixed-Point Designer	—
isfinite	Fixed-Point Designer	—
isinf	Fixed-Point Designer	—
isnan	Fixed-Point Designer	—
isnumeric	Fixed-Point Designer	—
isnumerictype	Fixed-Point Designer	—
isreal	Fixed-Point Designer	—
isrow	Fixed-Point Designer	—
isscalar	Fixed-Point Designer	—
issigned	Fixed-Point Designer	—
isvector	Fixed-Point Designer	—
le	MATLAB	—
le	Fixed-Point Designer	—
length	Fixed-Point Designer	—
logical	Fixed-Point Designer	—
lowerbound	Fixed-Point Designer	—
lsb	Fixed-Point Designer	—
lt	MATLAB	—
lt	Fixed-Point Designer	—
max	Fixed-Point Designer	—
min	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
`minus`	Fixed-Point Designer	—
`mpower`	MATLAB	Both inputs must be scalar, and the exponent input, `k`, must be an integer.
`mpower`	Fixed-Point Designer	Both inputs must be scalar, and the exponent input, `k`, must be a constant integer.
`mtimes(A,B)`	MATLAB	—
`mtimes`	Fixed-Point Designer	—
`ndims`	Fixed-Point Designer	—
`ne`	MATLAB	—
`ne`	Fixed-Point Designer	—
`nearest`	Fixed-Point Designer	—
`not`	MATLAB	—
`numberofelements`	Fixed-Point Designer	—
`numerictype`	Fixed-Point Designer	—
`ones`	MATLAB	Dimensions must be real, nonnegative integers.
`or`	MATLAB	—
`plus`	MATLAB	Inputs cannot be data type `logical`.
`plus`	Fixed-Point Designer	Inputs cannot be data type `logical`.
`power`	MATLAB	Both inputs must be scalar, and the exponent input, `k`, must be an integer.
`power`	Fixed-Point Designer	Both inputs must be scalar, and the exponent input, `k`, must be a constant integer.
`range`	Fixed-Point Designer	—
`real`	MATLAB	—

Name	Product	Remarks and Limitations
`real`	Fixed-Point Designer	—
`realmax`	Fixed-Point Designer	—
`realmin`	Fixed-Point Designer	—
`reinterpretcast`	Fixed-Point Designer	—
`repmat`	Fixed-Point Designer	—
`rescale`	Fixed-Point Designer	—
`reshape`	Fixed-Point Designer	—
`round`	Fixed-Point Designer	—
`sfi`	Fixed-Point Designer	—
`sign`	Fixed-Point Designer	—
`size`	Fixed-Point Designer	—
`sqrt`	Fixed-Point Designer	—
`sub`	Fixed-Point Designer	—
`subsasgn`	Fixed-Point Designer	Supported data types for HDL code generation are listed in “Supported Data Types” on page 2-2.
`subsref`	Fixed-Point Designer	Supported data types for HDL code generation are listed in “Supported Data Types” on page 2-2.
`sum`	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
`switch`	MATLAB	The conditional expression in a `switch` or `case` statement must use only: • `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32` data types • Scalar data If multiple `case` statements make assignments to the same variable, the numeric type and `fimath` specification for that variable must be the same in every `case` statement.
`times`	MATLAB	Inputs cannot be data type `logical`.
`times`	Fixed-Point Designer	Inputs cannot be data type `logical`.
`transpose`	MATLAB	—
`transpose`	Fixed-Point Designer	—
`ufi`	Fixed-Point Designer	—
`uint8`, `uint16`, `uint32`	Fixed-Point Designer	—
`uminus`	Fixed-Point Designer	—
`uplus`	Fixed-Point Designer	Inputs cannot be data type `logical`.
`upperbound`	Fixed-Point Designer	—
`vertcat`	Fixed-Point Designer	—
`xor`	MATLAB	—
`zeros`	MATLAB	Dimensions must be real, nonnegative integers.

Functions Supported for HDL Code Generation — Categorical List

In this section...

You can generate efficient HDL code for a subset of MATLAB built-in functions and toolbox functions that you call from MATLAB code. These functions are listed by MATLAB category or toolbox category in the following tables.

For an alphabetical list of supported functions, see “Functions Supported for HDL Code Generation — Alphabetical List” on page 1-2.

Arithmetic Operations in MATLAB

Name	Remarks and Limitations
`ctranspose(A)`	—
`mpower(A,B)`	`A` and `B` must be scalar, and `B` must be an integer.
`mtimes(A,B)`	—
`plus(A,B)`	Neither `A` nor `B` can be data type `logical`.
`power(A,B)`	`A` and `B` must be scalar, and `B` must be an integer.
`times(A,B)`	Neither `A` nor `B` can be data type `logical`.
`transpose(A)`	—

Bitwise Operations in MATLAB

Name	Remarks and Limitations
`bitand`	—

Name	Remarks and Limitations
`bitcmp`	—
`bitget`	—
`bitor`	—
`bitset`	—
`bitshift`	For efficient HDL code generation, use the Fixed-Point Designer functions `bitsll`, `bitsrl`, or `bitsra` instead of `bitshift`.
`bitxor`	—

Complex Numbers in MATLAB

Name	Remarks and Limitations
`complex`	—
`imag`	—
`real`	—

Control Flow in MATLAB

Name	Remarks and Limitations
`for`	Do not use `for` loops without static bounds. Do not use the `&` and `|` operators within conditions of a `for` statement. Instead, use the `&&` and `||` operators. HDL Coder does not support nonscalar expressions in the conditions of `for` statements. Instead, use the `all` or `any` functions to collapse logical vectors into scalars.
`if`	Do not use the `&` and `|` operators within conditions of an `if` statement. Instead, use the `&&` and `||` operators. HDL Coder does not support nonscalar expressions in the conditions of `if` statements. Instead, use the `all` or `any` functions to collapse logical vectors into scalars.

Name	Remarks and Limitations
switch	The conditional expression in a switch or case statement must use only: • uint8, uint16, uint32, int8, int16, or int32 data types • Scalar data If multiple case statements make assignments to the same variable, the numeric type and fimath specification for that variable must be the same in every case statement.

Logical Operators in MATLAB

Name	Remarks and Limitations
and	—
not	—
or	—
xor	—

Arrays in MATLAB

Name	Remarks and Limitations
ones	Dimensions must be real, nonnegative integers.
zeros	Dimensions must be real, nonnegative integers.

Relational Operators in MATLAB

Name	Remarks and Limitations
eq	—
ge	—
gt	—
le	—
lt	—
ne	—

Fixed-Point Designer

HDL code generation support for fixed-point run-time library functions from the Fixed-Point Designer is summarized in the following table. See "Fixed-Point Function Limitations" on page 2-35 for general limitations of fixed-point run-time library functions for code generation.

Function	Remarks and Limitations
`abs`	Double and complex data types not supported.
`add`	—
`all`	Double data type not supported.
`any`	Double data type not supported.
`bitand`	—
`bitandreduce`	—
`bitcmp`	—
`bitconcat`	—
`bitget`	—
`bitor`	—
`bitorreduce`	—
`bitreplicate`	—
`bitrol`	—
`bitror`	—
`bitset`	—
`bitshift`	—
`bitsliceget`	—
`bitsll`	—
`bitsra`	—
`bitsrl`	—
`bitxor`	—
`bitxorreduce`	—
`ceil`	—

Function	Remarks and Limitations
`complex`	—
`conj`	—
`convergent`	—
`ctranspose`	—
`divide`	• For HDL Code generation, the divisor must be a constant and a power of two. • Non-`fi` inputs must be constant; that is, their values must be known at compile time so that they can be cast to `fi` objects. • Complex and imaginary divisors are not supported. • Code generation in MATLAB does not support the syntax `T.divide(a,b)`.
`end`	—
`eps`	• Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix, `fi` single and `fi` double signals.
`eq`	—
`fi`	—
`fimath`	—
`fix`	—
`floor`	—
`ge`	—
`getlsb`	—
`getmsb`	—
`gt`	—
`horzcat`	—
`imag`	—
`int8`, `int16`, `int32`	—
`iscolumn`	—
`isempty`	—

Function	Remarks and Limitations
`isequal`	—
`isfi`	—
`isfimath`	—
`isfimathlocal`	—
`isfinite`	—
`isinf`	—
`isnan`	—
`isnumeric`	—
`isnumerictype`	—
`isreal`	—
`isrow`	—
`isscalar`	—
`issigned`	—
`isvector`	—
`le`	—
`length`	—
`logical`	—
`lowerbound`	—
`lsb`	—
`lt`	—
`max`	—
`min`	—
`minus`	—
`mpower`	Both inputs must be scalar, and the exponent input, `k`, must be a constant integer.
`mtimes`	—
`ndims`	—
`ne`	—

Function	Remarks and Limitations
`nearest`	—
`numberofelements`	—
`numerictype`	—
`ones`	Dimensions must be real, nonnegative integers.
`plus`	Inputs cannot be data type `logical`.
`power`	Both inputs must be scalar, and the exponent input, `k`, must be a constant integer.
`range`	—
`real`	—
`realmax`	—
`realmin`	—
`reinterpretcast`	—
`repmat`	—
`rescale`	—
`reshape`	—
`round`	—
`sfi`	—
`sign`	—
`size`	—
`sqrt`	—
`sub`	—
`subsasgn`	Supported data types for HDL code generation are listed in "Supported Data Types" on page 2-2.
`subsref`	Supported data types for HDL code generation are listed in "Supported Data Types" on page 2-2.
`sum`	—
`times`	Inputs cannot be data type `logical`.
`transpose`	—

Function	Remarks and Limitations
`ufi`	—
`uint8`, `uint16`, `uint32`	—
`uminus`	—
`uplus`	Inputs cannot be data type `logical`.
`upperbound`	—
`vertcat`	—

MATLAB Algorithm Design

Data Types and Scope

Supported Data Types

HDL Coder supports the following subset of MATLAB data types.

Types	Supported Data Types	Restrictions
Integer	• `uint8`, `uint16`, `uint32`, `uint64` • `int8`, `int16`, `int32`, `int64`	In Simulink®, MATLAB Function block ports must use numeric types `sfix64` or `ufix64` for 64-bit data.
Real	• `double` • `single`	HDL code generated with `double` or `single` data types can be used for simulation, but is not synthesizable.
Character	`char`	
Logical	`logical`	
Fixed point	• Scaled (binary point only) fixed-point numbers • Custom integers (zero binary point)	Fixed-point numbers with slope (not equal to 1.0) and bias (not equal to 0.0) are not supported. Maximum word size for fixed-point numbers is 128 bits.
Vectors	• unordered `{N}` • row `{1, N}` • column `{N, 1}`	The maximum number of vector elements allowed is 2^32. Before a variable is subscripted, it must be fully defined.
Matrices	`{N, M}`	Matrices are supported in the body of the design algorithm, but are not supported as inputs to the top-level design function. Do not use matrices in the testbench.
Structures	`struct`	Arrays of structures are not supported.

Types	Supported Data Types	Restrictions
		For the FPGA Turnkey and IP Core Generation workflows, structures are supported in the body of the design algorithm, but are not supported as inputs to the top-level design function.
Enumerations	`enumeration`	Enumeration values must be monotonically increasing. If your target language is Verilog®, all enumeration member names must be unique within the design. Enumerations at the top-level DUT ports are not supported with the following workflows or verification methods: • IP Core Generation workflow • FPGA Turnkey workflow • FPGA-in-the-Loop • HDL Cosimulation

Unsupported Data Types

In the current release, the following data types are not supported:

- Cell array
- `Inf`

Scope for Variables

Global variables are not supported for HDL code generation.

Operators

Arithmetic Operators

HDL Coder supports the arithmetic operators (and equivalent MATLAB functions) listed in the following table.

Note: HDL code generated for large vector and matrix inputs to arithmetic operations can result in inefficient code. The code for these operators is not automatically pipelined.

Operation	Operator Syntax	Equivalent Function	Restrictions
Binary addition	`A+B`	`plus(A,B)`	Neither `A` nor `B` can be data type `logical`.
Matrix multiplication	`A*B`	`mtimes(A,B)`	HDL code generated for matrix arithmetic operations is not pipelined, and can result in inefficient code.
Arraywise multiplication	`A.*B`	`times(A,B)`	Neither `A` nor `B` can be data type `logical`.
Matrix power	`A^B`	`mpower(A,B)`	`A` and `B` must be scalar, and `B` must be an integer. HDL code generated for matrix arithmetic operations is not pipelined, and can result in inefficient code.

Operation	Operator Syntax	Equivalent Function	Restrictions
Arraywise power	`A.^B`	`power(A,B)`	`A` and `B` must be scalar, and `B` must be an integer.
Complex transpose	`A'`	`ctranspose(A)`	
Matrix transpose	`A.'`	`transpose(A)`	
Matrix concat	`[A B]`	None	
Matrix index	`A(r c)`	None	Before you use a variable, you must fully define it.

Relational Operators

HDL Coder supports the relational operators (and equivalent MATLAB functions) listed in the following table.

Relation	Operator Syntax	Equivalent Function
Less than	`A<B`	`lt(A,B)`
Less than or equal to	`A<=B`	`le(A,B)`
Greater than or equal to	`A>=B`	`ge(A,B)`
Greater than	`A>B`	`gt(A,B)`
Equal	`A==B`	`eq(A,B)`
Not equal	`A~=B`	`ne(A,B)`

Logical Operators

HDL Coder supports the logical operators (and equivalent MATLAB functions) listed in the following table.

Relation	Operator Syntax	M Function Equivalent	Notes
Logical And	`A&B`	`and(A,B)`	
Logical Or	`A|B`	`or(A,B)`	

Relation	Operator Syntax	M Function Equivalent	Notes
Logical Xor	`A xor B`	`xor(A,B)`	
Logical And (short circuiting)	`A&&B`	N/A	Use short circuiting logical operators within conditionals. See also “Control Flow Statements” on page 2-7.
Logical Or (short circuiting)	`A||B`	N/A	Use short circuiting logical operators within conditionals. See also “Control Flow Statements” on page 2-7.
Element complement	`~A`	`not(A)`	

Control Flow Statements

HDL Coder supports the following control flow statements and constructs with restrictions.

Control Flow Statement	Restrictions
`for`	Do not use `for` loops without static bounds. Do not use the `&` and `|` operators within conditions of a `for` statement. Instead, use the `&&` and `||` operators. HDL Coder does not support nonscalar expressions in the conditions of `for` statements. Instead, use the `all` or `any` functions to collapse logical vectors into scalars.
`if`	Do not use the `&` and `|` operators within conditions of an `if` statement. Instead, use the `&&` and `||` operators. HDL Coder does not support nonscalar expressions in the conditions of `if` statements. Instead, use the `all` or `any` functions to collapse logical vectors into scalars.
`switch`	The conditional expression in a `switch` or `case` statement must use only: • `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32` data types • Scalar data If multiple `case` statements make assignments to the same variable, the numeric type and `fimath` specification for that variable must be the same in every `case` statement.

The following control flow statements are not supported:

- `while`
- `break`
- `continue`
- `return`
- `parfor`

Vector Function Limitations Related to Control Statements

Avoid using the following vector functions, as they may generate loops containing `break` statements:

- `isequal`
- `bitrevorder`

Persistent Variables

Persistent variables enable you to model registers. If you need to preserve state between invocations of your MATLAB algorithm, use persistent variables.

Before you use a persistent variable, you must initialize it with a statement specifying its size and type. You can initialize a persistent variable with either a constant value or a variable, as in the following examples:

```
% Initialize with a constant
persistent p;
if isempty(p)
    p = fi(0,0,8,0);
end


% Initialize with a variable
initval = fi(0,0,8,0);

persistent p;
if isempty(p)
    p = initval;
end
```

Use a logical expression that evaluates to a constant to test whether a persistent variable has been initialized, as in the preceding examples. Using a logical expression that evaluates to a constant ensures that the generated HDL code for the test is executed only once, as part of the reset process.

You can initialize multiple variables within a single logical expression, as in the following example:

```
% Initialize with  variables
initval1 = fi(0,0,8,0);
initval2 = fi(0,0,7,0);

persistent p;
if isempty(p)
    x = initval1;
    y = initval2;
end
```

Note: If persistent variables are not initialized as described above, extra sentinel variables can appear in the generated code. These sentinel variables can translate to inefficient hardware.

Persistent Array Variables

Persistent array variables enable you to model RAM.

By default, the HDL Coder software optimizes the area of your design by mapping persistent array variables to RAM. If persistent array variables are not mapped to RAM, they map to registers. RAM mapping can therefore reduce the area of your design in the target hardware.

To learn how persistent array variables map to RAM, see “Map Persistent Arrays and dsp.Delay to RAM” on page 8-3.

Complex Data Type Support

In this section...

Declaring Complex Signals

The following MATLAB code declares several local complex variables. `x` and `y` are declared by complex constant assignment; `z` is created using the using the `complex()` function.

```
function [x,y,z] = fcn

% create 8 bit complex constants
x = uint8(1 + 2i);
y = uint8(3 + 4j);
z = uint8(complex(5, 6));
```

The following code example shows VHDL® code generated from the previous MATLAB code.

```
ENTITY complex_decl IS
    PORT (
        clk : IN std_logic;
        clk_enable : IN std_logic;
        reset : IN std_logic;
        x_re : OUT std_logic_vector(7 DOWNTO 0);
        x_im : OUT std_logic_vector(7 DOWNTO 0);
        y_re : OUT std_logic_vector(7 DOWNTO 0);
        y_im : OUT std_logic_vector(7 DOWNTO 0);
        z_re : OUT std_logic_vector(7 DOWNTO 0);
        z_im : OUT std_logic_vector(7 DOWNTO 0));
END complex_decl;


ARCHITECTURE fsm_SFHDL OF complex_decl IS

BEGIN
    x_re <= std_logic_vector(to_unsigned(1, 8));
    x_im <= std_logic_vector(to_unsigned(2, 8));
    y_re <= std_logic_vector(to_unsigned(3, 8));
    y_im <= std_logic_vector(to_unsigned(4, 8));
    z_re <= std_logic_vector(to_unsigned(5, 8));
    z_im <= std_logic_vector(to_unsigned(6, 8));
END fsm_SFHDL;
```

As shown in the example, complex inputs, outputs and local variables declared in MATLAB code expand into real and imaginary signals. The naming conventions for these derived signals are:

- Real components have the same name as the original complex signal, suffixed with the default string `'_re'` (for example, `x_re`). To specify a different suffix, set the **Complex real part postfix** option (or the corresponding `ComplexRealPostfix` CLI property).
- Imaginary components have the same name as the original complex signal, suffixed with the string `'_im'` (for example, `x_im`). To specify a different suffix, set the **Complex imaginary part postfix** option (or the corresponding `ComplexImagPostfix` CLI property).

A complex variable declared in MATLAB code remains complex during the entire length of the program.

Conversion Between Complex and Real Signals

The MATLAB code provides access to the fields of a complex signal via the `real()` and `imag()` functions, as shown in the following code.

```
function [Re_part, Im_part]= fcn(c)
% Output real and imaginary parts of complex input signal

Re_part = real(c);
Im_part = imag(c);
```

HDL Coder supports these constructs, accessing the corresponding real and imaginary signal components in generated HDL code. In the following Verilog code example, the MATLAB complex signal variable `c` is flattened into the signals `c_re` and `c_im`. Each of these signals is assigned to the output variables `Re_part` and `Im_part`, respectively.

```
module Complex_To_Real_Imag (clk, clk_enable, reset, c_re, c_im, Re_part, Im_part );

    input clk;
    input clk_enable;
    input reset;
    input [3:0] c_re;
    input [3:0] c_im;
    output [3:0] Re_part;
    output [3:0] Im_part;

    // Output real and imaginary parts of complex input signal
    assign Re_part = c_re;
    assign Im_part = c_im;
```

Support for Vectors of Complex Numbers

You can generate HDL code for vectors of complex numbers. Like scalar complex numbers, vectors of complex numbers are flattened down to vectors of real and imaginary parts in generated HDL code.

For example in the following script `t` is a complex vector variable of base type `ufix4` and size `[1,2]`.

```
function y = fcn(u1, u2)

t = [u1 u2];
y = t+1;
```

In the generated HDL code the variable `t` is broken down into real and imaginary parts with the same two-element array. .

```
VARIABLE t_re : vector_of_unsigned4(0 TO 3);
VARIABLE t_im : vector_of_unsigned4(0 TO 3);
```

The real and imaginary parts of the complex number have the same vector of type `ufix4`, as shown in the following code.

```
TYPE vector_of_unsigned4 IS ARRAY (NATURAL RANGE <>) OF unsigned(3 DOWNTO 0);
```

Complex vector-based operations (`+`,`-`,`*` etc.,) are similarly broken down to vectors of real and imaginary parts. Operations are performed independently on the elements of such vectors, following MATLAB semantics for vectors of complex numbers.

In both VHDL and Verilog code generated from MATLAB code, complex vector ports are always flattened. If complex vector variables appear on inputs and outputs, real and imaginary vector components are further flattened to scalars.

In the following code, `u1` and `u2` are scalar complex numbers and `y` is a vector of complex numbers.

```
function y = fcn(u1, u2)

t = [u1 u2];
y = t+1;
```

This generates the following port declarations in a VHDL entity definition.

```
ENTITY _MATLAB_Function IS
    PORT (
        clk : IN std_logic;
        clk_enable : IN std_logic;
        reset : IN std_logic;
        u1_re : IN vector_of_std_logic_vector4(0 TO 1);
        u1_im : IN vector_of_std_logic_vector4(0 TO 1);
        u2_re : IN vector_of_std_logic_vector4(0 TO 1);
        u2_im : IN vector_of_std_logic_vector4(0 TO 1);
        y_re : OUT vector_of_std_logic_vector32(0 TO 3);
        y_im : OUT vector_of_std_logic_vector32(0 TO 3));
END _MATLAB_Function;
```

HDL Code Generation for System Objects

In this section...

HDL Coder supports both predefined and user-defined System objects for code generation.

Why Use System Objects?

System objects provide a design advantage because:

- You can save time during design and testing by using existing System object components.
- You can design and qualify custom System objects for reuse in multiple designs.
- You can define your algorithm in a System object once, and reuse multiple instances of it in a single MATLAB design.

  This idiom cannot be used with MATLAB functions that have state. For example, if the algorithm has state and requires the use of persistent variables, that function cannot be instantiated multiple times in a design. Instead, you would need to copy and rename the function for each instance.
- HDL code that you generate from System objects is modular and more readable.

Predefined System Objects

Predefined System objects that are available with MATLAB, DSP System Toolbox™, and Communications System Toolbox™ are supported for HDL code generation. For a list, see "Predefined System Objects Supported for HDL Code Generation" on page 2-18.

User-Defined System Objects

You can create user-defined System objects for HDL code generation. For an example, see "Generate Code for User-Defined System Objects" on page 2-21.

Limitations of HDL Code Generation for System Objects

The following limitations apply to HDL code generation for all System objects:

- Your design can call the `step` method only once per System object.
- `step` must not be inside a nested conditional statement, such as a nested loop, `if` statement, or `switch` statement.
- `step` must not be inside a conditional statement that contains a matrix indexing operation.
- A System object must be declared persistent if it has state.

  A System object has state when it has a tunable private or public property, or a property with the `DiscreteState` attribute.
- You can use the `dsp.Delay` System object only in feed-forward delay modeling.
- `UseMatrixTypesInHDL` attribute must be set to 'off', if you have a System object in your MATLAB code.
- Enumerations are not supported.
- Global variables are not supported.

Supported Methods

For predefined System Objects, `step` is the only method supported for HDL code generation.

For user-defined System Objects, either the `step` method, or the `output` and `update` methods, are supported for HDL code generation.

Additional Restrictions for Predefined System Objects

Predefined System objects are not supported for HDL code generation from within a MATLAB System block.

Additional Restrictions for User-Defined System Objects

In addition to the limitations for all System objects, the following restrictions apply to user-defined System objects for HDL code generation:

- In the `setupImpl` and `resetImpl` methods, if you assign values to properties or variables, the values must be constants.

- If your design uses the `output` and `update` methods, it can call each method only once per System object.
- Initial and reset values for properties must be compile-time constant.
- User-defined System objects must not be public properties.
- A `step` method with multiple outputs cannot be called within a conditional statement.

System object Examples for HDL Code Generation

To learn how to use System objects for HDL code generation, view the MATLAB designs in the following examples:

- "HDL Code Generation from System Objects" on page 5-14
- "Model State with Persistent Variables and System Objects" on page 2-37
- "Generate Code for User-Defined System Objects" on page 2-21
- "Integrate Custom HDL Code Into MATLAB Design" on page 5-21

Predefined System Objects Supported for HDL Code Generation

In this section...

Predefined System Objects in MATLAB Code

HDL Coder supports the following MATLAB System objects for HDL code generation:

- hdl.RAM
- hdl.BlackBox

HDL Coder supports the following Communications System Toolbox System objects for HDL code generation:

- comm.BPSKModulator, comm.BPSKDemodulator
- comm.PSKModulator, comm.PSKDemodulator
- comm.QPSKModulator, comm.QPSKDemodulator
- comm.RectangularQAMModulator, comm.RectangularQAMDemodulator
- comm.ConvolutionalInterleaver, comm.ConvolutionalDeinterleaver
- comm.ViterbiDecoder
- comm.HDLCRCDetector, comm.HDLCRCGenerator
- comm.HDLRSDecoder, comm.HDLRSEncoder

HDL Coder supports the following DSP System Toolbox System objects for HDL code generation:

- dsp.Delay
- dsp.Maximum
- dsp.Minimum
- dsp.BiquadFilter
- dsp.DCBlocker
- dsp.HDLComplexToMagnitudeAngle
- dsp.HDLFIRRateConverter

- dsp.HDLFFT, dsp.HDLIFFT
- dsp.HDLChannelizer
- dsp.HDLNCO
- dsp.FIRFilter
- dsp.HDLFIRFilter

HDL Coder supports the following Vision HDL Toolbox™ System objects for HDL code generation:

- visionhdl.ChromaResampler
- visionhdl.ColorSpaceConverter
- visionhdl.DemosaicInterpolator
- visionhdl.EdgeDetector
- visionhdl.GammaCorrector
- visionhdl.LookupTable
- visionhdl.Histogram
- visionhdl.ImageStatistics
- visionhdl.ROISelector
- visionhdl.PixelStreamAligner
- visionhdl.ImageFilter
- visionhdl.MedianFilter
- visionhdl.Closing
- visionhdl.Dilation
- visionhdl.Erosion
- visionhdl.Opening
- visionhdl.GrayscaleClosing
- visionhdl.GrayscaleDilation
- visionhdl.GrayscaleErosion
- visionhdl.GrayscaleOpening

Predefined System Objects in the MATLAB System Block

A subset of these predefined System objects are supported for code generation when you use them in a MATLAB System block. To learn more, see MATLAB System.

Load constants from a MAT-File

You can load compile-time constants from a MAT-file with the `coder.load` function in your MATLAB design.

For example, you can create a MAT-file, `sinvals.mat`, that contains fixed-point values of `sin` by entering the following commands in MATLAB:

```
sinvals = sin(fi(-pi:0.1:pi, 1, 16,15));
save sinvals.mat sinvals;
```

You can then generate HDL code from the following MATLAB code, which loads the constants from `sinvals.mat` into a persistent variable, `pConstStruct`, and assigns the values to a variable that is not persistent, `sv`.

```
persistent pConstStruct;
if isempty(pConstStruct)
    pConstStruct = coder.load('sinvals.mat');
end
sv = pConstStruct.sinvals;
```

Generate Code for User-Defined System Objects

In this section...

How To Create A User-Defined System object

To create a user-defined System object and generate code:

1 Create a class that subclasses from matlab.System.
2 Define one of the following sets of methods:

- `setupImpl` and `stepImpl`
- `setupImpl`, `outputImpl`, and `updateImpl`

  To use the `outputImpl` and `updateImpl` methods, your System object must also inherit from the `matlab.system.mixin.Nondirect` class.

3 Optionally, if your System object has private state properties, define the `resetImpl` method to initialize them to zero.
4 Write a top-level design function that creates an instance of your System object and calls the `step` method, or the `output` and `update` methods.

Note: The `resetImpl` method runs automatically during System object initialization. For HDL code generation, you cannot call the public `reset` method.

5 Write a test bench function that exercises the top-level design function.
6 Generate HDL code.

User-Defined System object Example

This example shows how to generate HDL code for a user-defined System object that implements the `setupImpl` and `stepImpl` methods.

1 In a writable folder, create a System object, `CounterSysObj`, which subclasses from `matlab.System`. Save the code as `CounterSysObj.m`.

```
classdef CounterSysObj < matlab.System
```

```
    properties (Nontunable)
        Threshold = int32(1)
    end
    properties (Access=private)
        State
        Count
    end
    methods
        function obj = CounterSysObj(varargin)
            setProperties(obj,nargin,varargin{:});
        end
    end

    methods (Access=protected)
        function setupImpl(obj, ~)
            % Initialize states
            obj.Count = int32(0);
            obj.State = int32(0);
        end
        function y = stepImpl(obj, u)
            if obj.Threshold > u(1)
                obj.Count(:) = obj.Count + int32(1); % Increment count
            end
            y = obj.State;          % Delay output
            obj.State = obj.Count;  % Put new value in state
        end
    end
end
```

The `stepImpl` method implements the System object functionality. The `setupImpl` method defines the initial values for the persistent variables in the System object.

2 Write a function that uses this System object and save it as `myDesign.m`. This function is your DUT.

```
function y = myDesign(u)

persistent obj
if isempty(obj)
    obj = CounterSysObj('Threshold',5);
end

y = step(obj, u);

end
```

3. Write a test bench that calls the DUT function and save it as `myDesign_tb.m`.

   ```
   clear myDesign
   for ii=1:10
       y = myDesign(int32(ii));
   end
   ```

4. Generate HDL code for the DUT function as you would for any other MATLAB code, but skip fixed-point conversion.

More About

- "HDL Code Generation for System Objects" on page 2-15

Map Matrices to ROM

To map a matrix constant to ROM:

- Read one matrix element at a time.
- The matrix size must be greater than or equal to the **RAM Mapping Threshold** value.

  To learn how to set the RAM mapping threshold in Simulink, see "RAMMappingThreshold" on page 12-24. To learn how to set the RAM mapping threshold in MATLAB, see "How To Enable RAM Mapping" on page 8-3.
- Read accesses to the matrix must not be within a feedback loop.

If your MATLAB code meets these requirements, HDL Coder inserts a no-reset register at the output of the matrix in the generated code. Many synthesis tools infer a ROM from this code pattern.

Fixed-Point Bitwise Functions

The following table summarizes bitwise functions in MATLAB and Fixed-Point Designer that are supported for HDL code generation. The following conventions are used in the table:

- `a,b`: Denote fixed-point integer operands.
- `idx`: Denotes an index to a bit within an operand. Indexes can be scalar or vector, depending on the function.

  MATLAB code uses 1-based indexing conventions. In generated HDL code, such indexes are converted to zero-based indexing conventions.
- `lidx, ridx`: denote indexes to the left and right boundaries delimiting bit fields. Indexes can be scalar or vector, depending on the function.
- `val`: Denotes a Boolean value.

Note: Indexes, operands, and values passed as arguments bitwise functions can be scalar or vector, depending on the function. For information on the individual functions, see “Bitwise Operations” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

MATLAB Syntax	Description	See Also
`bitand(a, b)`	Bitwise AND	`bitand`
`bitandreduce(a, lidx, ridx)`	Bitwise AND of a field of consecutive bits within `a`. The field is delimited by `lidx` , `ridx`. Output data type: `ufix1` For VHDL, generates the bitwise AND operator operating on a set of individual slices For Verilog, generates the reduce operator: `&a[lidx:ridx]`	`bitandreduce`
`bitcmp(a)`	Bitwise complement	`bitcmp`
`bitconcat(a, b)`	Concatenate fixed-point operands.	`bitconcat`

MATLAB Syntax	Description	See Also
`bitconcat([a_vector]` `bitconcat(a, b,c,d,...)`	Operands can be of different signs. Output data type: `ufixN`, where `N` is the sum of the word lengths of `a` and `b`. For VHDL, generates the concatenation operator: `(a & b)` For Verilog, generates the concatenation operator: `{a , b}`	
`bitget(a,idx)`	Access a bit at position `idx`. For VHDL, generates the slice operator: `a(idx)` For Verilog, generates the slice operator: `a[idx]`	`bitget`
`bitor(a, b)`	Bitwise OR	`bitor`
`bitorreduce(a, lidx, ridx)`	Bitwise OR of a field of consecutive bits within `a`. The field is delimited by `lidx` and `ridx`. Output data type: `ufix1` For VHDL, generates the bitwise OR operator operating on a set of individual slices. For Verilog, generates the reduce operator: `|a[lidx:ridx]`	`bitorreduce`
`bitset(a, idx, val)`	Set or clear bit(s) at position `idx`. If `val = 0`, clears the indicated bit(s). Otherwise, sets the indicated bits.	`bitset`
`bitreplicate(a, n)`	Concatenate bits of `fi` object `a` `n` times	`bitreplicate`

MATLAB Syntax	Description	See Also
`bitrol(a, idx)`	Rotate left. `idx` must be a positive integer. The value of `idx` can be greater than the word length of `a`. `idx` is normalized to `mod(idx, wlen)`. `wlen` is the word length of `a`. For VHDL, generates the `rol` operator. For Verilog, generates the following expression (where `wl` is the word length of `a`: `a << idx || a >> wl - idx`	`bitrol`
`bitror(a, idx)`	Rotate right. `idx` must be a positive integer. The value of `idx` can be greater than the word length of `a`. `idx` is normalized to `mod(idx, wlen)` . `wlen` is the word length of `a`. For VHDL, generates the `ror` operator. For Verilog, generates the following expression (where `wl` is the word length of `a`: `a >> idx || a << wl - idx`	`bitror`
`bitset(a, idx, val)`	Set or clear bit(s) at position `idx`. If `val = 0`, clears the indicated bit(s). Otherwise, sets the indicated bits.	`bitset`

MATLAB Syntax	Description	See Also
`bitshift(a, idx)`	**Note:** For efficient HDL code generation, use `bitsll`, `bitsrl`, *or* `bitsra` *instead of* `bitshift`. Shift left or right, based on the positive or negative integer value of‘`idx`. `idx` must be an integer. For positive values of `idx`, shift left `idx` bits. For negative values of `idx`, shift right `idx` bits. If `idx` is a variable, generated code contains logic for both left shift and right shift. Result values saturate if the `overflowMode` of `a` is set to `saturate`.	`bitshift`
`bitsliceget(a, lidx, ridx)`	Access consecutive set of bits from `lidx` to `ridx`. Output data type: `ufixN`, where `N = lidx-ridix+1`.	`bitsliceget`
`bitsll(a, idx)`	Shift left logical. `idx` must be a scalar within the range `0 <= idx < wl` `wl` is the word length of `a`. Overflow and rounding modes of input operand `a` are ignored. Generates `sll` operator in VHDL. Generates << operator in Verilog.	`bitsll`

MATLAB Syntax	Description	See Also
`bitsra(a, idx)`	Shift right arithmetic. `idx` must be a scalar within the range `0 <= idx < wl` `wl` is the word length of `a`, Overflow and rounding modes of input operand `a` are ignored. Generates `sra` operator in VHDL. Generates >>> operator in Verilog.	`bitsra`
`bitsrl(a, idx)`	Shift right logical. `idx` must be a scalar within the range `0 <= idx < wl` `wl` is the word length of `a`. Overflow and rounding modes of input operand `a` are ignored. Generates `srl` operator in VHDL. Generates >> operator in Verilog.	`bitsrl`
`bitxor(a, b)`	Bitwise XOR	`bitxor`

MATLAB Syntax	Description	See Also
`bitxorreduce(a, lidx, ridx)`	Bitwise XOR reduction. Bitwise XOR of a field of consecutive bits within `a`. The field is delimited by `lidx` and `ridx`. Output data type: `ufix1` For VHDL, generates a set of individual slices. For Verilog, generates the reduce operator: `^a[lidx:ridx]`	`bitxorreduce`
`getlsb(a)`	Return value of LSB.	`getlsb`
`getmsb(a)`	Return value of MSB.	`getmsb`

Related Examples

- “Bit Shifting and Bit Rotation” on page 2-41
- “Bit Slicing and Bit Concatenation” on page 2-44

More About

- “Bitwise Operations” (Fixed-Point Designer)
- “Fixed-Point Run-Time Library Functions” on page 2-31

Fixed-Point Run-Time Library Functions

HDL code generation support for fixed-point run-time library functions from the Fixed-Point Designer is summarized in the following table. See “Fixed-Point Function Limitations” on page 2-35 for general limitations of fixed-point run-time library functions for code generation.

Function	Remarks and Limitations
`abs`	Double and complex data types not supported.
`add`	—
`all`	Double data type not supported.
`any`	Double data type not supported.
`bitand`	—
`bitandreduce`	—
`bitcmp`	—
`bitconcat`	—
`bitget`	—
`bitor`	—
`bitorreduce`	—
`bitreplicate`	—
`bitrol`	—
`bitror`	—
`bitset`	—
`bitshift`	—
`bitsliceget`	—
`bitsll`	—
`bitsra`	—
`bitsrl`	—
`bitxor`	—
`bitxorreduce`	—
`ceil`	—

Function	Remarks and Limitations
`complex`	—
`conj`	—
`convergent`	—
`ctranspose`	—
`divide`	• For HDL Code generation, the divisor must be a constant and a power of two. • Non-`fi` inputs must be constant; that is, their values must be known at compile time so that they can be cast to `fi` objects. • Complex and imaginary divisors are not supported. • Code generation in MATLAB does not support the syntax `T.divide(a,b)`.
`end`	—
`eps`	• Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix, `fi` single and `fi` double signals.
`eq`	—
`fi`	—
`fimath`	—
`fix`	—
`floor`	—
`ge`	—
`getlsb`	—
`getmsb`	—
`gt`	—
`horzcat`	—
`imag`	—
`int8`, `int16`, `int32`	—
`iscolumn`	—
`isempty`	—

Function	Remarks and Limitations
isequal	—
isfi	—
isfimath	—
isfimathlocal	—
isfinite	—
isinf	—
isnan	—
isnumeric	—
isnumerictype	—
isreal	—
isrow	—
isscalar	—
issigned	—
isvector	—
le	—
length	—
logical	—
lowerbound	—
lsb	—
lt	—
max	—
min	—
minus	—
mpower	Both inputs must be scalar, and the exponent input, k, must be a constant integer.
mtimes	—
ndims	—
ne	—

Function	Remarks and Limitations
`nearest`	—
`numberofelements`	—
`numerictype`	—
`ones`	Dimensions must be real, nonnegative integers.
`plus`	Inputs cannot be data type `logical`.
`power`	Both inputs must be scalar, and the exponent input, `k`, must be a constant integer.
`range`	—
`real`	—
`realmax`	—
`realmin`	—
`reinterpretcast`	—
`repmat`	—
`rescale`	—
`reshape`	—
`round`	—
`sfi`	—
`sign`	—
`size`	—
`sqrt`	—
`sub`	—
`subsasgn`	Supported data types for HDL code generation are listed in "Supported Data Types" on page 2-2.
`subsref`	Supported data types for HDL code generation are listed in "Supported Data Types" on page 2-2.
`sum`	—
`times`	Inputs cannot be data type `logical`.
`transpose`	—

Function	Remarks and Limitations
`ufi`	—
`uint8`, `uint16`, `uint32`	—
`uminus`	—
`uplus`	Inputs cannot be data type `logical`.
`upperbound`	—
`vertcat`	—

Fixed-Point Function Limitations

In addition to function-specific limitations listed in the table, the following general limitations apply to the use of Fixed-Point Designer functions in generated HDL code:

- `fipref` and `quantizer` objects are not supported.
- Slope and bias scaling are not supported.
- Dot notation is only supported for getting the values of `fimath` and `numerictype` properties. Dot notation is not supported for `fi` objects, and it is not supported for setting properties.
- Word lengths greater than 128 bits are not supported.
- You cannot change the `fimath` or `numerictype` of a given variable after that variable has been created.
- The `boolean` and `ScaledDouble` values of the `DataTypeMode` and `DataType` properties are not supported.
- For all `SumMode` property settings other than `FullPrecision`, the `CastBeforeSum` property must be set to `true`.
- The `numel` function returns the number of elements of `fi` objects in the generated code.
- General limitations of C/C++ code generated from MATLAB apply. See “MATLAB Language Features That Code Generation Does Not Support” (Fixed-Point Designer) for more information.

Related Examples

- “Bit Shifting and Bit Rotation” on page 2-41

- “Bit Slicing and Bit Concatenation” on page 2-44

More About

- “Bitwise Operations” (Fixed-Point Designer)
- “Fixed-Point Bitwise Functions” on page 2-25

Model State with Persistent Variables and System Objects

This example shows how to use persistent variables and System objects to model state and delays in a MATLAB® design for HDL code generation.

Introduction

Using System objects to model delay results in concise generated code.

In MATLAB, multiple calls to a function having persistent variables do not result in multiple delays. Instead, the state in the function gets updated multiple times.

```
% In order to reuse code implemented in a function with states,
% you need to duplicate functions multiple times to create multiple
% instances of the algorithm with delay.
```

Examine the MATLAB Code

Let us take a quick look at the implementation of the Sobel algorithm.

Examine the design to see how the delays and line buffers are modeled using:

- Persistent variables: mlhdlc_sobel
- System objects: mlhdlc_sysobj_sobel

Notice that the 'filterdelay' function is duplicated with different function names in 'mlhdlc_sobel' code to instantiate multiple versions of the algorithm in MATLAB for HDL code generation.

The delay line implementation is more complicated when done using MATLAB persistent variables.

Now examine the simplified implementation of the same algorithm using System objects in 'mlhdlc_sysobj_sobel'.

When used within the constraints of HDL code generation, the dsp.Delay objects always map to registers. For persistent variables to be inferred as registers, you have to be careful to read the variable before writing to it to map it to a register.

MATLAB Design

```
demo_files = {...
    'mlhdlc_sysobj_sobel', ...
    'mlhdlc_sysobj_sobel_tb', ...
    'mlhdlc_sobel', ...
    'mlhdlc_sobel_tb'
    };
```

Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_delay_modeling'];

% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

for ii=1:numel(demo_files)
    copyfile(fullfile(mlhdlc_demo_dir, [demo_files{ii},'.m*']), mlhdlc_temp_dir);
end
```

Known Limitations

HDL Coder™ only supports the 'step' method of the System object and does not support 'output' and 'update' methods.

With support for only the step method, delays cannot be used in modeling feedback paths. For example, the following piece of MATLAB code cannot be supported using the dsp.Delay System object.

```
%#codegen
function y = accumulate(u)
persistent p;
if isempty(p)
   p = 0;
end
y = p;
p = p + u;
```

Create a New HDL Coder Project

To create a new project, enter the following command:

```
coder -hdlcoder -new mlhdlc_sobel
```

Next, add the file 'mlhdlc_sobel.m' to the project as the MATLAB Function and 'mlhdlc_sobel_tb.m' as the MATLAB Test Bench.

You can refer to the Getting Started with MATLAB to HDL Workflow tutorial for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Run Fixed-Point Conversion and HDL Code Generation

Launch the Workflow Advisor and right-click the 'Code Generation' step. Choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

Examine the generated HDL code by clicking the hyperlinks in the Code Generation Log window.

Now, create a new project for the system object design:

```
coder -hdlcoder -new mlhdlc_sysobj_sobel
```

Add the file 'mlhdlc_sysobj_sobel.m' to the project as the MATLAB Function and 'mlhdlc_sysobj_sobel_tb.m' as the MATLAB Test Bench.

Repeat the code generation steps and examine the generated fixed-point MATLAB and HDL code.

Additional Notes:

You can model integer delay using dsp.Delay object by setting the 'Length' property to be greater than 1. These delay objects will be mapped to shift registers in the generated code.

If the optimization option 'Map persistent array variables to RAMs' is enabled, delay System objects will get mapped to block RAMs under the following conditions:

- 'InitialConditions' property of the dsp.Delay is set to zero.
- Delay input data type is not floating-point.
- RAMSize (DelayLength * InputWordLength) is greater than or equal to the 'RAM Mapping Threshold'.

Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_delay_modeling'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

Bit Shifting and Bit Rotation

HDL Coder supports shift and rotate functions that mimic HDL-specific operators without saturation and rounding logic.

The following code implements a barrel shifter/rotator that performs a selected operation (based on the `mode` argument) on a fixed-point input operand.

```
function y   = fcn(u, mode)
% Multi Function Barrel Shifter/Rotator

% fixed width shift operation
fixed_width = uint8(3);

switch mode
    case 1
        % shift left logical
        y = bitsll(u, fixed_width);
    case 2
        % shift right logical
        y = bitsrl(u, fixed_width);
    case 3
        % shift right arithmetic
        y = bitsra(u, fixed_width);
    case 4
        % rotate left
        y = bitrol(u, fixed_width);
    case 5
        % rotate right
        y = bitror(u, fixed_width);
    otherwise
        % do nothing
        y = u;
end
```

In VHDL code generated for this function, the shift and rotate functions map directly to shift and rotate instructions in VHDL.

```
     CASE mode IS
            WHEN "00000001" =>
                -- shift left logical
                --'<S2>:1:8'
                cr := signed(u) sll 3;
                y <= std_logic_vector(cr);
            WHEN "00000010" =>
                -- shift right logical
                --'<S2>:1:11'
                b_cr := signed(u) srl 3;
                y <= std_logic_vector(b_cr);
            WHEN "00000011" =>
                -- shift right arithmetic
                --'<S2>:1:14'
                c_cr := SHIFT_RIGHT(signed(u) , 3);
                y <= std_logic_vector(c_cr);
            WHEN "00000100" =>
                -- rotate left
                --'<S2>:1:17'
                d_cr := signed(u) rol 3;
                y <= std_logic_vector(d_cr);
            WHEN "00000101" =>
                -- rotate right
                --'<S2>:1:20'
                e_cr := signed(u) ror 3;
                y <= std_logic_vector(e_cr);
            WHEN OTHERS =>
                -- do nothing
                --'<S2>:1:23'
                y <= u;
        END CASE;
```

The corresponding Verilog code is similar, except that Verilog does not have native operators for rotate instructions.

```
case ( mode)
   1 :
       begin
           // shift left logical
           //'<S2>:1:8'
           cr = u <<< 3;
           y = cr;
       end
   2 :
       begin
           // shift right logical
           //'<S2>:1:11'
           b_cr = u >> 3;
           y = b_cr;
       end
   3 :
```

```
                begin
                    // shift right arithmetic
                    //'<S2>:1:14'
                    c_cr = u >>> 3;
                    y = c_cr;
                end
            4 :
                begin
                    // rotate left
                    //'<S2>:1:17'
                    d_cr = {u[12:0], u[15:13]};
                    y = d_cr;
                end
            5 :
                begin
                    // rotate right
                    //'<S2>:1:20'
                    e_cr = {u[2:0], u[15:3]};
                    y = e_cr;
                end
            default :
                begin
                    // do nothing
                    //'<S2>:1:23'
                    y = u;
                end
        endcase
```

Related Examples

- "Bit Slicing and Bit Concatenation" on page 2-44

More About

- "Bitwise Operations" (Fixed-Point Designer)
- "Fixed-Point Bitwise Functions" on page 2-25
- "Fixed-Point Run-Time Library Functions" on page 2-31

Bit Slicing and Bit Concatenation

This section describes how to use the functions `bitsliceget` and `bitconcat` to access and manipulate bit slices (fields) in a fixed-point or integer word. As an example, consider the operation of swapping the upper and lower 4-bit nibbles of an 8-bit byte. The following example accomplishes this task without resorting to traditional mask-and-shift techniques.

```
function y = fcn(u)
% NIBBLE SWAP
y = bitconcat( …
      bitsliceget(u, 4, 1),
      bitsliceget(u, 8, 5));
```

The `bitsliceget` and `bitconcat` functions map directly to slice and concat operators in both VHDL and Verilog.

The following listing shows the corresponding generated VHDL code.

```
ENTITY fcn IS
    PORT (
        clk : IN std_logic;
        clk_enable : IN std_logic;
        reset : IN std_logic;
        u : IN std_logic_vector(7 DOWNTO 0);
        y : OUT std_logic_vector(7 DOWNTO 0));
END nibble_swap_7b;


ARCHITECTURE fsm_SFHDL OF fcn IS


BEGIN
    -- NIBBLE SWAP
    y <= u(3 DOWNTO 0) & u(7 DOWNTO 4);
END fsm_SFHDL;
```

The following listing shows the corresponding generated Verilog code.

```
module fcn (clk, clk_enable, reset, u, y );
    input clk;
    input clk_enable;
    input reset;
    input [7:0] u;
    output [7:0] y;

    // NIBBLE SWAP
    assign y = {u[3:0], u[7:4]};

endmodule
```

Related Examples

- “Bit Shifting and Bit Rotation” on page 2-41

More About

- “Bitwise Operations” (Fixed-Point Designer)
- “Fixed-Point Bitwise Functions” on page 2-25
- “Fixed-Point Run-Time Library Functions” on page 2-31

Guidelines for Efficient HDL Code

When you generate HDL code from your MATLAB design, you are converting an algorithm into an architecture that must meet hardware area and speed requirements.

For better HDL code and faster code generation, design your MATLAB code according to the following best practices:

- *Serialize your input and output data.* Parallel data processing structures require more hardware resources and a higher pin count.
- *Use add and subtract algorithms instead of algorithms that use functions like sin, divide, and modulo.* Add and subtract operations use fewer hardware resources.
- *Avoid large arrays and matrices.* Large arrays and matrices require more registers and RAM for storage.
- *Convert your code from floating-point to fixed-point.* Floating-point data types are inefficient for hardware realization. HDL Coder provides an automated workflow for floating-point to fixed-point conversion.
- *Unroll loops.* Unroll loops to increase speed at the cost of higher area; unroll fewer loops and enable the loop streaming optimization to conserve area at the cost of lower throughput.

MATLAB Design Requirements for HDL Code Generation

Your MATLAB design has the following requirements:

- MATLAB code within the design must be supported for HDL code generation.
- Inputs and outputs must not be matrices or structures.

If you are generating code from the command line, verify your code readiness for code generation with the following command:

```
coder.screener(‘design_function_name’)
```

If you use the HDL Workflow Advisor to generate code, this check runs automatically.

For a MATLAB language support reference, including supported functions from the Fixed-Point Designer, see “MATLAB Algorithm Design”.

What Is a MATLAB Test Bench?

A test bench is a MATLAB script or function that you write to test the algorithm in your MATLAB design function. The test bench varies the input data to the design to simulate real world conditions. It can also can check that the output data meets design specifications.

HDL Coder uses the data it gathers from running your test bench with your design to infer fixed-point data types for floating-point to fixed-point conversion. The coder also uses the data to generate HDL test data for verifying your generated code. For more information on how to write your test bench for the best results, see “MATLAB Test Bench Requirements and Best Practices” on page 2-49.

MATLAB Test Bench Requirements and Best Practices

MATLAB Test Bench Requirements

You can use any MATLAB data type and function in your test bench.

A MATLAB test bench has the following requirements:

- For floating-point to fixed-point conversion, the test bench must be a script or a function with no inputs.
- The inputs and outputs in your MATLAB design interface must use the same data types, sizes, and complexity in each call site in your test bench.
- If you enable the **Accelerate test bench for faster simulation** option in the Float-to-Fixed Workflow, the MATLAB constructs in your test bench loop must be compilable.

MATLAB Test Bench Best Practices

Use the following MATLAB test bench best practices:

- *Design your test bench to cover the full numeric range of data that the design must handle.* HDL Coder uses the data that it accumulates from running the test bench to infer fixed-point data types during floating-point to fixed-point conversion.

  If you call the design function multiple times from your test bench, the coder uses the accumulated data from each instance to infer fixed-point types. Both the design and the test bench can call local functions within the file or other functions on the MATLAB path. The call to the design function can be at any level of your test bench hierarchy.
- *Before trying to generate code, run your test bench in MATLAB* . If simulation is slow, accelerate your test bench. To learn how to accelerate your simulation, see “Accelerate MATLAB Algorithms” (MATLAB Coder).
- If you have a loop that calls your design function, use only compilable MATLAB constructs within the loop and enable the **Accelerate test bench for faster simulation** option.
- Before each test bench simulation run, use the `clear variables` command to reset your persistent variables.

To see an example of a test bench, enter this command:

```
showdemo mlhdlc_tutorial_float2fixed_files
```

MATLAB Best Practices and Design Patterns for HDL Code Generation

Model a Counter for HDL Code Generation

In this section...

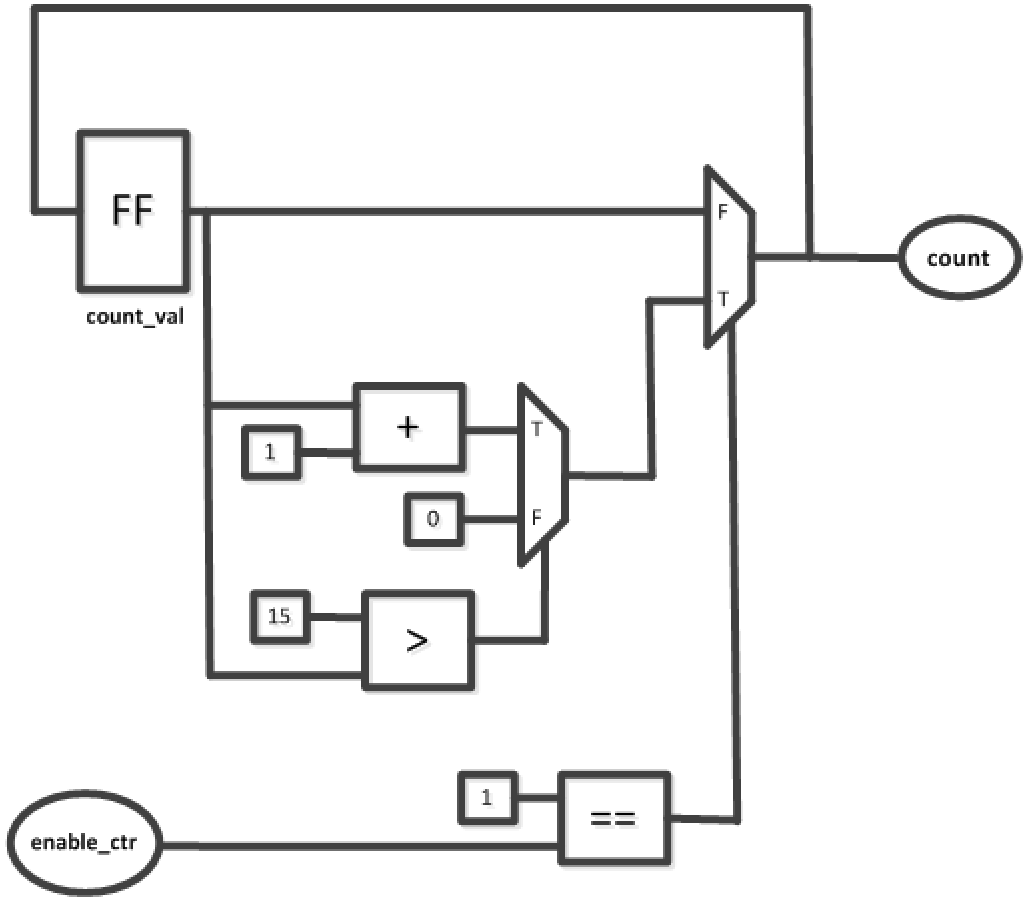
MATLAB Counter

This design pattern shows a MATLAB example of a counter, which is suitable for HDL code generation.

This model demonstrates the following best practices for writing MATLAB code to generate HDL code:

- Initialize persistent variables.
- Read persistent variables before they are modified.

The schematic below shows the counter modeled in this example.



MATLAB Code for the Counter

The function `mlhdlc_counter` is a behavioral model of a four bit synchronous up counter. The input signal, `enable_ctr`, triggers the value of the count register, `count_val`, to increase by one. The counter continues to increase by one each time the input is nonzero, until the count reaches a limit of 15. After the counter reaches this limit, the counter returns to zero. A persistent variable, which is initialized to zero, represents the current value of the count. Two `if` statements determine the value of the count based on the input.

The following section of code defines the `mldhlc_counter` function.

```
%#codegen
function count = mlhdlc_counter(enable_ctr)
%four bit synchronous up counter

%persistent variable for the state
persistent count_val;
if isempty(count_val)
    count_val = 0;
end

%counting up
if enable_ctr
    count_val=count_val+1;

    %limit to four bits
    if count_val>15
        count_val=0;
    end
end

count=count_val;

end
```

Best Practices in this Example

This design pattern demonstrates two best practices for writing MATLAB code for HDL code generation:

- Initialize persistent variables to a specific value. In this example, an `if` statement and the `isempty` function initialize the persistent variable. If the persistent variable is not initialized then HDL code cannot be generated.
- Inside a function, read persistent variables before they are modified, in order for the persistent variables to be inferred as registers.

Model a State Machine for HDL Code Generation

In this section...

MATLAB State Machines

The following design pattern shows MATLAB examples of Mealy and Moore state machines which are suitable for HDL code generation.

The MATLAB code in these models demonstrates best practices for writing MATLAB models for HDL code generation.

- With a `switch` block, use the `otherwise` statement to account for all conditions.
- Use variables to designate states in a state machine.

In a Mealy state machine, the output depends on the state and the input. In a Moore state machine, the output depends only on the state.

MATLAB Code for the Mealy State Machine

The following MATLAB code defines the `mlhdlc_fsm_mealy` function. A persistent variable represents the current state. A `switch` block uses the current state and input to determine the output and new state. In each `case` in the `switch` block, an `if-else` statement calculates the new state and output.

```
%#codegen
function Z = mlhdlc_fsm_mealy(A)
% Mealy State Machine

% y = f(x,u) :
% all actions are condition actions and
% outputs are function of state and input

% define states
S1 = 0;
```

```
S2 = 1;
S3 = 2;
S4 = 3;

persistent current_state;
if isempty(current_state)
    current_state = S1;
end

% switch to new state based on the value state register
switch (current_state)

    case S1,

        % value of output 'Z' depends both on state and inputs
        if (A)
            Z = true;
            current_state = S1;
        else
            Z = false;
            current_state = S2;
        end

    case S2,

        if (A)
            Z = false;
            current_state = S1;
        else
            Z = true;
            current_state = S2;
        end

    case S3,

        if (A)
            Z = false;
            current_state = S2;
        else
            Z = true;
            current_state = S3;
        end

    case S4,
        if (A)
            Z = true;
            current_state = S1;
        else
            Z = false;
            current_state = S3;
        end

    otherwise,

        Z = false;
end
```

MATLAB Code for the Moore State Machine

The following MATLAB code defines the `mlhdlc_fsm_moore` function. A persistent variable represents the current state, and a `switch` block uses the current state to determine the output and new state. In each `case` in the `switch` block, an `if-else` statement calculates the new state and output. The value of the state is represented by numerical variables.

```
%#codegen
function Z = mlhdlc_fsm_moore(A)
% Moore State Machine

% y = f(x) :
% all actions are state actions and
% outputs are pure functions of state only

% define states
S1 = 0;
S2 = 1;
S3 = 2;
S4 = 3;


% using persistent keyword to model state registers in hardware
persistent curr_state;
if isempty(curr_state)
    curr_state = S1;
end

% switch to new state based on the value state register
```

```
switch (curr_state)

    case S1,

        % value of output 'Z' depends only on state and not on inputs
        Z = true;

        % decide next state value based on inputs
        if (~A)
            curr_state = S1;
        else
            curr_state = S2;
        end

    case S2,

        Z = false;

        if (~A)
            curr_state = S1;
        else
            curr_state = S2;
        end

    case S3,

        Z = false;

        if (~A)
            curr_state = S2;
        else
            curr_state = S3;
        end

    case S4,

        Z = true;
        if (~A)
            curr_state = S1;
        else
            curr_state = S3;
        end

    otherwise,
        Z = false;
end
```

Best Practices

This design pattern demonstrates two best practices for writing MATLAB code for HDL code generation.

- With a `switch` block, use the `otherwise` statement to ensure that the model accounts for all conditions. If the model does not cover all conditions, the generated HDL code can contain errors.
- To designate the states in a state machine, use variables with numerical values.

Generate Hardware Instances For Local Functions

In this section...

MATLAB Local Functions

The following example shows how to use local functions in MATLAB, so that each execution of a local function corresponds to a separate hardware module in the generated HDL code. This example demonstrates best practices for writing local functions in MATLAB code that is suitable for HDL code generation.

- If your MATLAB code executes a local function multiple times, the generated HDL code does not necessarily instantiate multiple hardware modules. Rather than instantiating multiple hardware modules, multiple calls to a function typically update the state variable.
- If you want the generated HDL code to contain multiple hardware modules corresponding to each execution of a local function, specify two different local functions with the same code but different function names. If you want to avoid code duplication, consider using System objects to implement the behavior in the function, and instantiate the System object multiple times.
- If you want to specify a separate HDL file for each local function in the MATLAB code, in the Workflow Advisor, on the **Advanced** tab in the HDL Code Generation section, select **Generate instantiable code for functions** .

MATLAB Code for `mlhdlc_two_counters.m`

This function creates two counters and adds the output of these counters. To create two counters, there are two local functions with identical code, `counter` and `counter2`. The main method calls each of these local functions once. If the function were to call the `counter` function twice, separate hardware modules for the counters would not be generated in the HDL code.

```
%#codegen
function total_count = mlhdlc_two_counters(a,b)

%This function contains a two different local functions with identical
```

```
%counters and calls each counter once.

total_count1=counter(a);

total_count2=counter2(b);

total_count=total_count1+total_count2;

function count = counter(enable_ctr)
%four bit synchronous up counter

%persistent variable for the state
persistent count_val;
if isempty(count_val)
    count_val = 0;
end

%counting up
if enable_ctr
    count_val=count_val+1;
end

%limit from four bits
if count_val>15
    count_val=0;
end

count=count_val;



function count = counter2(enable_ctr)
%four bit synchronous up counter

%persistent variable for the state
persistent count_val;
if isempty(count_val)
    count_val = 0;
end

%counting up
if enable_ctr
    count_val=count_val+1;
end
```

```
%limit from four bits
if count_val>15
    count_val=0;
end

count=count_val;
```

Implement RAM Using MATLAB Code

In this section...

Implementation of RAM

You can write MATLAB code that maps to RAM during HDL code generation by using:

- Persistent arrays or private properties in a user-defined System object.
- `hdl.RAM` System objects.

The following examples model the same line delay in MATLAB. However, one example uses a persistent array and the other uses an `hdl.RAM` System object to model the RAM behavior.

The line delay uses memory in a ring structure. Data is written to one location and read from another location in such a way that the data written is read after a delay of a specific number of cycles. The RAM read address is generated by a counter. The write address is generated by adding a constant value to the read address.

For a comparison of the ways you can write MATLAB code to map to RAM during HDL code generation, and for an overview of the tradeoffs, see "RAM Mapping Comparison for MATLAB Code" on page 8-8. For more information, see "Map Persistent Arrays and dsp.Delay to RAM" on page 8-3.

Implement RAM Using a Persistent Array or System object Properties

This example shows a line delay that implements the RAM behavior using a persistent array with the function `mlhdlc_hdlram_persistent`. Changing a specific value in the persistent array is equivalent to writing to the RAM. Accessing a specific value in the array is equivalent to reading from the RAM.

You can implement RAM by using user-defined System object private properties in the same way.

```
%#codegen
function data_out = mlhdlc_hdlram_persistent(data_in)

persistent hRam;
if isempty(hRam)
    hRam = zeros(128,1);
end

% read address counter
persistent rdAddrCtr;
if isempty(rdAddrCtr)
    rdAddrCtr = 1;
end

% ring counter length
ringCtrLength = 10;
ramWriteAddr = rdAddrCtr + ringCtrLength;

ramWriteData = data_in;
%ramWriteEnable = true;

ramReadAddr = rdAddrCtr;

% execute single step of RAM

hRam(ramWriteAddr)=ramWriteData;
ramRdDout=hRam(ramReadAddr);

rdAddrCtr = rdAddrCtr + 1;

data_out = ramRdDout;
```

Implement RAM Using `hdl.RAM`

This example shows a line delay that implements the RAM behavior using `hdl.RAM` with the function, `mlhdlc_hdlram_sysobj`. In this function, the `step` method of the `hdl.RAM` System object reads and writes to specific locations in `hRam`.

```
%#codegen
function data_out = mlhdlc_hdlram_sysobj(data_in)
persistent hRam;
if isempty(hRam)
    hRam = hdl.RAM('RAMType', 'Dual port');
end
```

```
% read address counter
persistent rdAddrCtr;
if isempty(rdAddrCtr)
    rdAddrCtr = 0;
end

% ring counter length
ringCtrLength = 10;
ramWriteAddr = rdAddrCtr + ringCtrLength;

ramWriteData = data_in;
ramWriteEnable = true;

ramReadAddr = rdAddrCtr;

% execute single step of RAM
[~,ramRdDout] = step(hRam,ramWriteData,ramWriteAddr, ...
                      ramWriteEnable,ramReadAddr);

rdAddrCtr = rdAddrCtr + 1;

data_out = ramRdDout;
```

`hdl.RAM` Restrictions for Code Generation

Code generation from `hdl.RAM` has the same restrictions as code generation from other System objects. For details, see "Limitations of HDL Code Generation for System Objects" on page 2-16.

For-Loop Best Practices for HDL Code Generation

In this section...

MATLAB Loops

Some best practices for using loops in MATLAB code for HDL code generation are:

- Use monotonically increasing loop counters, with increments of 1, to minimize the amount of hardware generated in the HDL code.
- If you want to use the loop streaming optimization:
  - When assigning new values to persistent variables inside a loop, do not use other persistent variables on the right side of the assignment. Instead, use an intermediate variable.
  - If a loop modifies any elements in a persistent array, the loop should modify all of the elements in the persistent array.

Monotonically Increasing Loop Counters

By using monotonically increasing loop counters with increments of 1, you can reduce the amount of hardware in the generated HDL code. The following loop is an example of a monotonically increasing loop counter with increments of 1.

```
a=1;
for i=1:10
    a=a+1;
end
```

If a loop counter increases by an increment other than 1, the generated HDL code can require additional adders. Due to this additional hardware, do not use the following type of loop.

```
a=1;
for i=1:2:10
    a=a+1;
```

```
end
```

If a loop counter decreases, the generated HDL code can require additional adders. Due to this additional hardware, do not use the following type of loop.

```
a=1;
for i=10:-1:1
    a=a+1;
end
```

Persistent Variables in Loops

If a loop contains multiple persistent variables, when you assign values to persistent variables, use intermediate variables that are not persistent on the right side of the assignment. This practice makes dependencies clear to the compiler and assists internal optimizations during the HDL code generation process. If you want to use the loop streaming optimization to reduce the amount of generated hardware, this practice is recommended.

In the following example, `var1` and `var2` are persistent variables. `var1` is used on the right side of the assignment. Because a persistent variable is on the right side of an assignment, do not use this type of loop:

```
for i=1:10
    var1 = 1 + i;
    var2 = var1 * 2;
end
```

Instead of using `var1` on the right side of the assignment, use an intermediate variable that is not persistent. This example demonstrates this with the intermediate variable `var_intermediate`.

```
for i=1:10
    var_intermediate = 1 + i;
    var1 = var_intermediate;
    var2 = var_intermediate * 2;
end
```

Persistent Arrays in Loops

If a loop modifies elements in a persistent array, make sure that the loop modifies all of the elements in the persistent array. If all elements of the persistent array are not modified within the loop, HDL Coder cannot perform the loop streaming optimization.

In the following example, `a` is a persistent array. The first element is modified outside of the loop. Do not use this type of loop.

```
for i=2:10
    a(i)=1+i;
 end
 a(1)=24;
```

Rather than modifying the first element outside the loop, modify all of the elements inside the loop.

```
for i=1:10
   if i==1
      a(i)=24;
   else
      a(i)=1+i;
   end
end
```

Fixed-Point Conversion

Floating-Point to Fixed-Point Conversion

This example shows how to start with a floating-point design in MATLAB, iteratively converge on an efficient fixed-point design in MATLAB, and verify the numerical accuracy of the generated fixed-point design.

Signal processing applications for reconfigurable platforms require algorithms that are typically specified using floating-point operations. However, for power, cost, and performance reasons, they are usually implemented with fixed-point operations either in software for DSP cores or as special-purpose hardware in FPGAs. Fixed-point conversion can be very challenging and time-consuming, typically demanding 25 to 50 percent of the total design and implementation time. Automated tools can simplify and accelerate the conversion process.

For software implementations, the aim is to define an optimized fixed-point specification which minimizes the code size and the execution time for a given computation accuracy constraint. This optimization is achieved through the modification of the binary point location (for scaling) and the selection of the data word length according to the different data types supported by the target processor.

For hardware implementations, the complete architecture can be optimized. An efficient implementation will minimize both the area used and the power consumption. Thus, the conversion process goal typically is focused around minimizing the operator word length.

The floating-point to fixed-point workflow is currently integrated in the HDL Workflow Advisor that you have been introduced to in the tutorial Getting Started with MATLAB to HDL Workflow.

Introduction

The floating-point to fixed-point conversion workflow in HDL Coder™ includes the following steps:

1. Verify that the floating-point design is compatible with code generation.
2. Compute fixed-point types based on the simulation of the testbench.
3. Generate readable and traceable fixed-point MATLAB code by applying proposed types.
4. Verify the generated fixed-point design.
5. Compare the numerical accuracy of the generated fixed-point code with the original floating point code.

MATLAB Design

The MATLAB code used in this example is a simple second-order direct-form 2 transposed filter. This example also contains a MATLAB testbench that exercises the filter.

```
design_name = 'mlhdlc_df2t_filter';
testbench_name = 'mlhdlc_df2t_filter_tb';
```

Examine the MATLAB design.

```
type(design_name);
```

```
%#codegen
function y = mlhdlc_df2t_filter(x)

%   Copyright 2011-2015 The MathWorks, Inc.

persistent z;
if isempty(z)
    % Filter states as a column vector
    z = zeros(2,1);
end

% Filter coefficients as constants
b = [0.29290771484375   0.585784912109375  0.292907714843750];
a = [1.0                0.0                0.171600341796875];

y    =  b(1)*x + z(1);
z(1) = (b(2)*x + z(2)) - a(2) * y;
z(2) =  b(3)*x - a(3) * y;

end
```

For the floating-point to fixed-point workflow, it is desirable to have a complete testbench. The quality of the proposed fixed-point data types depends on how well the testbench covers the dynamic range of the design with the desired accuracy.

For more details on the requirements for the floating-point design and the testbench, refer to the 'Floating-Point Design Structure' structure section of the Working with Generated Fixed-Point Files tutorial.

```
type(testbench_name);
```

```
%   Copyright 2011-2015 The MathWorks, Inc.


Fs = 256;              % Sampling frequency
Ts = 1/Fs;             % Sample time
t = 0:Ts:1-Ts;         % Time vector from 0 to 1 second
f1 = Fs/2;             % Target frequency of chirp set to Nyquist
in = sin(pi*f1*t.^2);  % Linear chirp from 0 to Fs/2 Hz in 1 second
out = zeros(size(in)); % Output the same size as the input

for ii=1:length(in)
    out(ii) = mlhdlc_df2t_filter(in(ii));
end

% Plot
figure('Name', [mfilename, '_plot']);
subplot(2,1,1);
plot(in);
xlabel('Time')
ylabel('Amplitude')
title('Input Signal (with Noise)')

subplot(2,1,2);
plot(out);
xlabel('Time')
ylabel('Amplitude')
title('Output Signal (filtered)')
```

Create a New Folder and Copy Relevant Files

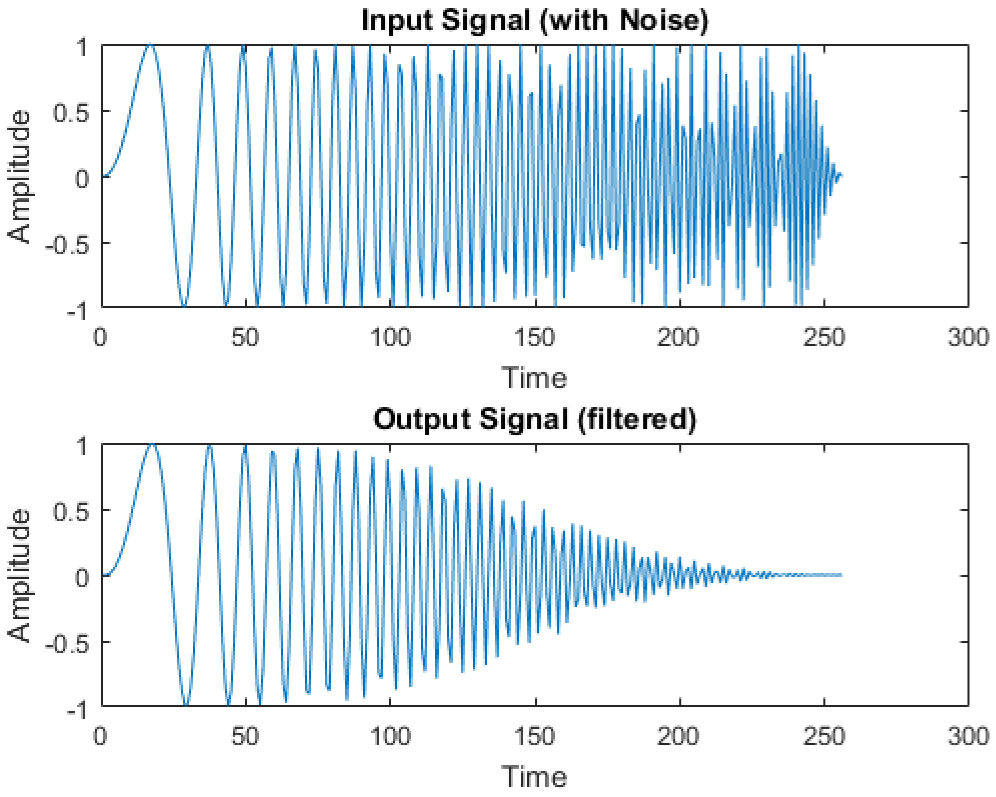
Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_flt2fix_prj'];

% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name,'.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name,'.m*']), mlhdlc_temp_dir);
```

Simulate the Design

Simulate the design with the test bench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_df2t_filter_tb
```



Create a New HDL Coder Project

To create a new project, enter the following command:

```
coder -hdlcoder -new flt2fix_project
```

Next, add the file 'mlhdlc_filter.m' to the project as the MATLAB Function and 'mlhdlc_filter_tb.m' as the MATLAB Test Bench.

You can refer to Getting Started with MATLAB to HDL Workflow tutorial for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

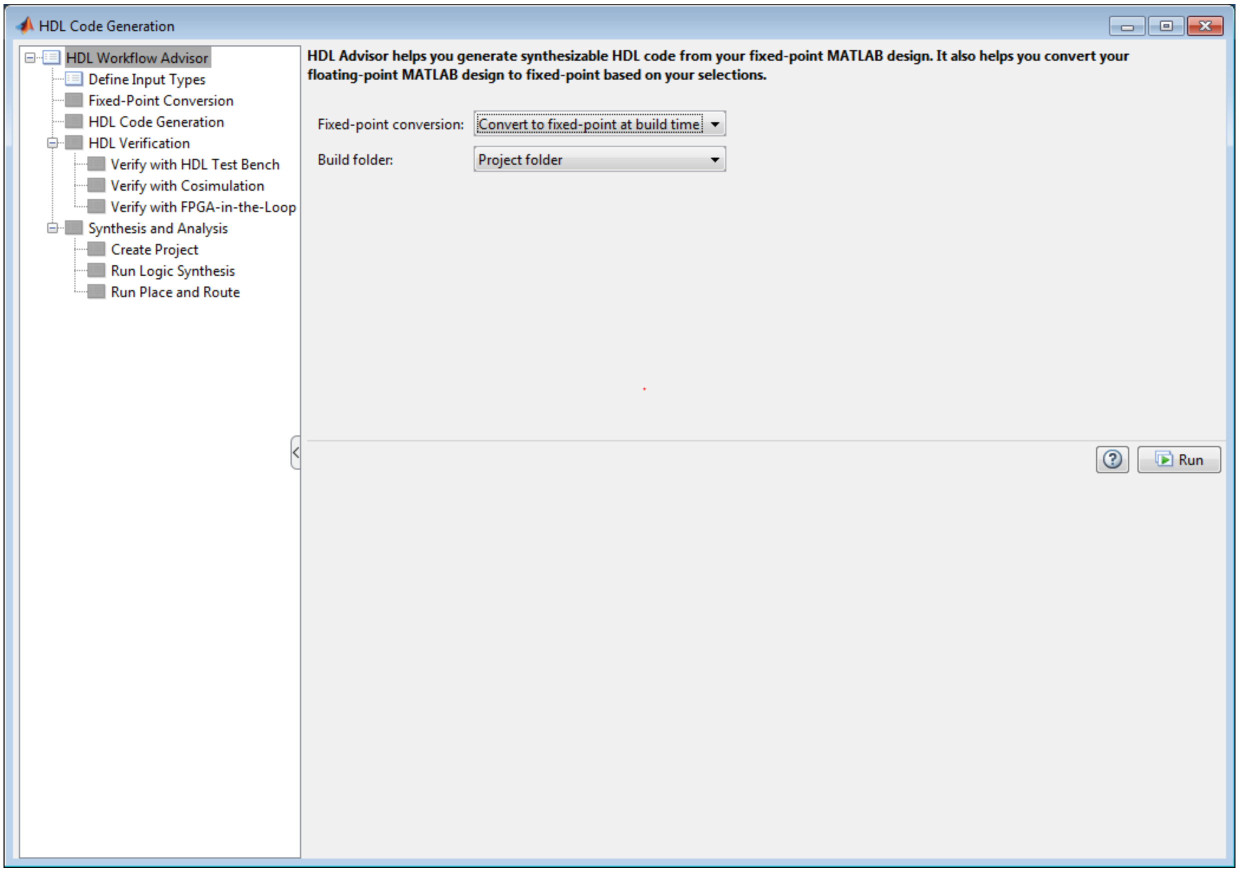
Fixed-Point Code Generation Workflow

The floating-point to fixed-point conversion workflow allows you to:

- Verify that the floating-point design is code generation compliant
- Propose fixed-point types based on simulation data and word length settings
- Allow the user to manually adjust the proposed fixed-point types
- Validate the proposed fixed-point types
- Verify that the generated fixed-point MATLAB code has the desired numeric accuracy

Step 1: Launch Workflow Advisor

1. Click on the Workflow Advisor button to launch the HDL Workflow Advisor.
2. Choose 'Convert to fixed-point at build time' for the option 'Fixed-point conversion'.



HDL Advisor helps you generate synthesizable HDL code from your fixed-point MATLAB design. It also helps you convert your floating-point MATLAB design to fixed-point based on your selections.

Fixed-point conversion: Convert to fixed-point at build time

Build folder: Project folder

Step 2: Define Input Types

In this step you can define input types manually or by specifying and running the testbench.

1 Click 'Run' to execute this step.

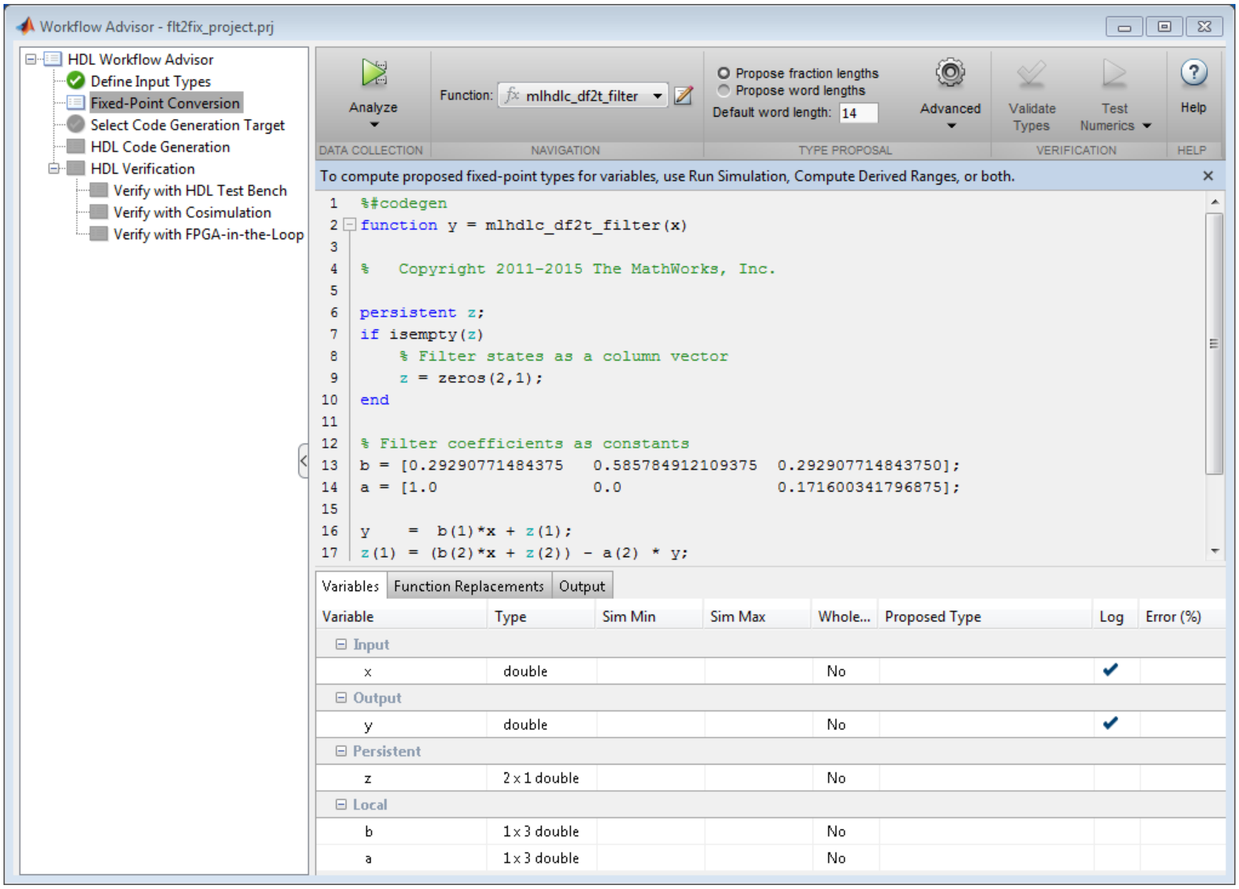
After simulation notice that the input variable 'x' is defined as scalar double 'double(1x1)'

Step 3: Run Simulation

1 Click on the 'Fixed-Point Conversion' step.

The design is compiled with the input types defined in the previous step and after the compilation is successful the variable table shows inferred types for all the functions in the design.

In this step, the original design is instrumented so that the minimum and maximum values for all variables in the design are collected during simulation.



Function: mlhdlc_df2t_filter

[✓ Propose fraction lengths] Propose word lengths

Default word length: 14

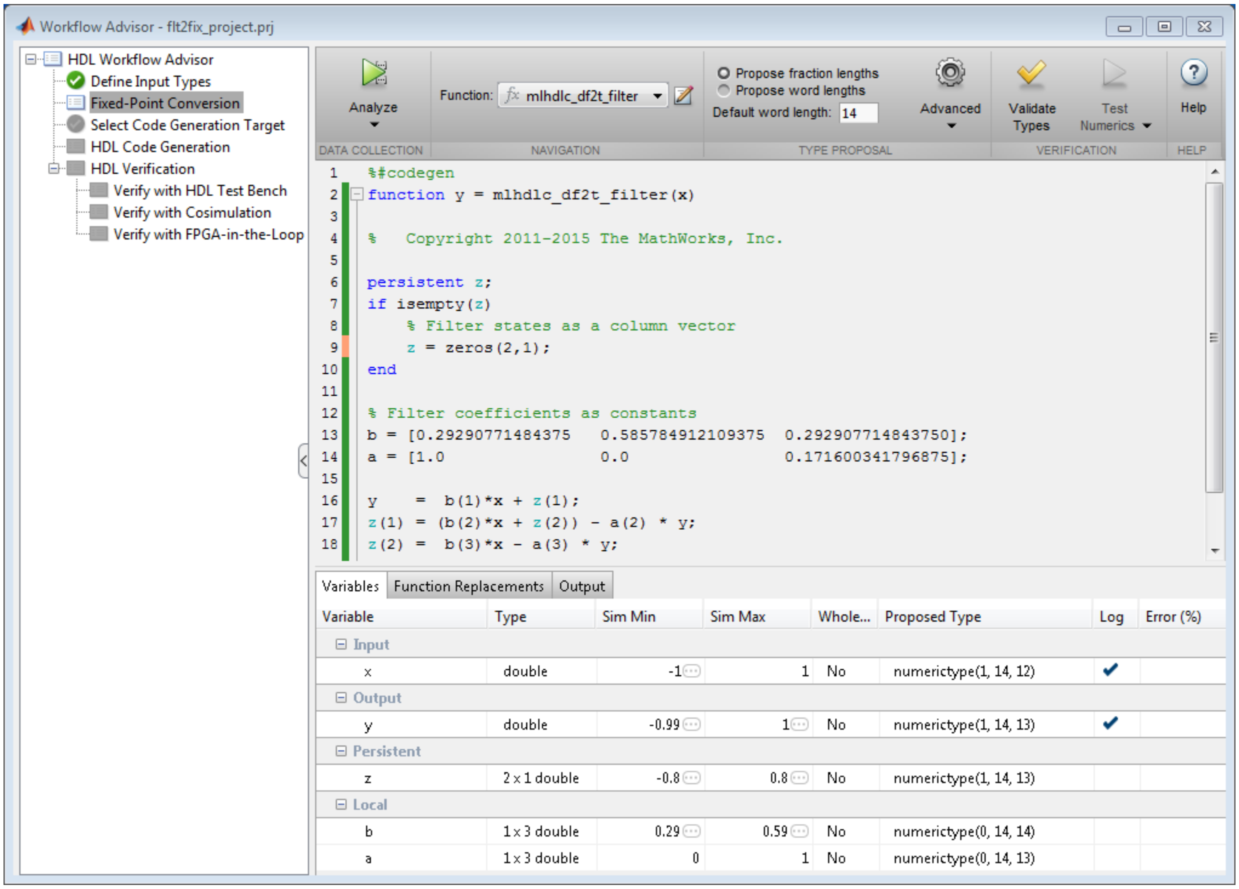
To compute proposed fixed-point types for variables, use Run Simulation, Compute Derived Ranges, or both.

```
%#codegen
function y = mlhdlc_df2t_filter(x)

%   Copyright 2011-2015 The MathWorks, Inc.

persistent z;
if isempty(z)
    % Filter states as a column vector
    z = zeros(2,1);
end

% Filter coefficients as constants
b = [0.29290771484375   0.585784912109375   0.292907714843750];
a = [1.0                0.0                 0.171600341796875];

y    =  b(1)*x + z(1);
z(1) =  (b(2)*x + z(2)) - a(2) * y;
```

Variable	Type	Sim Min	Sim Max	Whole...	Proposed Type	Log	Error (%)
⊟ Input							
x	double			No		✓	
⊟ Output							
y	double			No		✓	
⊟ Persistent							
z	2 x 1 double			No			
⊟ Local							
b	1 x 3 double			No			
a	1 x 3 double			No			

1 Click on the 'Analyze' button.

Notice that the 'Sim Min' and 'Sim Max' table is now populated with simulation ranges. Fixed-point types are proposed based on the default word length settings.



Function: mlhdlc_df2t_filter

[✓ Propose fraction lengths] Propose word lengths

Default word length: 14

```
%#codegen
function y = mlhdlc_df2t_filter(x)

%   Copyright 2011-2015 The MathWorks, Inc.

persistent z;
if isempty(z)
    % Filter states as a column vector
    z = zeros(2,1);
end

% Filter coefficients as constants
b = [0.29290771484375   0.585784912109375   0.292907714843750];
a = [1.0                0.0                 0.171600341796875];

y    =  b(1)*x + z(1);
z(1) = (b(2)*x + z(2)) - a(2) * y;
z(2) =  b(3)*x - a(3) * y;
```

Variable	Type	Sim Min	Sim Max	Whole...	Proposed Type	Log	Error (%)
⊟ Input							
x	double	-1	1	No	numerictype(1, 14, 12)	✓	
⊟ Output							
y	double	-0.99	1	No	numerictype(1, 14, 13)	✓	
⊟ Persistent							
z	2 x 1 double	-0.8	0.8	No	numerictype(1, 14, 13)		
⊟ Local							
b	1 x 3 double	0.29	0.59	No	numerictype(0, 14, 14)		
a	1 x 3 double	0	1	No	numerictype(0, 14, 13)		

At this stage, based on computed simulation ranges for all variables, you can compute:

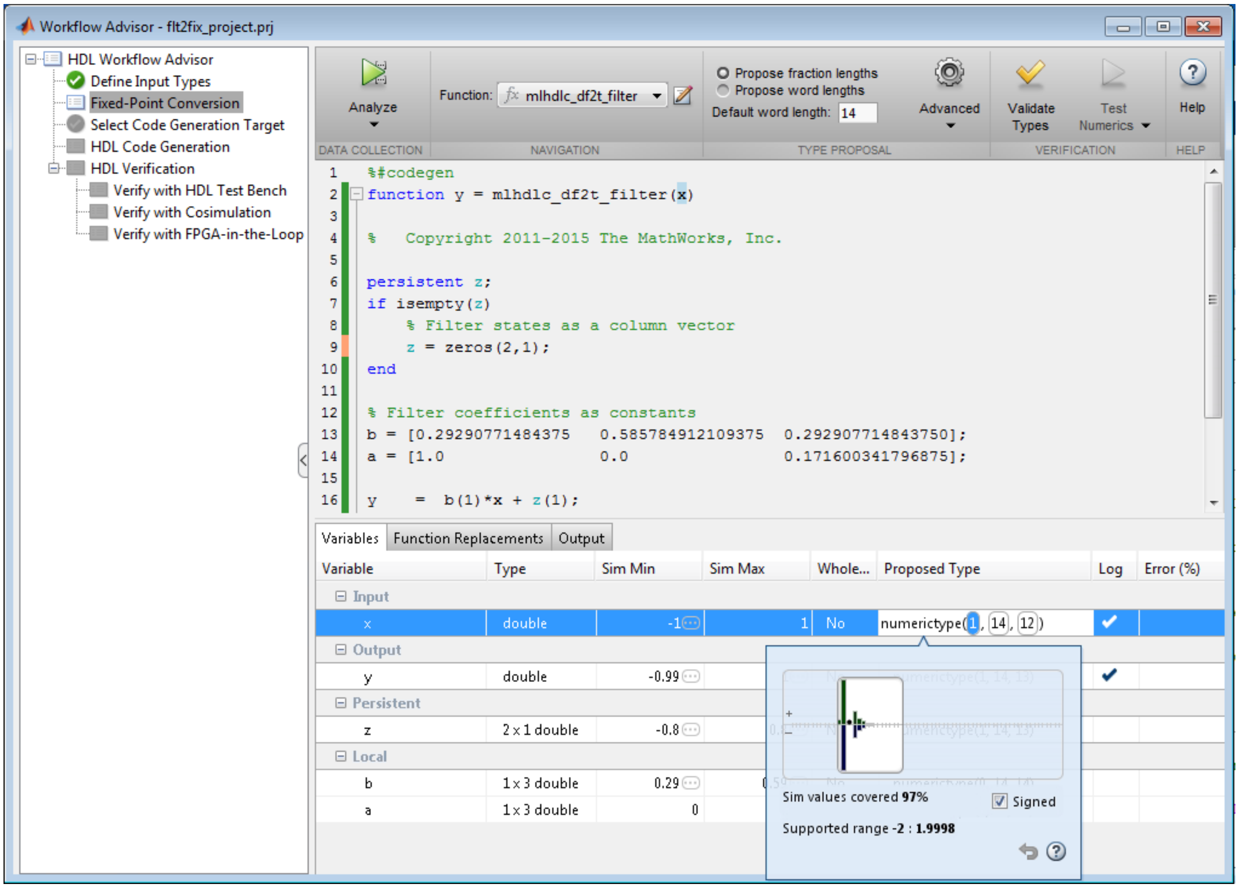
- Fraction lengths for a given fixed word length setting, or
- Word lengths for a given fixed fraction length setting.

The type table contains the following information for each variable existing in the floating-point MATLAB design, organized by function:

- Sim Min: The minimum value assigned to the variable during simulation.
- Sim Max: The maximum value assigned to the variable during simulation.
- Whole Number: Whether all values assigned during simulation are integers.

The type proposal step uses the above information and combines it with the user-specified word length settings to propose a fixed-point type for each variable.

You can also enable the 'Log histogram data' option in the 'Analyze' button's menu to enable logging of histogram data.



Function: mlhdlc_df2t_filter

[✓ Propose fraction lengths] Propose word lengths

Default word length: 14

```
%#codegen
function y = mlhdlc_df2t_filter(x)

%   Copyright 2011-2015 The MathWorks, Inc.

persistent z;
if isempty(z)
    % Filter states as a column vector
    z = zeros(2,1);
end

% Filter coefficients as constants
b = [0.29290771484375   0.585784912109375   0.292907714843750];
a = [1.0                0.0                 0.171600341796875];

y    =  b(1)*x + z(1);
```

Variable	Type	Sim Min	Sim Max	Whole...	Proposed Type	Log	Error (%)
Input							
x	double	-1	1	No	numerictype(1, 14, 12)	✓	
Output							
y	double	-0.99				✓	
Persistent							
z	2 x 1 double	-0.8					
Local							
b	1 x 3 double	0.29					
a	1 x 3 double	0					

Sim values covered **97%**

[✓ Signed]

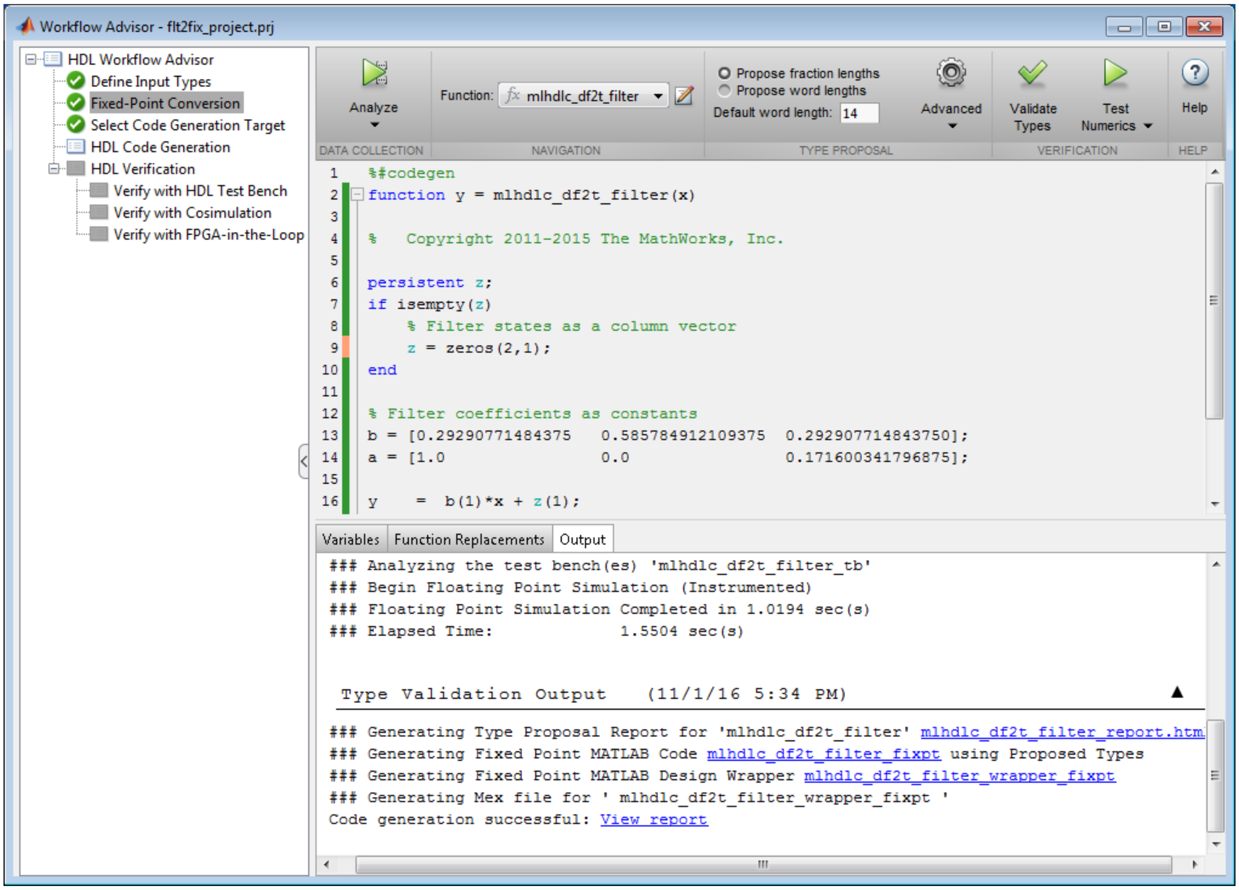
Supported range **-2** : **1.9998**

The histogram view concisely gives information about dynamic range of the simulation data for a variable. The x-axis correspond to bit weights and y-axis represents number of occurrences. The proposed numeric type information is overlaid on top of this graph and is editable. Moving the bounding white box left or right changes the position of binary point. Moving the right or left edges correspondingly change fraction length or wordlength. All the changes made to the proposed type are saved in the project.

Step 4: Validate types

In this step, the fixed-point types from the previous step are used to generate a fixed-point MATLAB design from the original floating-point implementation.

1 Click on the 'Validate Types' button.



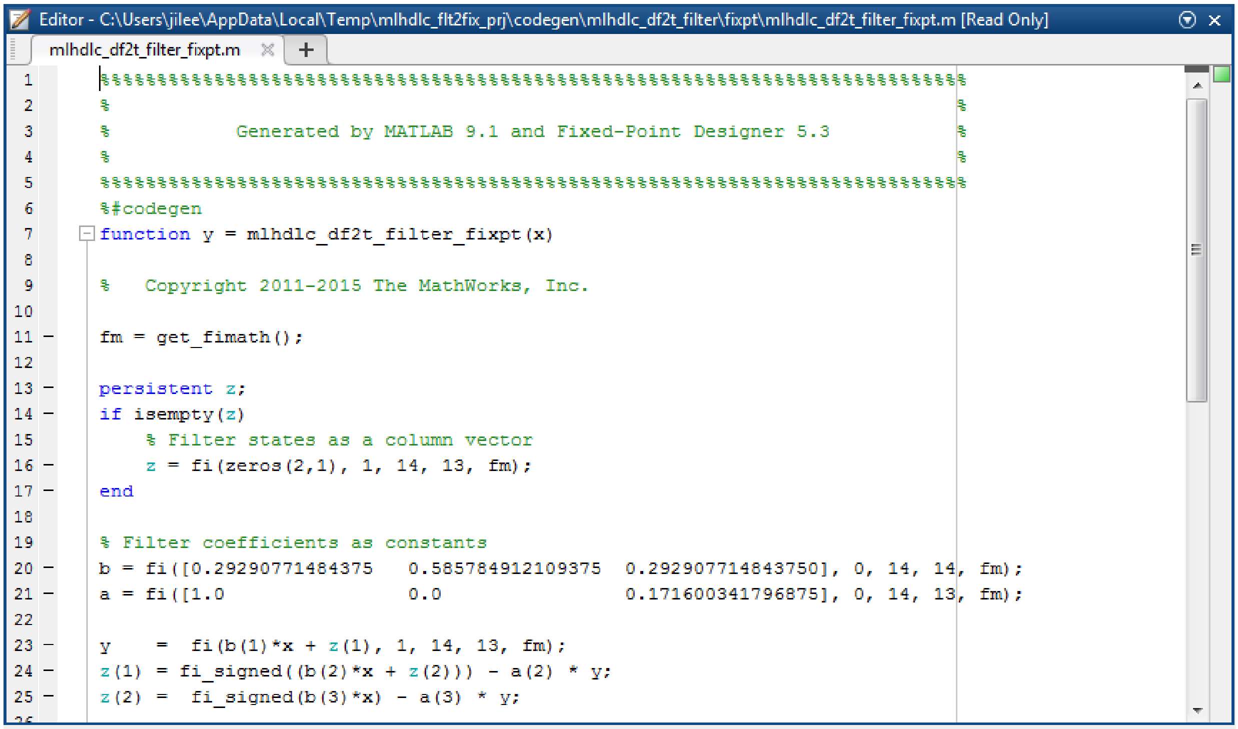
Function: mlhdlc_df2t_filter

Default word length: 14

```
%#codegen
function y = mlhdlc_df2t_filter(x)

%   Copyright 2011-2015 The MathWorks, Inc.

persistent z;
if isempty(z)
    % Filter states as a column vector
    z = zeros(2,1);
end

% Filter coefficients as constants
b = [0.29290771484375   0.585784912109375  0.292907714843750];
a = [1.0                0.0                0.171600341796875];

y    = b(1)*x + z(1);
```

```
### Analyzing the test bench(es) 'mlhdlc_df2t_filter_tb'
### Begin Floating Point Simulation (Instrumented)
### Floating Point Simulation Completed in 1.0194 sec(s)
### Elapsed Time:             1.5504 sec(s)


 Type Validation Output    (11/1/16 5:34 PM)

### Generating Type Proposal Report for 'mlhdlc_df2t_filter' mlhdlc_df2t_filter_report.htm
### Generating Fixed Point MATLAB Code mlhdlc_df2t_filter_fixpt using Proposed Types
### Generating Fixed Point MATLAB Design Wrapper mlhdlc_df2t_filter_wrapper_fixpt
### Generating Mex file for ' mlhdlc_df2t_filter_wrapper_fixpt '
Code generation successful: View report
```

The generated code and other conversion artifacts are available via hyperlinks in the output window. The fixed-point types are explicitly shown in the generated MATLAB code.



Editor - C:\Users\jilee\AppData\Local\Temp\mlhdlc_flt2fix_prj\codegen\mlhdlc_df2t_filter\fixpt\mlhdlc_df2t_filter_fixpt.m [Read Only]

mlhdlc_df2t_filter_fixpt.m

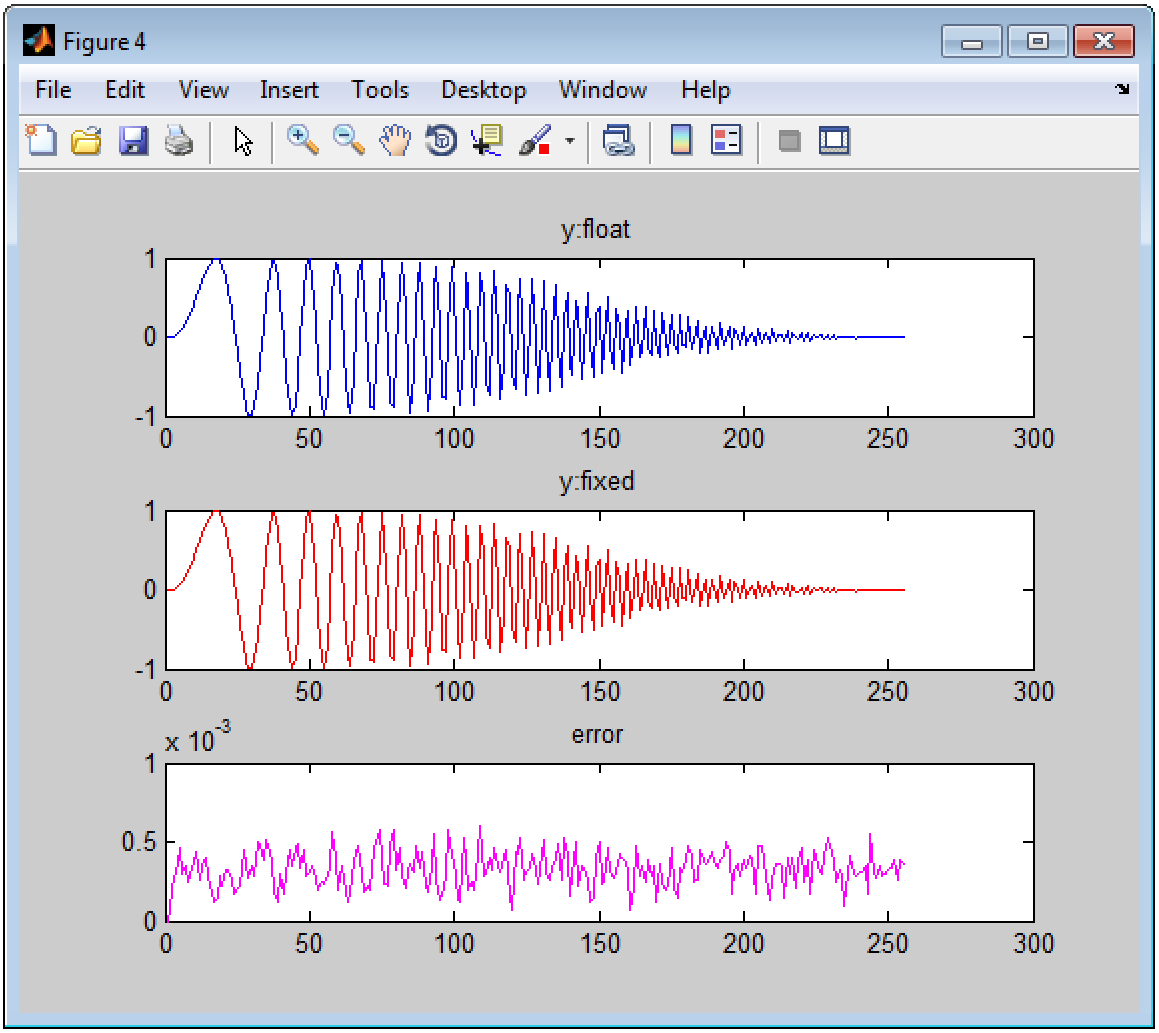
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                           %
%           Generated by MATLAB 9.1 and Fixed-Point Designer 5.3            %
%                                                                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%#codegen
function y = mlhdlc_df2t_filter_fixpt(x)

%   Copyright 2011-2015 The MathWorks, Inc.

fm = get_fimath();

persistent z;
if isempty(z)
    % Filter states as a column vector
    z = fi(zeros(2,1), 1, 14, 13, fm);
end

% Filter coefficients as constants
b = fi([0.29290771484375   0.585784912109375   0.292907714843750], 0, 14, 14, fm);
a = fi([1.0                0.0                 0.171600341796875], 0, 14, 13, fm);

y    =  fi(b(1)*x + z(1), 1, 14, 13, fm);
z(1) = fi_signed((b(2)*x + z(2))) - a(2) * y;
z(2) =  fi_signed(b(3)*x) - a(3) * y;
```

Step 5: Test Numerics

1 Click on the 'Test Numerics' button.

In this step, the generated fixed-point code is executed using MATLAB Coder.

If you enable the 'Log all inputs and outputs for comparison plots' option on the 'Test Numerics' pane, an additional plot is generated for each scalar output that shows the floating point and fixed point results, as well as the difference between the two. For non-scalar outputs, only the error information is shown.



Step 6: Iterate on the Results

If the numerical results do not meet your desired accuracy after fixed-point simulation, you can return to the 'Propose Fixed-Point Types' step in the Workflow Advisor. Adjust the word length settings or individually modify types as desired, and repeat the rest of the steps in the workflow until you achieve your desired results.

You can refer to the Fixed-Point Type Conversion and Refinement example for more details on how to iterate and refine the numerics of the algorithm in the generated fixed-point code.

Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_flt2fix_prj'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

Fixed-Point Type Conversion and Refinement

This example shows how to achieve your desired numerical accuracy when converting fixed-point MATLAB® code to floating-point code using the HDL Workflow Advisor.

Introduction

The floating-point to fixed-point conversion workflow in HDL Coder™ includes the following steps:

1. Verify the floating-point design is compatible for code generation.
2. Compute fixed-point types based on the simulation of the testbench.
3. Generate readable and traceable fixed-point MATLAB® code.
4. Verify the generated fixed-point design.

This tutorial uses Kalman filter suitable for HDL code generation to illustrate some key aspects of fixed-point conversion workflow, specifically steps 2 and 3 in the above list.

MATLAB Design

The MATLAB code used in this example implements a simple Kalman filter. This example also contains a MATLAB testbench that exercises the filter.

Kalman filter implementation suitable for HDL code generation

```
design_name = 'mlhdlc_kalman_hdl';
testbench_name = 'mlhdlc_kalman_hdl_tb';
%
% MATLAB Design: <matlab:edit('mlhdlc_kalman_hdl') mlhdlc_kalman_hdl>
% MATLAB testbench: <matlab:edit('mlhdlc_kalman_hdl_tb') mlhdlc_kalman_hdl_tb>
%
```

Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_flt2fix'];

% create a temporary folder and copy the MATLAB files
```

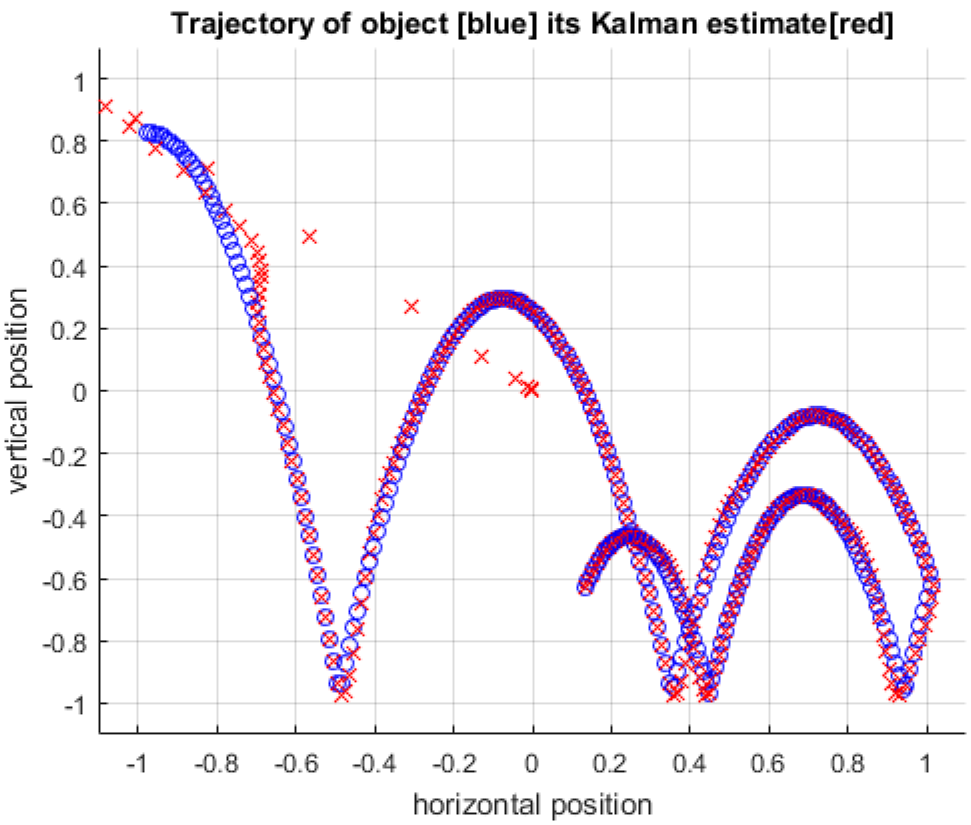
```
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name,'.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name,'.m*']), mlhdlc_temp_dir);
```

Simulate the Design

Simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_kalman_hdl_tb
```

```
Running --------> mlhdlc_kalman_hdl_tb

Current plot held
Current plot released
```



Create a New HDL Coder Project

To create a new project, enter the following command:

```
coder -hdlcoder -new flt2fix_project
```

Next, add the file 'mlhdlc_kalman_hdl.m' to the project as the MATLAB Function and 'mlhdlc_kalman_hdl_tb.m' as the MATLAB Test Bench.

You can refer to Getting Started with MATLAB to HDL Workflow tutorial for a more complete tutorial on creating and populating HDL Coder projects.

Fixed-Point Code Generation Workflow

Perform the following tasks before moving on to the fixed-point type proposal step:

1. Click the 'Workflow Advisor' button to launch the HDL Workflow Advisor.
2. Choose 'Convert to fixed-point at build time' for the 'Fixed-point conversion' option.
3. Click 'Run' button to define input types for the design from the testbench.
4. Select the 'Fixed-Point Conversion' workflow step.
5. Click 'Analyze' to execute the instrumented floating-point simulation.

Refer to Floating-Point to Fixed-Point Conversion for a more complete tutorial on these steps.

Determine the Initial Fixed Point Types

After instrumented floating-point simulation completes, you will see 'Fixed-Point Types are proposed' based on the simulation results.

At this stage of the conversion proposes fixed-point types for each variable in the design based on the recorded min/max values of the floating point variables and user input.

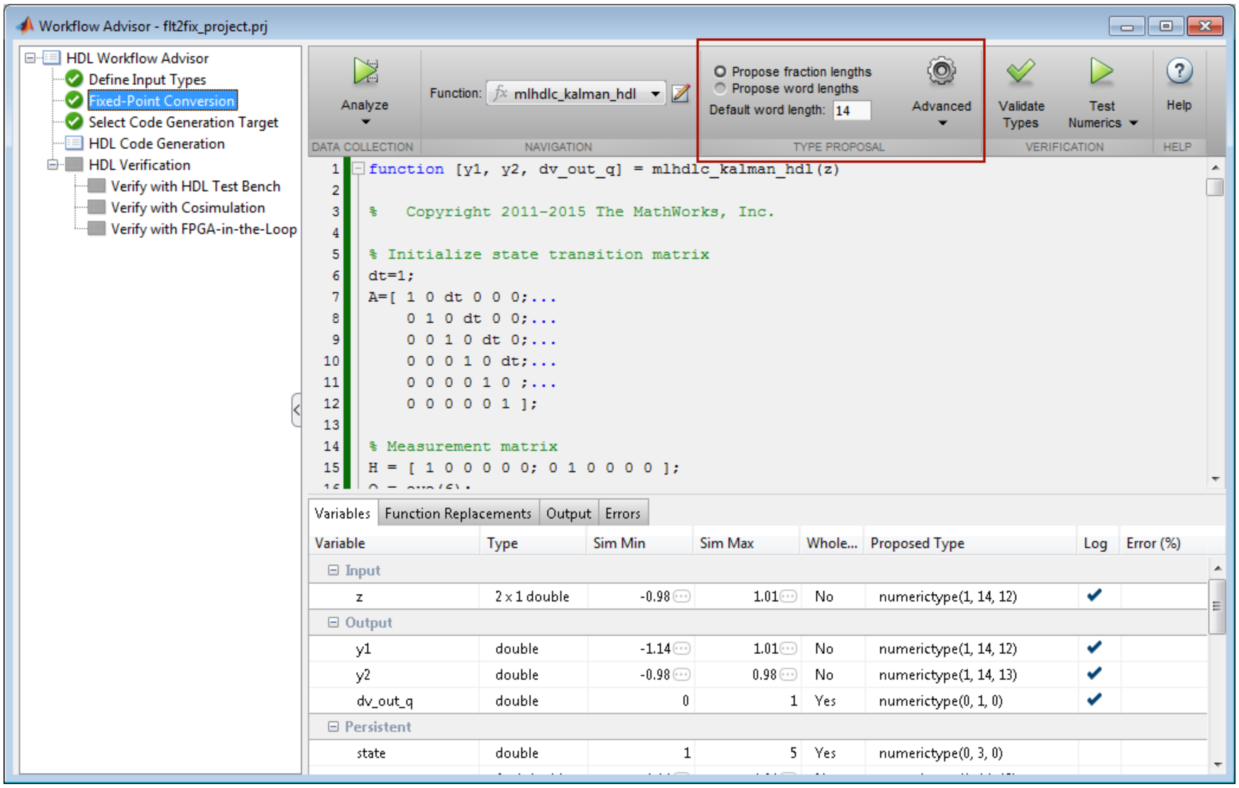
At this point, for all variables, you can (re)compute and propose:

- Fraction lengths for a given fixed word length setting, or
- Word lengths for a given fixed fraction length setting.

Choose the Word Length Setting

When you are starting with a floating-point design and going through the floating-point to fixed-point conversion for the first time, it is a good practice to start by specifying a 'Default Word Length' setting based on the largest dynamic range of all the variables in the design.

In this example, we start with a default word length of 22 and run the 'Propose Fixed-Point Types' step.



Function: mlhdlc_kalman_hdl

● Propose fraction lengths
○ Propose word lengths
Default word length: 14

```
function [y1, y2, dv_out_q] = mlhdlc_kalman_hdl(z)

%   Copyright 2011-2015 The MathWorks, Inc.

% Initialize state transition matrix
dt=1;
A=[ 1 0 dt 0 0 0;...
    0 1 0 dt 0 0;...
    0 0 1 0 dt 0;...
    0 0 0 1 0 dt;...
    0 0 0 0 1 0 ;...
    0 0 0 0 0 1 ];

% Measurement matrix
H = [ 1 0 0 0 0 0; 0 1 0 0 0 0 ];
```

Variable	Type	Sim Min	Sim Max	Whole...	Proposed Type	Log	Error (%)
⊟ Input							
z	2 x 1 double	-0.98	1.01	No	numerictype(1, 14, 12)	✓	
⊟ Output							
y1	double	-1.14	1.01	No	numerictype(1, 14, 12)	✓	
y2	double	-0.98	0.98	No	numerictype(1, 14, 13)	✓	
dv_out_q	double	0	1	Yes	numerictype(0, 1, 0)	✓	
⊟ Persistent							
state	double	1	5	Yes	numerictype(0, 3, 0)		

Explore the Proposed Fixed-Point Type Table

The type table contains the following information for each variable, organized by function, existing in the floating-point MATLAB design:

- Sim Min: The minimum value assigned to the variable during simulation.
- Sim Max: The maximum value assigned to the variable during simulation.
- Whole Number: Whether all values assigned during simulation are integer.

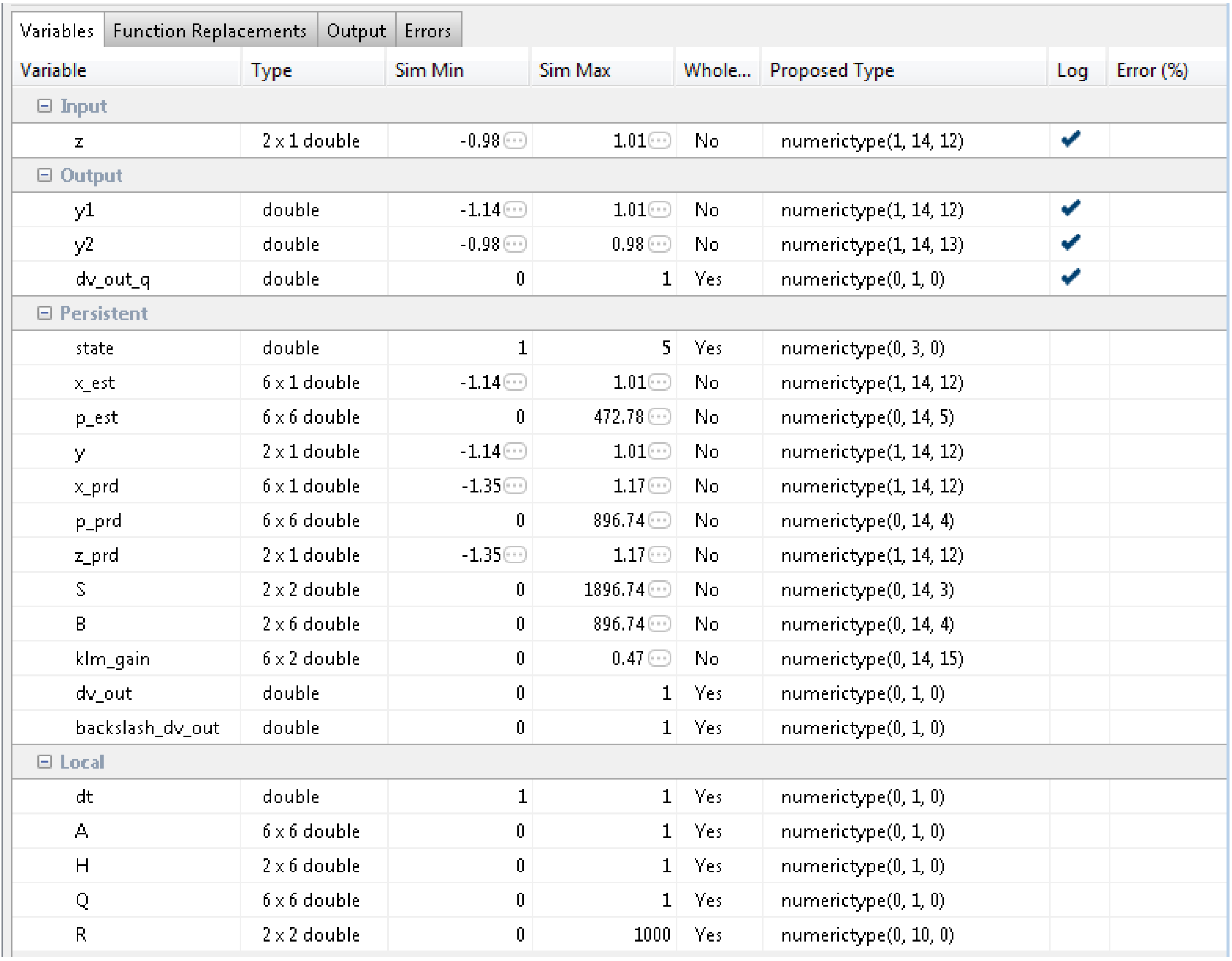
The type proposal step uses the above information and combines it with the user-specified word length settings to propose a fixed-point type for each variable.

You can also use 'Compute Derived Range Analysis' to compute derived ranges and that is covered in detail in this tutorial Computing Derived Ranges in fixed-point conversion

Interpret the Proposed Numeric Types for Variables

Based on the simulation range (min & max) values and the default word length setting, a numeric type is proposed for each variable.

The following table shows numeric type proposals for a 'Default word length' of 22 bits.



Variables | Function Replacements | Output | Errors

Variable	Type	Sim Min	Sim Max	Whole...	Proposed Type	Log	Error (%)
⊟ Input							
z	2 x 1 double	-0.98	1.01	No	numerictype(1, 14, 12)	✓	
⊟ Output							
y1	double	-1.14	1.01	No	numerictype(1, 14, 12)	✓	
y2	double	-0.98	0.98	No	numerictype(1, 14, 13)	✓	
dv_out_q	double	0	1	Yes	numerictype(0, 1, 0)	✓	
⊟ Persistent							
state	double	1	5	Yes	numerictype(0, 3, 0)		
x_est	6 x 1 double	-1.14	1.01	No	numerictype(1, 14, 12)		
p_est	6 x 6 double	0	472.78	No	numerictype(0, 14, 5)		
y	2 x 1 double	-1.14	1.01	No	numerictype(1, 14, 12)		
x_prd	6 x 1 double	-1.35	1.17	No	numerictype(1, 14, 12)		
p_prd	6 x 6 double	0	896.74	No	numerictype(0, 14, 4)		
z_prd	2 x 1 double	-1.35	1.17	No	numerictype(1, 14, 12)		
S	2 x 2 double	0	1896.74	No	numerictype(0, 14, 3)		
B	2 x 6 double	0	896.74	No	numerictype(0, 14, 4)		
klm_gain	6 x 2 double	0	0.47	No	numerictype(0, 14, 15)		
dv_out	double	0	1	Yes	numerictype(0, 1, 0)		
backslash_dv_out	double	0	1	Yes	numerictype(0, 1, 0)		
⊟ Local							
dt	double	1	1	Yes	numerictype(0, 1, 0)		
A	6 x 6 double	0	1	Yes	numerictype(0, 1, 0)		
H	2 x 6 double	0	1	Yes	numerictype(0, 1, 0)		
Q	6 x 6 double	0	1	Yes	numerictype(0, 1, 0)		
R	2 x 2 double	0	1000	Yes	numerictype(0, 10, 0)		

Examine the types proposed in the above table for variables instrumented in the top-level design.

Floating-Point Range for variable 'B':

- Simulation Info: SimMin: 0, SimMax: 896.74.., Whole Number: No
- Type Proposed: numerictype(0,22,12) (Signedness: Unsigned, WordLength: 22, FractionLength: 12)

The floating-point range:

- Has the same number of bits as the 'Default word length'.
- Uses the minimum number of bits to completely represent the range.
- Uses the rest of the bits to represent the precision.

Integer Range for variable 'A':

- Simulation Info: SimMin: 0, SimMax: 1, Whole Number: Yes
- Type Proposed: numerictype(0,1,0) (Signedness: Unsigned, WordLength: 1, FractionLength: 0)

The integer range:

- Has the minimum number of bits to represent the whole integer range.
- Has no fractional bits.

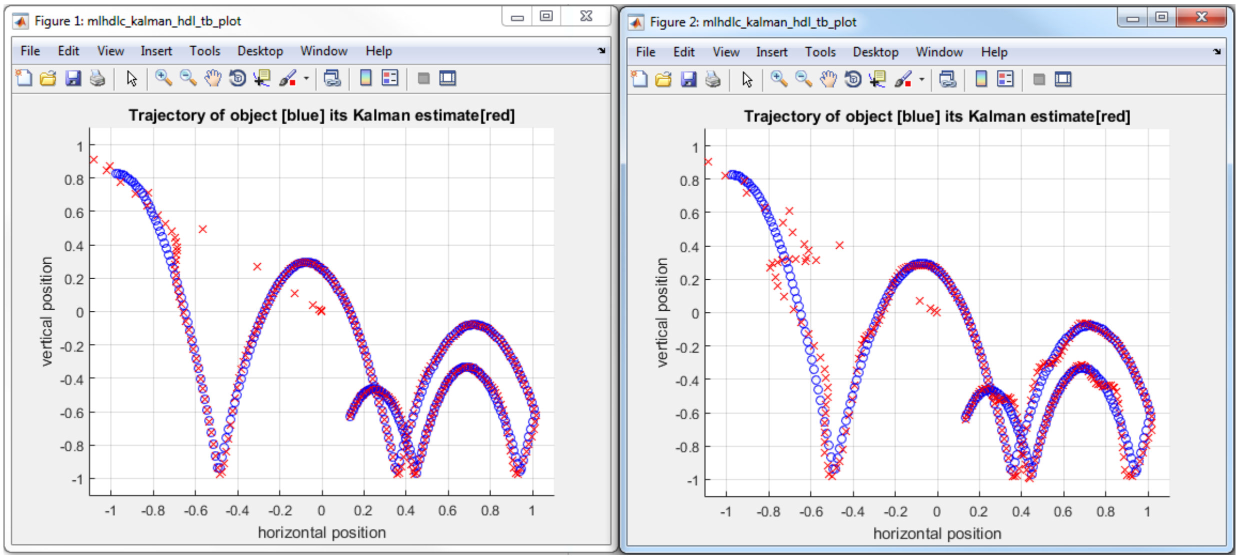
All the information in the table is editable, persists across iterations, and is saved with your code generation project.

Generate Fixed-Point Code and Verify the Generated Code

Based on the numeric types proposed for a default word length of 22, continue with fixed-point code generation and verification steps and observe the plots.

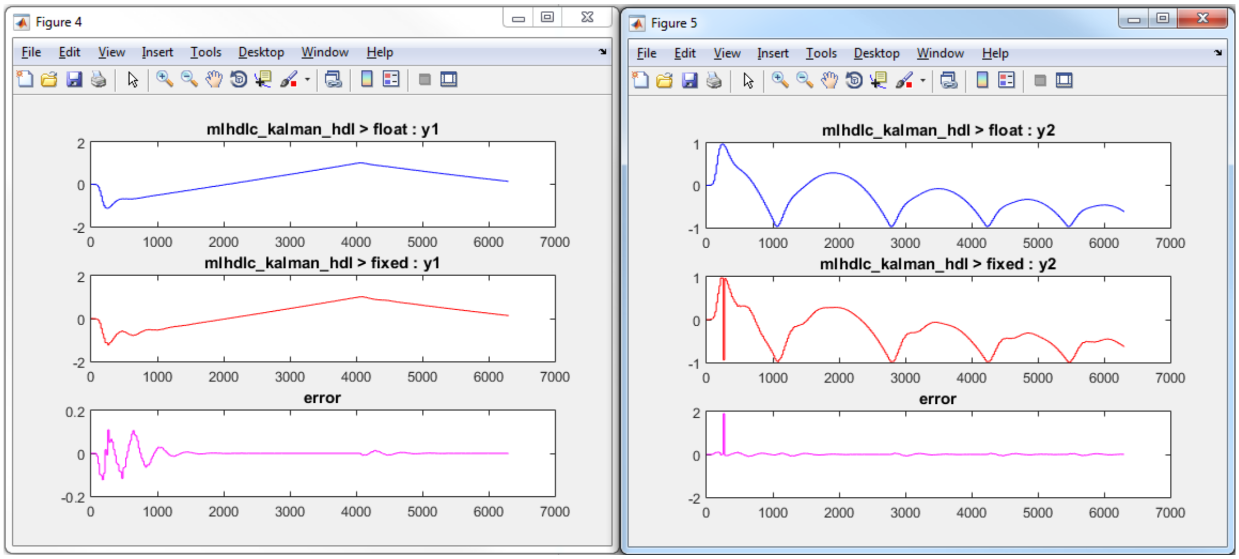
1. Click on 'Validate Types' to apply computed fixed-point types.
2. Next choose the option 'Log inputs and outputs for comparison plots' and then click on the 'Test Numerics' to rerun the testbench on the fixed-point code.

The plot on the left is generated from testbench during the simulation of floating-point code, the one on the right is generated from the simulation of the generated fixed-point code. Notice, the plots do not match.



Having chosen comparison plots option you will see additional plots that compare the floating and fixed point simulation results for each output variable.

Examine the error graph for each output variable. It is very high for this particular design.



Iterate on the Results

One way to reduce the error is to increase 'Default word length' and repeat the fixed-point conversion.

In this example design, when a word length of 22 bits is chosen there is a lot of truncation error when representing the precision. More bits are required to the right of the binary point to reduce the truncation errors.

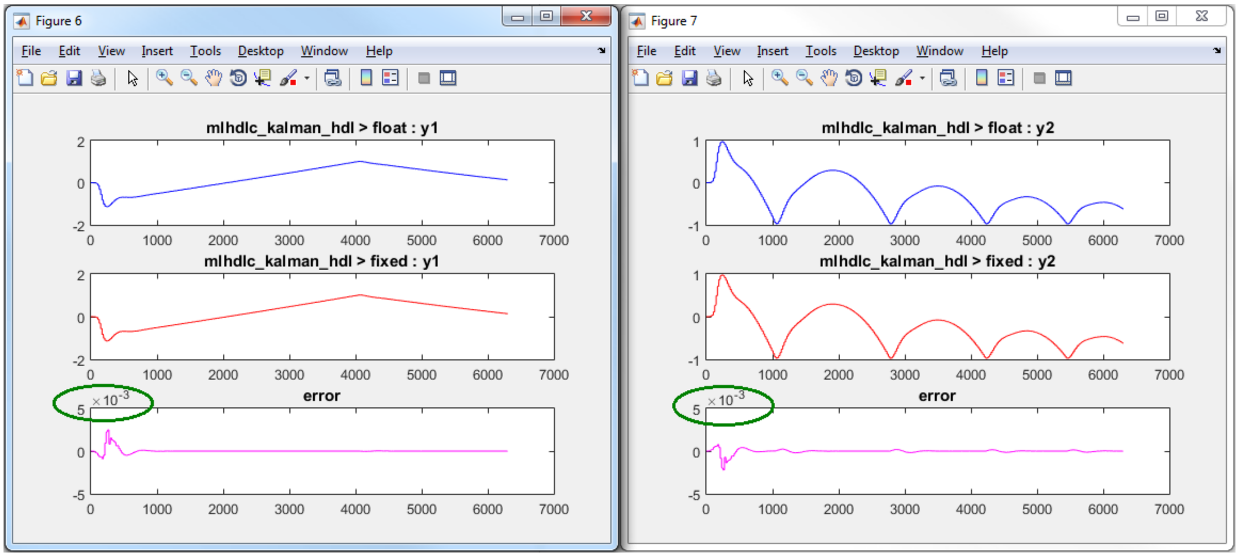
Let us now increase the default word length to 28 bits and repeat the type proposal and validation steps.

1 Select a 'Default word length' of 28.

Changing default word length automatically triggers the type proposal step and new fixed-point types are proposed based on the new word length setting. Also notice that type validation needs to be rerun and numerics need to be verified again.

1 Click on 'Validate Types'.
2 Click on 'Test Numerics' to rerun the testbench on the fixed-point code.

Once these steps are complete, re-examine the comparison plots and notice that the error is now roughly three orders of magnitude smaller.



Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matla
mlhdlc_temp_dir = [tempdir 'mlhdlc_flt2fix'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

Working with Generated Fixed-Point Files

This example shows how to work with the files generated during floating-point to fixed-point conversion.

Introduction

This tutorial uses a simple filter implemented in floating-point and an associated testbench to illustrate the file structure of the generated fixed-point code.

```
design_name = 'mlhdlc_filter';
testbench_name = 'mlhdlc_filter_tb';
```

MATLAB® Code

1 MATLAB Design: mlhdlc_filter
2 MATLAB testbench: mlhdlc_filter_tb

Create a New Folder and Copy Relevant Files

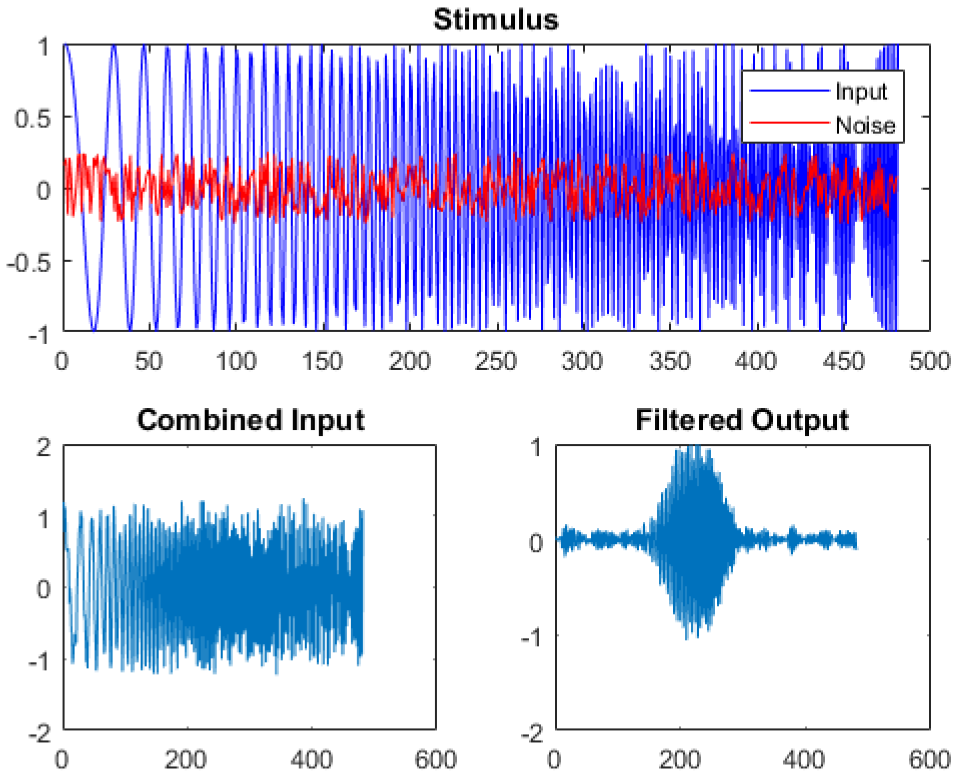
Executing the following lines of code copies the necessary example files into a temporary folder.

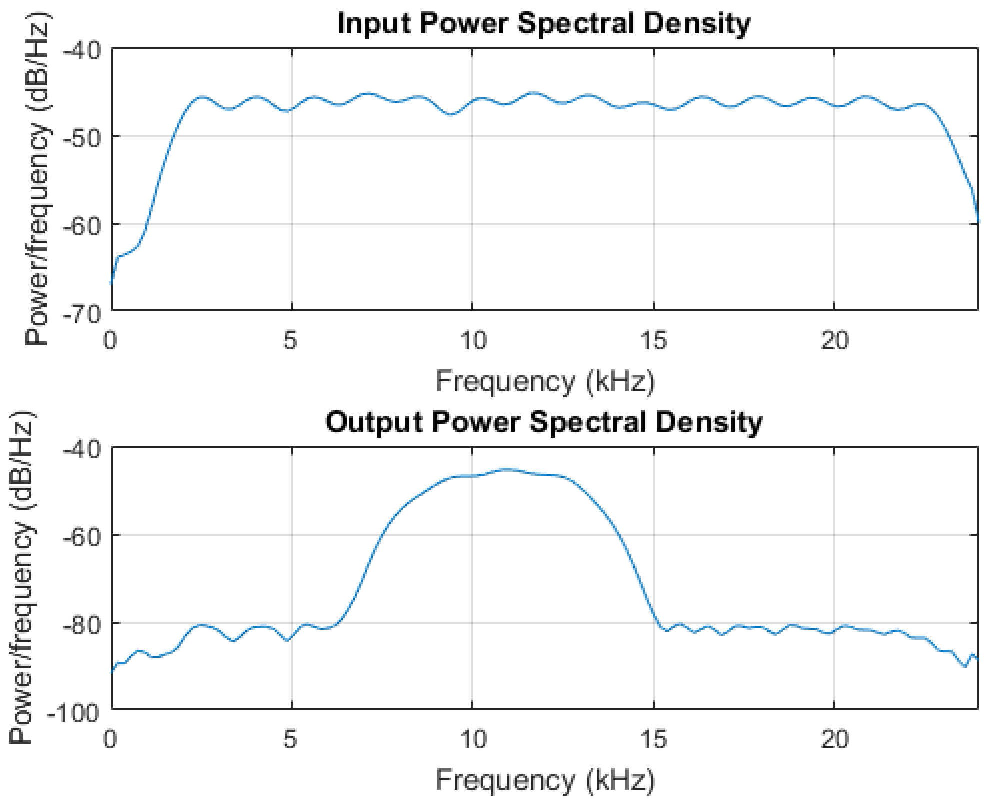
```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_flt2fix'];

% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name,'.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name,'.m*']), mlhdlc_temp_dir);
```

Simulate the Design

Simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_filter_tb
```


Stimulus
Input
Noise
Combined Input
Filtered Output



Create a New HDL Coder™ Project

To create a new project, enter the following command:

```
coder -hdlcoder -new flt2fix_project
```

Next, add the file 'mlhdlc_filter' to the project as the MATLAB Function and 'mlhdlc_filter_tb' as the MATLAB Test Bench.

You can refer to the Getting Started with MATLAB to HDL Workflow tutorial for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Fixed-Point Code Generation Workflow

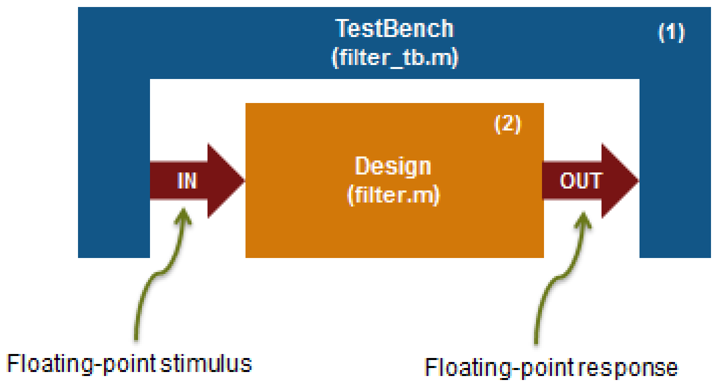
Perform the following tasks in preparation for the fixed-point code generation step:

1 Click the Advisor button to launch the Workflow Advisor.
2 Choose 'Yes' for the option 'Design needs conversion to fixed-point'.
3 Right-click the 'Propose Fixed-Point Types' step.
4 Choose 'Run to Selected Task' to execute the instrumented floating-point simulation.

Refer to the Floating-Point to Fixed-Point Conversion tutorial for a more complete description of these steps.

Floating-Point Design Structure

The original floating-point design and testbench have the following relationship.



For floating-point to fixed-point conversion, the following requirements apply to the original design and the testbench:

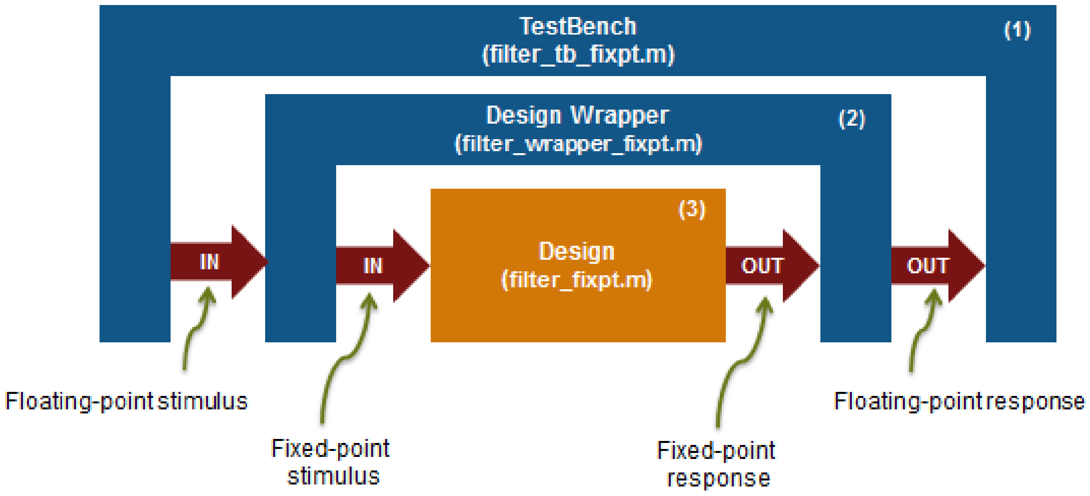
- The testbench 'mlhdlc_filter_tb.m' (1) must be a script or a function with no inputs.
- The design 'mlhdlc_filter.m' (2) must be a function.
- There must be at least one call to the design from the testbench. All call sites contribute when determining the proposed fixed-point types.

- Both the design and testbench can call other sub-functions within the file or other functions on the MATLAB path. Functions that exist within matlab/toolbox are not converted to fixed-point.

In the current example, the MATLAB testbench 'mlhdlc_filter_tb' has a single call to the design function 'mlhdlc_filter'. The testbench calls the design with floating-point inputs and accumulates the floating-point results for plotting.

Validate Types

During the type validation step, fixed-point code is generated for this design and complied to verify that there are no errors when applying the types. The output files will have the following structure.



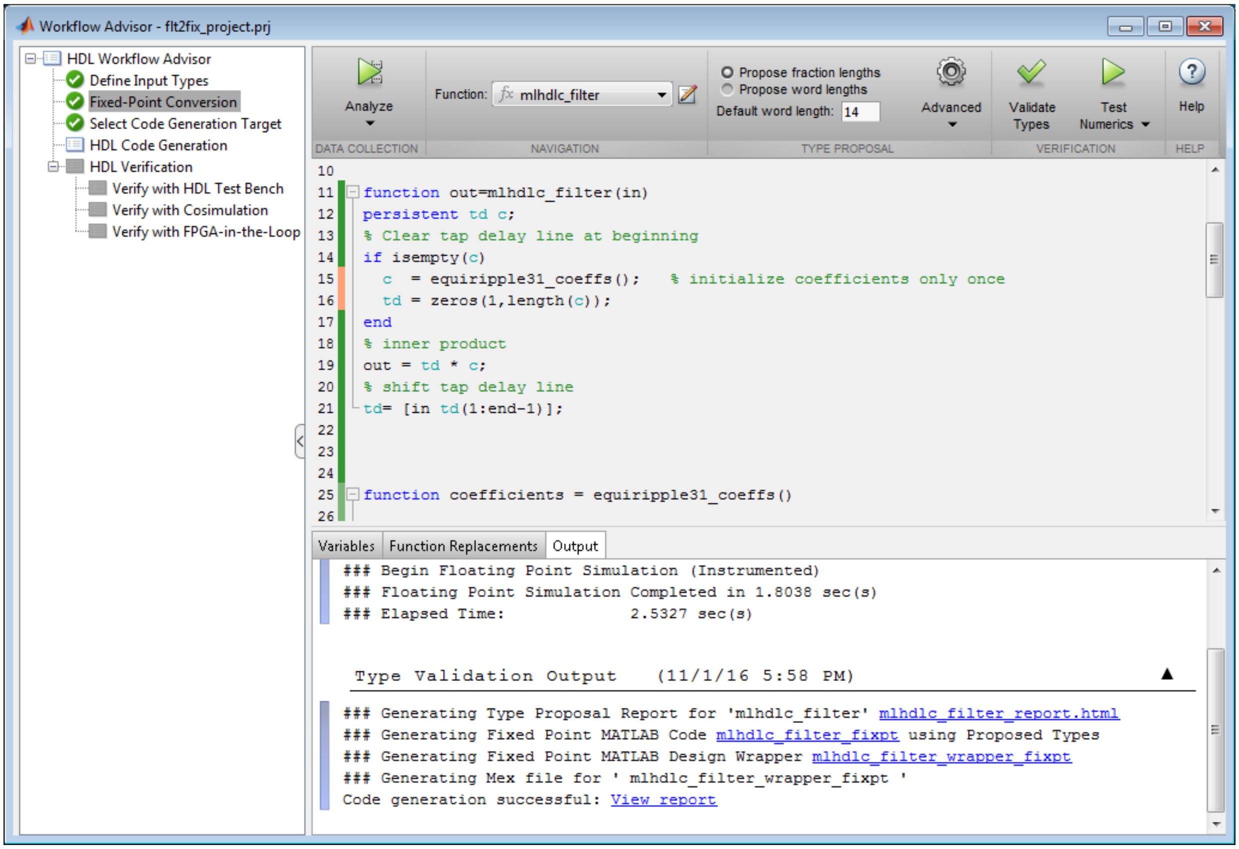
The following steps are performed during fixed-point type validation process:

1. The design file 'mlhdlc_filter.m' is converted to fixed-point to generates fixed-point MATLAB code, 'mlhdlc_filter_FixPt.m' (3).
2. All user-written functions called in the floating-point design are converted to fixed-point and included in the generated design file.

3 A new design wrapper file is created, called 'mlhdlc_filter_wrapper_FixPt.m' (2). This file converts the floating-point data values supplied by the testbench to the fixed-point types determined for the design inputs during the conversion step. These fixed-point values are fed into the converted fixed-point design, 'mlhdlc_filter_FixPt.m'.

4 'mlhdlc_filter_FixPt.m' will be used for HDL code generation.

5 All the generated fixed-point files are stored in the output directory 'codegen/filter/fixpt'.



```
function out=mlhdlc_filter(in)
persistent td c;
% Clear tap delay line at beginning
if isempty(c)
  c  = equiripple31_coeffs();   % initialize coefficients only once
  td = zeros(1,length(c));
end
% inner product
out = td * c;
% shift tap delay line
td= [in td(1:end-1)];



function coefficients = equiripple31_coeffs()
```

```
### Begin Floating Point Simulation (Instrumented)
### Floating Point Simulation Completed in 1.8038 sec(s)
### Elapsed Time:             2.5327 sec(s)


 Type Validation Output    (11/1/16 5:58 PM)

### Generating Type Proposal Report for 'mlhdlc_filter' mlhdlc_filter_report.html
### Generating Fixed Point MATLAB Code mlhdlc_filter_fixpt using Proposed Types
### Generating Fixed Point MATLAB Design Wrapper mlhdlc_filter_wrapper_fixpt
### Generating Mex file for ' mlhdlc_filter_wrapper_fixpt '
Code generation successful: View report
```

Click the links to the generated code in the Workflow Advisor log window to examine the generated fixed-point design, wrapper, and test bench.

Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_flt2fix'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

Specify Type Proposal Options

Basic Type Proposal Settings	Values	Description
Fixed-point type proposal mode	Propose fraction lengths for specified word length	Use the specified word length for data type proposals and propose the minimum fraction lengths to avoid overflows.
	Propose word lengths for specified fraction length (default)	Use the specified fraction length for data type proposals and propose the minimum word lengths to avoid overflows.
Default word length	14 (default)	Default word length to use when **Fixed-point type proposal mode** is set to `Propose fraction lengths for specified word lengths`
Default fraction length	4 (default)	Default fraction length to use when **Fixed-point type proposal mode** is set to `Propose word lengths for specified fraction lengths`

Advanced Type Proposal Settings	Values	Description
When proposing types **Note:** Manually-entered static ranges always take precedence over simulation ranges.	ignore simulation ranges	Propose data types based on derived ranges.
	ignore derived ranges	Propose data types based on simulation ranges.
	use all collected data (default)	Propose data types based on both simulation and derived ranges.
Propose target container types	Yes	Propose data type with the smallest word length that can represent the range and is suitable for C code

Advanced Type Proposal Settings	Values	Description
		generation (8,16,32, 64 ...). For example, for a variable with range `[0..7]`, propose a word length of 8 rather than 3.
	No (default)	Propose data types with the minimum word length needed to represent the value.
Optimize whole numbers	No	Do not use integer scaling for variables that were whole numbers during simulation.
	Yes (default)	Use integer scaling for variables that were whole numbers during simulation.
Signedness	Automatic (default)	Proposes signed and unsigned data types depending on the range information for each variable.
	Signed	Propose signed data types.
	Unsigned	Propose unsigned data types.
Safety margin for sim min/max (%)	0 (default)	Specify safety factor for simulation minimum and maximum values. The simulation minimum and maximum values are adjusted by the percentage designated by this parameter, allowing you to specify a range different from that obtained from the simulation run. For example, a value of `55` specifies that you want a range at least 55 percent larger. A value of `-15` specifies that a range up to 15 percent smaller is acceptable.
Search paths	`''` (default)	Add paths to the list of paths to search for MATLAB files. Separate list items with a semicolon.

fimath Settings	Values	Description
Rounding method	Ceiling	Specify the `fimath` properties for the generated fixed-point data types. The default fixed-point math properties use the `Floor` rounding and `Wrap` overflow. These settings generate the most efficient code but might cause problems with overflow. After code generation, if required, modify these settings to optimize the generated code, or example, avoid overflow or eliminate bias, and then rerun the verification. For more information on `fimath` properties, see “fimath Object Properties” (Fixed-Point Designer).
	Convergent	
	Floor (default)	
	Nearest	
	Round	
	Zero	
Overflow action	Saturate	
	Wrap (default)	
Product mode	FullPrecision (default)	
	KeepLSB	
	KeepMSB	
	SpecifyPrecision	
Sum mode	FullPrecision (default)	
	KeepLSB	
	KeepMSB	
	SpecifyPrecision	

Generated File Settings	Value	Description
Generated fixed-point file name suffix	_fixpt (default)	Specify the suffix to add to the generated fixed-point file names.

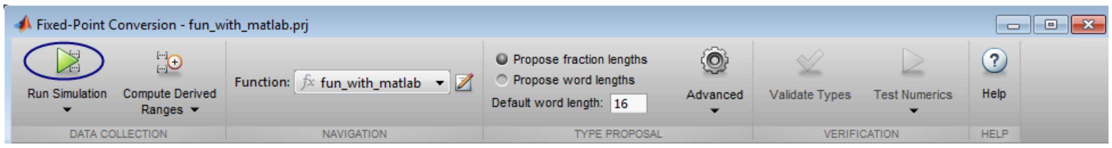
Plotting and Reporting Settings	Values	Description
Custom plot function	`''` (default)	Specify the name of a custom plot function to use for comparison plots.
Plot with Simulation Data Inspector	No (default)	

Plotting and Reporting Settings	Values	Description
	Yes	Specify whether to use the Simulation Data Inspector for comparison plots.
Highlight potential data type issues	No (default)	Specify whether to highlight potential data types in the generated html report. If this option is turned on, the report highlights single-precision, double-precision, and expensive fixed-point operation usage in your MATLAB code.
	Yes	

Log Data for Histogram

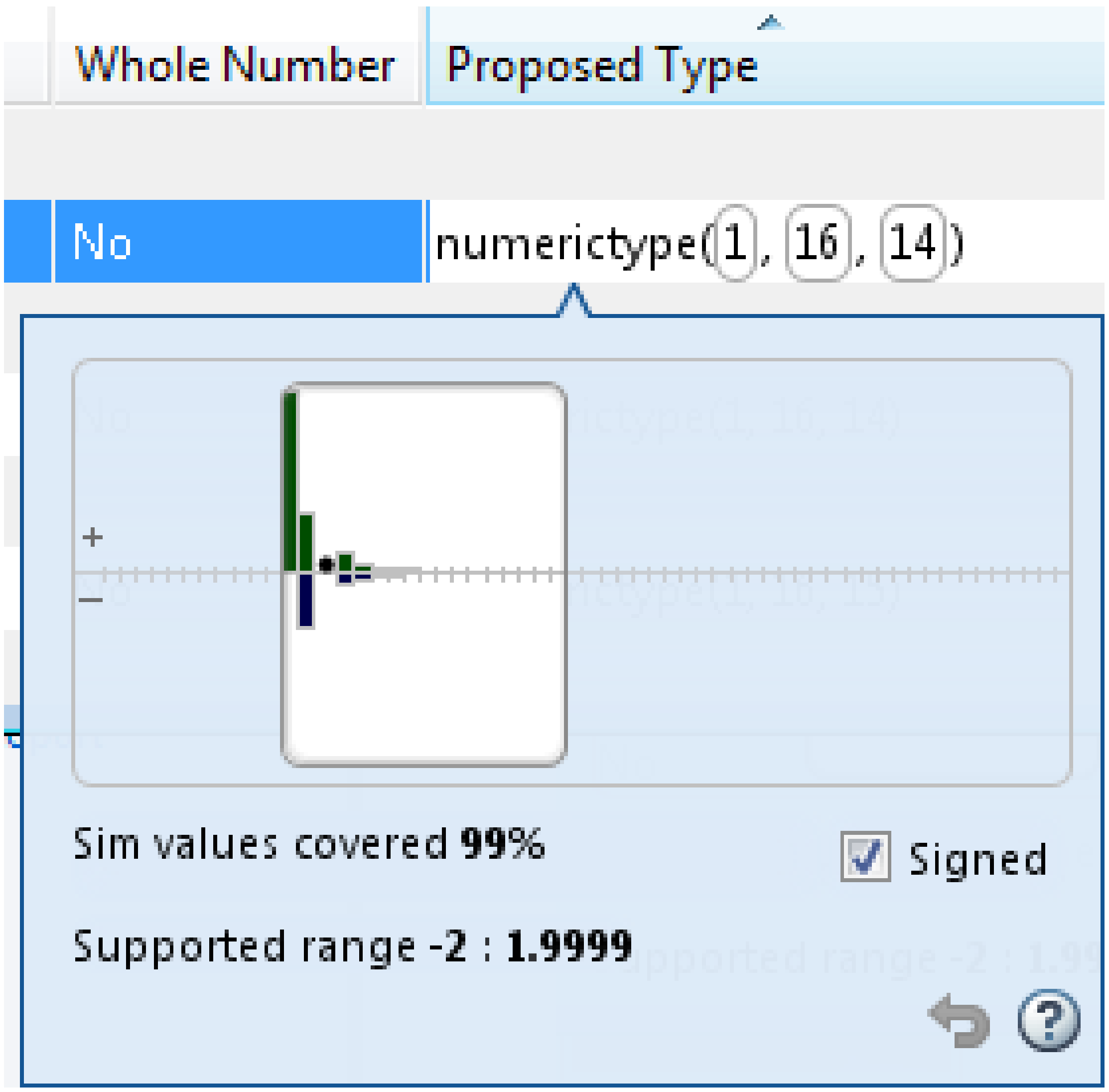
To log data for histograms:

1 In the Fixed-Point Conversion window, click **Run Simulation** and select `Log data for histogram`, and then click the Run Simulation button.



The simulation runs and the simulation minimum and maximum ranges are displayed on the **Variables** tab. Using the simulation range data, the software proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column.

2 To view a histogram for a variable, click the variable's **Proposed Type** field.



Whole Number	Proposed Type
No	numerictype(1, 16, 14)

Sim values covered **99%**

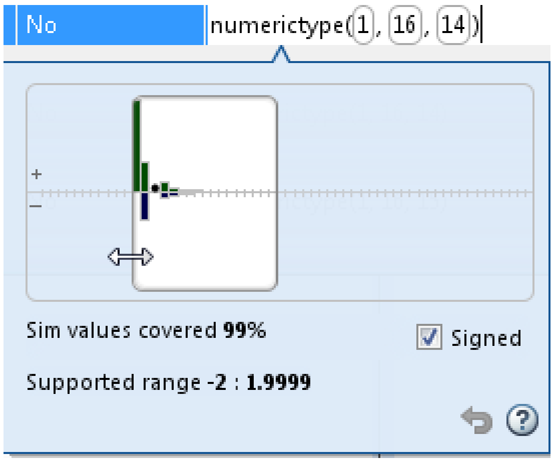
Signed ✓

Supported range -2 : **1.9999**

3 You can view the effect of changing the proposed data types by:

- Selecting and dragging the white bounding box in the histogram window. This action does not change the word length of the proposed data type, but modifies the position of the binary point within the word so that the fraction length of the proposed data type changes.

- Selecting and dragging the left edge of the bounding box to increase or decrease the word length. This action does not change the fraction length or the position of the binary point.



numerictype(1, 16, 14)

Sim values covered **99%**

☑ Signed

Supported range **-2 : 1.9999**

- Selecting and dragging the right edge to increase or decrease the fraction length of the proposed data type. This action does not change the position of the binary point. The word length changes to accommodate the fraction length.
- Selecting or clearing **Signed**. Clear **Signed** to ignore negative values.

Before committing changes, you can revert to the types proposed by the automatic conversion by clicking ⮌.

View and Modify Variable Information

View Variable Information

On the **Convert to Fixed Point** page of the app, you can view information about the variables in the MATLAB functions. To view information about the variables that you select in the **Source Code** pane, use the **Variables** tab or place your cursor over a variable in the code window. For more information, see “Viewing Variables” on page 4-52.

You can view the variable information:

- **Variable**

  Variable name. Variables are classified and sorted as inputs, outputs, persistent, or local variables.
- **Type**

  The original size, type, and complexity of each variable.
- **Sim Min**

  The minimum value assigned to the variable during simulation.
- **Sim Max**

  The maximum value assigned to the variable during simulation.

To search for a variable in the MATLAB code window and on the **Variables** tab, use `Ctrl+F`. The app highlights occurrences of the variable in the code.

Modify Variable Information

If you modify variable information, the app highlights the modified values using bold text. You can modify the following fields:

- **Static Min**

  You can enter a value for **Static Min** into the field or promote **Sim Min** information. See “Promote Sim Min and Sim Max Values” on page 4-42.

Editing this field does not trigger static range analysis, but the app uses the edited values in subsequent analyses.

- **Static Max**

  You can enter a value for **Static Max** into the field or promote **Sim Max** information. See “Promote Sim Min and Sim Max Values” on page 4-42.

  Editing this field does not trigger static range analysis, but the app uses the edited values in subsequent analyses.

- **Whole Number**

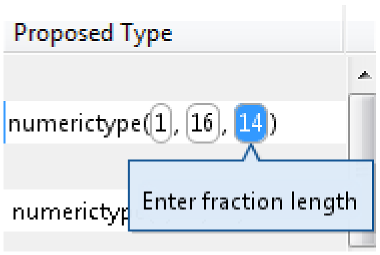
  The app uses simulation data to determine whether the values assigned to a variable during simulation were always integers. You can manually override this field.

  Editing this field does not trigger static range analysis, but the app uses the edited value in subsequent analyses.

- **Proposed Type**

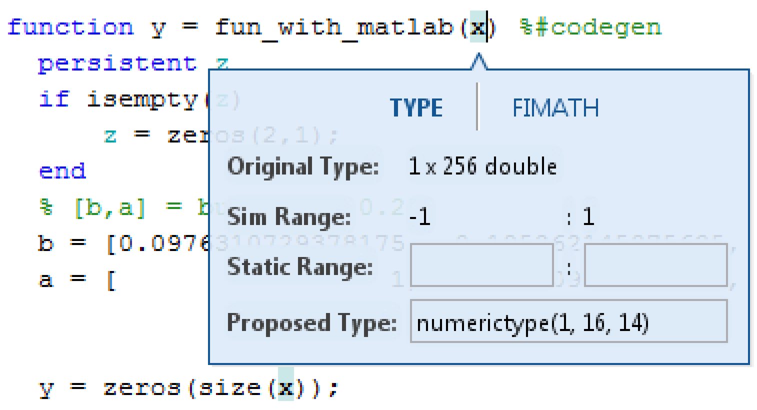
  You can modify the signedness, word length, and fraction length settings individually:

  - On the **Variables** tab, modify the value in the **ProposedType** field.



```
numerictype(1, 16, 14)
numerictyp
```

  - In the code window, select a variable, and then modify the **ProposedType** field.



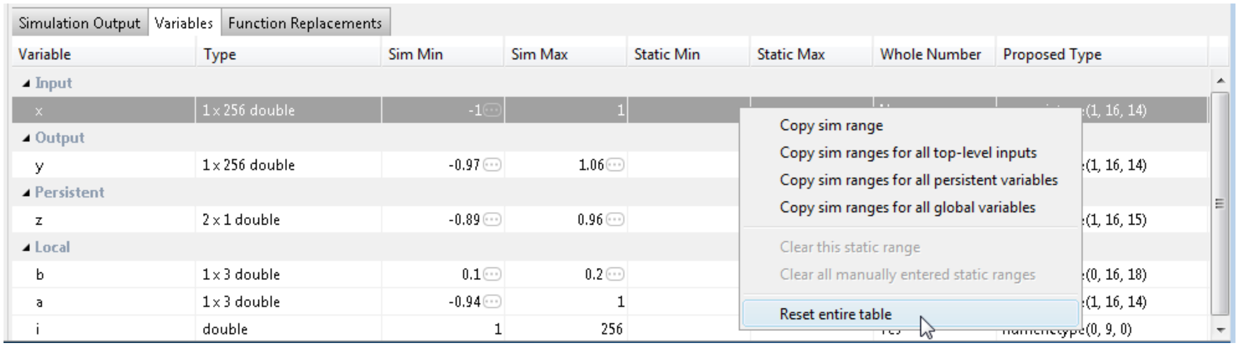
```
function y = fun_with_matlab(x) %#codegen
  persistent z
  if isempty(
      z = ze
  end
  % [b,a] = b
  b = [0.0976
  a = [

  y = zeros(size(x));
```

Original Type:	1 x 256 double
Sim Range:	-1 : 1
Static Range:	:
Proposed Type:	numerictype(1, 16, 14)

If you selected to log data for a histogram, the histogram dynamically updates to reflect the modifications to the proposed type. You can also modify the proposed type in the histogram, see “Histogram” on page 4-58.

Revert Changes

- To clear results and revert edited values, right-click the **Variables** tab and select `Reset entire table`.



Simulation Output
Variables
Function Replacements

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
x	1 x 256 double	-1	1				:(1, 16, 14)
Output							
y	1 x 256 double	-0.97	1.06				:(1, 16, 14)
Persistent							
z	2 x 1 double	-0.89	0.96				:(1, 16, 15)
Local							
b	1 x 3 double	0.1	0.2				:(0, 16, 18)
a	1 x 3 double	-0.94	1				:(1, 16, 14)
i	double	1	256				(0, 9, 0)

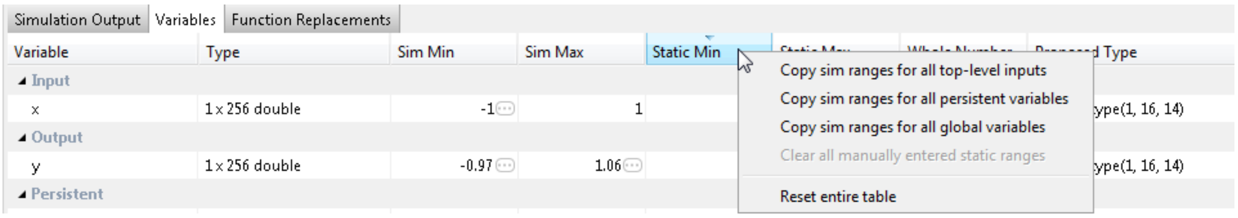
Copy sim range
Copy sim ranges for all top-level inputs
Copy sim ranges for all persistent variables
Copy sim ranges for all global variables
Clear this static range
Clear all manually entered static ranges
Reset entire table

- To revert the type of a selected variable to the type computed by the app, right-click the field and select `Undo changes`.
- To revert changes to variables, right-click the field and select `Undo changes for all variables`.

- To clear a static range value, right-click an edited field and select `Clear this static range`.
- To clear manually entered static range values, right-click anywhere on the **Variables** tab and select `Clear all manually entered static ranges`.

Promote Sim Min and Sim Max Values

With the app, you can promote simulation minimum and maximum values to static minimum and maximum values. This capability is useful if you have not specified static ranges and you have simulated the model with inputs that cover the full intended operating range.


Simulation Output
Variables
Function Replacements
Copy sim ranges for all top-level inputs
Copy sim ranges for all persistent variables
Copy sim ranges for all global variables
Clear all manually entered static ranges
Reset entire table

Variable	Type	Sim Min	Sim Max	Static Min			d Type
◢ Input							
x	1 x 256 double	-1	1				ype(1, 16, 14)
◢ Output							
y	1 x 256 double	-0.97	1.06				ype(1, 16, 14)
◢ Persistent							

To copy:

- A simulation range for a selected variable, select a variable, right-click, and then select `Copy sim range`.
- Simulation ranges for top-level inputs, right-click the Static Min or Static Max column, and then select `Copy sim ranges for all top-level inputs`.
- Simulation ranges for persistent variables, right-click the Static Min or Static Max column, and then select `Copy sim ranges for all persistent variables`.

Automated Fixed-Point Conversion

In this section...

License Requirements

Fixed-point conversion requires the following licenses:

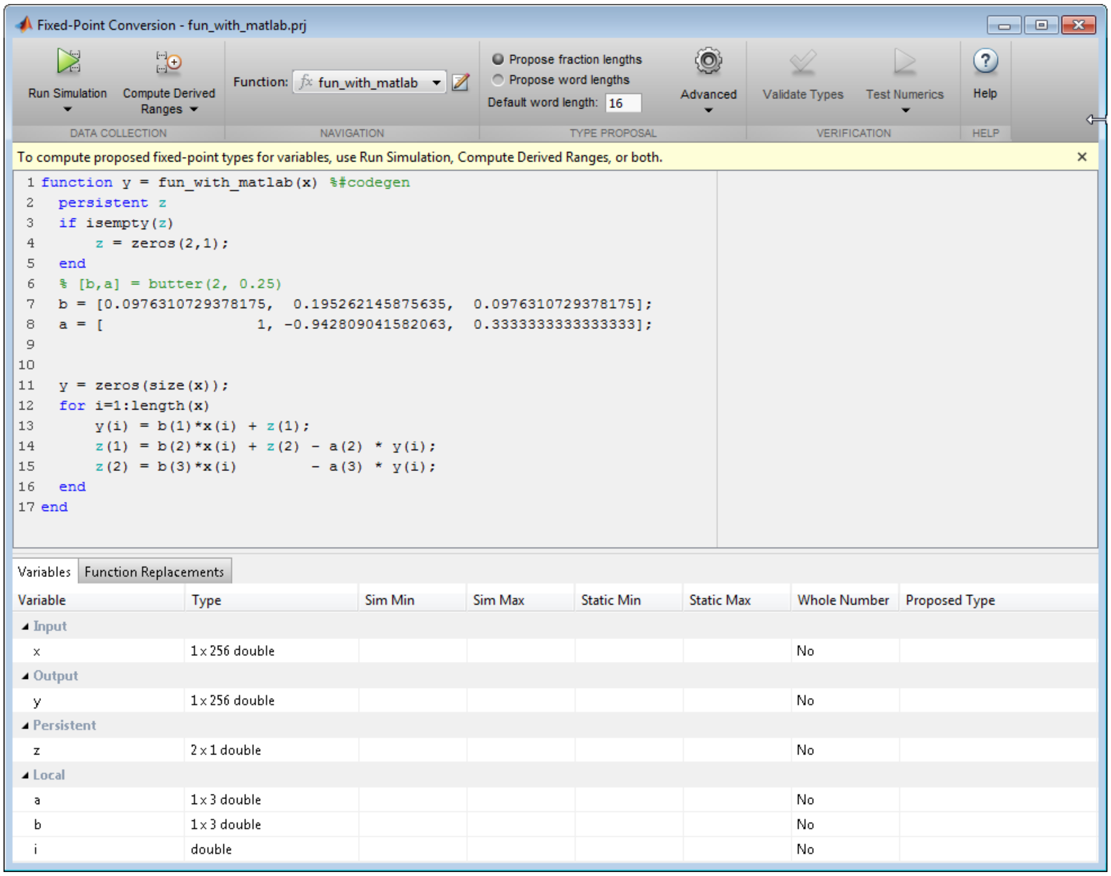
- Fixed-Point Designer
- MATLAB Coder™

Automated Fixed-Point Conversion Capabilities

You can convert floating-point MATLAB code to fixed-point code using the Fixed-Point Conversion tool in HDL Coder projects. You can choose to propose data types based on simulation range data, derived (also known as static) range data, or both.

You can manually enter static ranges. These manually-entered ranges take precedence over simulation ranges and the tool uses them when proposing data types. In addition, you can modify and lock the proposed type so that the tool cannot change it. For more information, see "Locking Proposed Data Types" on page 4-51.

For a list of supported MATLAB features and functions, see "MATLAB Language Features Supported for Automated Fixed-Point Conversion" (MATLAB Coder).



To compute proposed fixed-point types for variables, use Run Simulation, Compute Derived Ranges, or both.

```
function y = fun_with_matlab(x) %#codegen
  persistent z
  if isempty(z)
      z = zeros(2,1);
  end
  % [b,a] = butter(2, 0.25)
  b = [0.0976310729378175,  0.195262145875635,  0.0976310729378175];
  a = [                 1, -0.942809041582063,  0.3333333333333333];


  y = zeros(size(x));
  for i=1:length(x)
      y(i) = b(1)*x(i) + z(1);
      z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
      z(2) = b(3)*x(i)        - a(3) * y(i);
  end
end
```

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
x	1 x 256 double					No	
Output							
y	1 x 256 double					No	
Persistent							
z	2 x 1 double					No	
Local							
a	1 x 3 double					No	
b	1 x 3 double					No	
i	double					No	

During fixed-point conversion, you can:

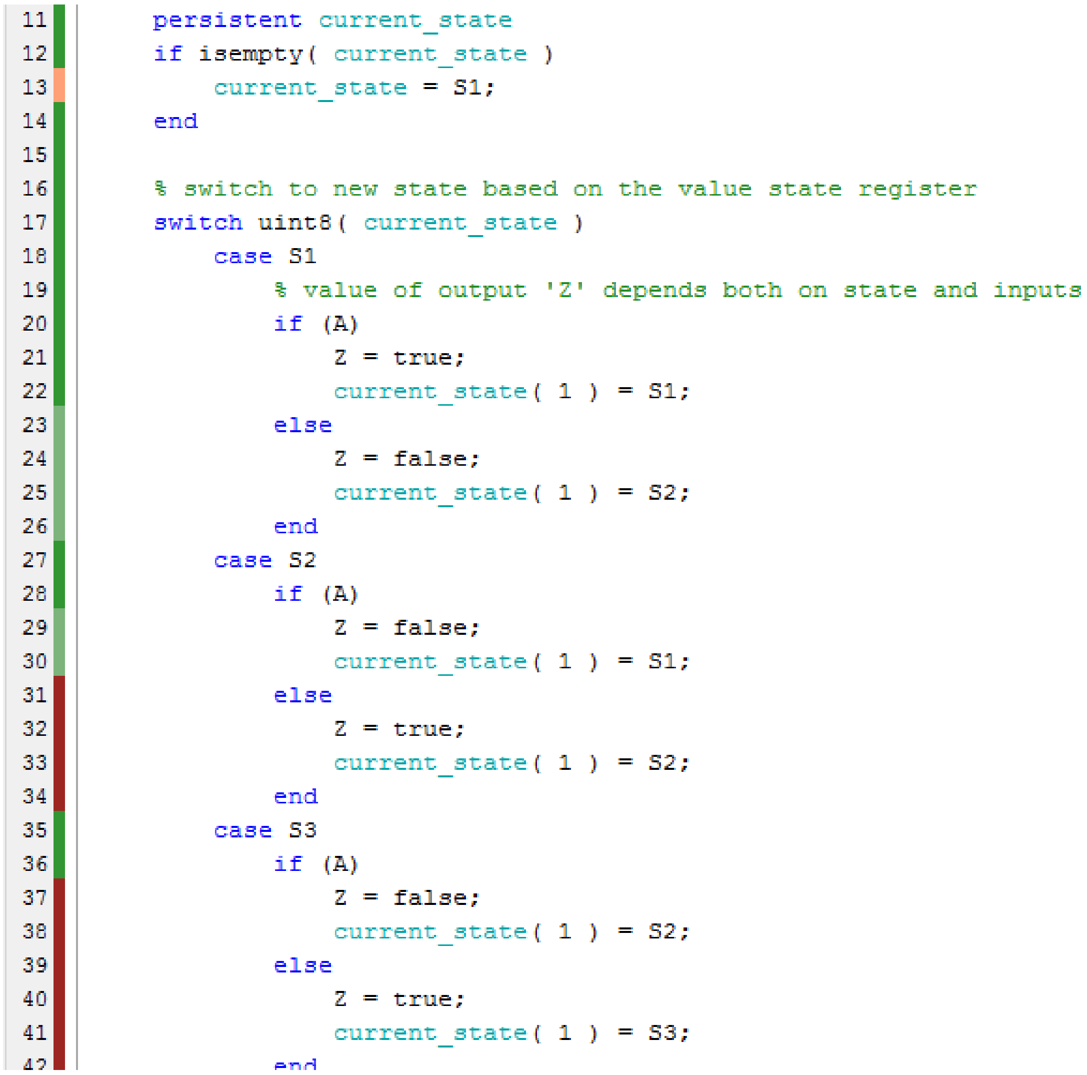
- Verify that your test files cover the full intended operating range of your algorithm using code coverage results.
- Propose fraction lengths based on default word lengths.
- Propose word lengths based on default fraction lengths.
- Optimize whole numbers.
- Specify safety margins for simulation min/max data.
- Validate that you can build your project with the proposed data types.
- Test numerics by running the test bench with the fixed-point types applied.

- View a histogram of bits used by each variable.
- Detect overflows.

Code Coverage

By default, the Fixed-Point Conversion tool shows code coverage results. Your test files must exercise the algorithm over its full operating range so that the simulation ranges are accurate. The quality of the proposed fixed-point data types depends on how well the test files cover the operating range of the algorithm with the accuracy that you want. Reviewing code coverage results helps you verify that your test files are exercising the algorithm adequately. If the code coverage is inadequate, modify the test files or add more test files to increase coverage. If you simulate multiple test files in one run, the tool displays cumulative coverage. However, if you specify multiple test files but run them one at a time, the tool displays the coverage of the file that ran last.

The tool displays a color-coded coverage bar to the left of the code.



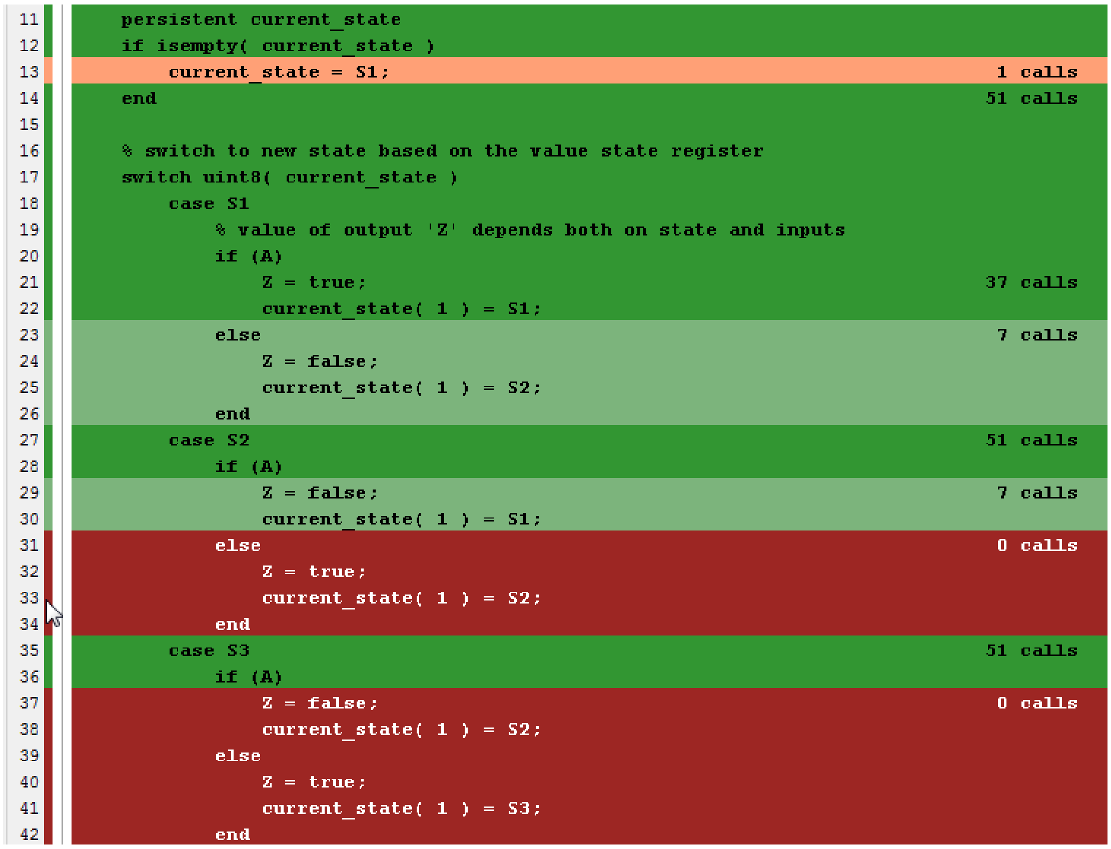
```
    persistent current_state
    if isempty( current_state )
        current_state = S1;
    end

    % switch to new state based on the value state register
    switch uint8( current_state )
        case S1
            % value of output 'Z' depends both on state and inputs
            if (A)
                Z = true;
                current_state( 1 ) = S1;
            else
                Z = false;
                current_state( 1 ) = S2;
            end
        case S2
            if (A)
                Z = false;
                current_state( 1 ) = S1;
            else
                Z = true;
                current_state( 1 ) = S2;
            end
        case S3
            if (A)
                Z = false;
                current_state( 1 ) = S2;
            else
                Z = true;
                current_state( 1 ) = S3;
            end
```

This table describes the color coding.

Coverage Bar Color	Indicates
Green	One of the following situations:

Coverage Bar Color	Indicates
	• The entry-point function executes multiple times and the code executes more than one time. • The entry-point function executes one time and the code executes one time. Different shades of green indicate different ranges of line execution counts. The darkest shade of green indicates the highest range.
Orange	The entry-point function executes multiple times, but the code executes one time.
Red	Code does not execute.

When you place your cursor over the coverage bar, the color highlighting extends over the code. For each section of code, the app displays the number of times that section executes.



```
    persistent current_state
    if isempty( current_state )
        current_state = S1;                                    1 calls
    end                                                       51 calls

    % switch to new state based on the value state register
    switch uint8( current_state )
        case S1
            % value of output 'Z' depends both on state and inputs
            if (A)
                Z = true;                                     37 calls
                current_state( 1 ) = S1;
            else                                               7 calls
                Z = false;
                current_state( 1 ) = S2;
            end
        case S2                                               51 calls
            if (A)
                Z = false;                                     7 calls
                current_state( 1 ) = S1;
            else                                               0 calls
                Z = true;
                current_state( 1 ) = S2;
            end
        case S3                                               51 calls
            if (A)
                Z = false;                                     0 calls
                current_state( 1 ) = S2;
            else
                Z = true;
                current_state( 1 ) = S3;
            end
```

To verify that your test files are testing your algorithm over the intended operating range, review the code coverage results.

Coverage Bar Color	Action
Green	If you expect sections of code to execute more frequently than the coverage shows, either modify the MATLAB code or the test files.
Orange	This behavior is expected for initialization code, for example, the initialization of persistent variables. If you expect the code to execute more than one time, either modify the MATLAB code or the test files.
Red	If the code that does not execute is an error condition, this behavior is acceptable. If you expect the code to execute, either modify the

Coverage Bar Color	Action
	MATLAB code or the test files. If the code is written conservatively and has upper and lower boundary limits, and you cannot modify the test files to reach this code, add static minimum and maximum values. See "Computing Derived Ranges" on page 4-50.

Code coverage is on by default. Turn it off only after you have verified that you have adequate test file coverage. Turning off code coverage can speed up simulation. To turn off code coverage, in the Fixed-Point Conversion tool:

1 Click **Run Simulation**.
2 Clear `Show code coverage`.

Proposing Data Types

The Fixed-Point Conversion tool proposes fixed-point data types based on computed ranges and the word length or fraction length setting. The computed ranges are based on simulation range data, derived range data, or both. If you run a simulation and compute derived ranges, the conversion tool merges the simulation and derived ranges.

Note: You cannot propose data types based on derived ranges for MATLAB classes.

You can manually enter static ranges. These manually-entered ranges take precedence over simulation ranges and the tool uses them when proposing data types. In addition, you can modify and lock the proposed type so that the tool cannot change it. For more information, see "Locking Proposed Data Types" on page 4-51.

Running a Simulation

When you open the Fixed-Point Conversion tool, the tool generates an instrumented MEX function for your MATLAB design. If the build completes without errors, the tool displays compiled information (type, size, complexity) for functions and variables in your code. To navigate to local functions, click the **Functions** tab. If build errors occur, the tool provides error messages that link to the line of code that caused the build issues. You must address these errors before running a simulation. Use the link to navigate to the offending line of code in the MATLAB editor and modify the code to fix the issue. If your code uses functions that are not supported for fixed-point conversion, the tool displays them on the **Function Replacements** tab. See “Function Replacements” on page 4-60.

Before running a simulation, specify the test bench that you want to run. When you run a simulation, the tool runs the test bench, calling the instrumented MEX function. If you modify the MATLAB design code, the tool automatically generates an updated MEX function before running the test bench.

If the test bench runs successfully, the simulation minimum and maximum values and the proposed types are displayed on the **Variables** tab. If you manually enter static ranges for a variable, the manually-entered ranges take precedence over the simulation ranges. If you manually modify the proposed types by typing or using the histogram, the data types are locked so that the tool cannot modify them.

If the test bench fails, the errors are displayed on the **Simulation Output** tab.

The test bench should exercise your algorithm over its full operating range. The quality of the proposed fixed-point data types depends on how well the test bench covers the operating range of the algorithm with the desired accuracy.

Optionally, you can select to log data for histograms. After running a simulation, you can view the histogram for each variable. For more information, see “Histogram” on page 4-58.

Computing Derived Ranges

The advantage of proposing data types based on derived ranges is that you do not have to provide test files that exercise your algorithm over its full operating range. Running such test files often takes a very long time.

To compute derived ranges and propose data types based on these ranges, provide static minimum and maximum values or proposed data types for all input variables. To improve the analysis, enter as much static range information as possible for other variables. You can manually enter ranges or promote simulation ranges to use as static ranges. Manually-entered static ranges always take precedence over simulation ranges.

If you know what data type your hardware target uses, set the proposed data types to match this type. Manually-entered data types are locked so that the tool cannot modify them. The tool uses these data types to calculate the input minimum and maximum values and to derive ranges for other variables. For more information, see “Locking Proposed Data Types” on page 4-51.

When you select **Compute Derived Ranges**, the tool runs a derived range analysis to compute static ranges for variables in your MATLAB algorithm. When the analysis is complete, the static ranges are displayed on the **Variables** tab. If the run produces `+/-Inf` derived ranges, consider defining ranges for all persistent variables.

Optionally, you can select **Quick derived range analysis**. With this option, the conversion tool performs faster static analysis. The computed ranges might be larger than necessary. Select this option in cases where the static analysis takes more time than you can afford.

If the derived range analysis for your project is taking a long time, you can optionally set a timeout. The tool aborts the analysis when the timeout is reached.

Locking Proposed Data Types

You can lock proposed data types against changes by the Fixed-Point Conversion tool using one of the following methods:

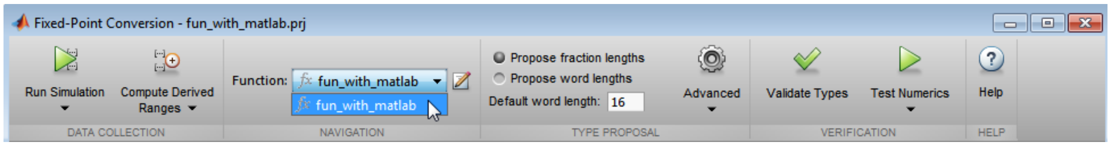
- Manually setting a proposed data type in the Fixed-Point Conversion tool.
- Right-clicking a type proposed by the tool and selecting `Lock computed value`.

The tool displays locked data types in bold so that they are easy to identify. You can unlock a type using one of the following methods:

- Manually overwriting it.
- Right-clicking it and selecting `Undo changes`. This action unlocks only the selected type.
- Right-clicking and selecting `Undo changes for all variables`. This action unlocks all locked proposed types.

Viewing Functions

You can view a list of functions in your project on the **Navigation** pane. This list also includes function specializations and class methods. When you select a function from the list, the MATLAB code for that function or class method is displayed in the Fixed-Point Conversion tool code window.



After conversion, the left pane also displays a list of output files including the fixed-point version of the original algorithm. If your function is not specialized, the conversion retains the original function name in the fixed-point filename and appends the fixed-point suffix. For example, the fixed-point version of `fun_with_matlab.m` is `fun_with_matlab_fixpt.m`.

Viewing Variables

The **Variables** tab provides the following information for each variable in the function selected in the **Navigation** pane:

- **Type** — The original data type of the variable in the MATLAB algorithm.
- **Sim Min** and **Sim Max** — The minimum and maximum values assigned to the variable during simulation.

  You can edit the simulation minimum and maximum values. Edited fields are shown in bold. Editing these fields does not trigger static range analysis, but the tool uses the edited values in subsequent analyses. You can revert to the types proposed by the tool.
- **Static Min** and **Static Max** — The static minimum and maximum values.

  To compute derived ranges and propose data types based on these ranges, provide static minimum and maximum values for all input variables. To improve the analysis, enter as much static range information as possible for other variables.

  When you compute derived ranges, the Fixed-Point Conversion tool runs a static analysis to compute static ranges for variables in your code. When the analysis is complete, the static ranges are displayed. You can edit the computed results. Edited fields are shown in bold. Editing these fields does not trigger static range analysis, but the tool uses the edited values in subsequent analyses. You can revert to the types proposed by the tool.
- **Whole Number** — Whether all values assigned to the variable during simulation are integers.

The Fixed-Point Conversion tool determines whether a variable is always a whole number. You can modify this field. Edited fields are shown in bold. Editing these fields does not trigger static range analysis, but the tool uses the edited values in subsequent analyses. You can revert to the types proposed by the tool.

- The proposed fixed-point data type for the specified word (or fraction) length. Proposed data types use the `numerictype` notation. For example, `numerictype(1,16,12)` denotes a signed fixed-point type with a word length of 16 and a fraction length of 12. `numerictype(0,16,12)` denotes an unsigned fixed-point type with a word length of 16 and a fraction length of 12.

  Because the tool does not apply data types to expressions, it does not display proposed types for them. Instead, it displays their original data types.

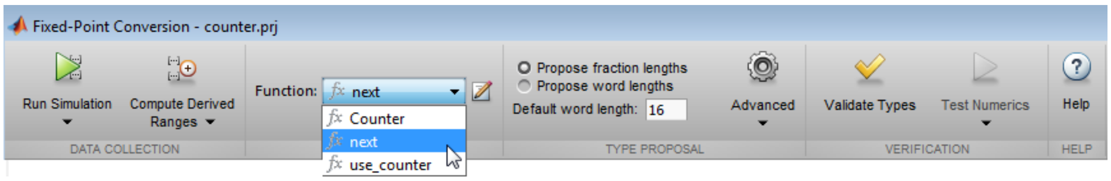
You can also view and edit variable information in the code pane by placing your cursor over a variable name.

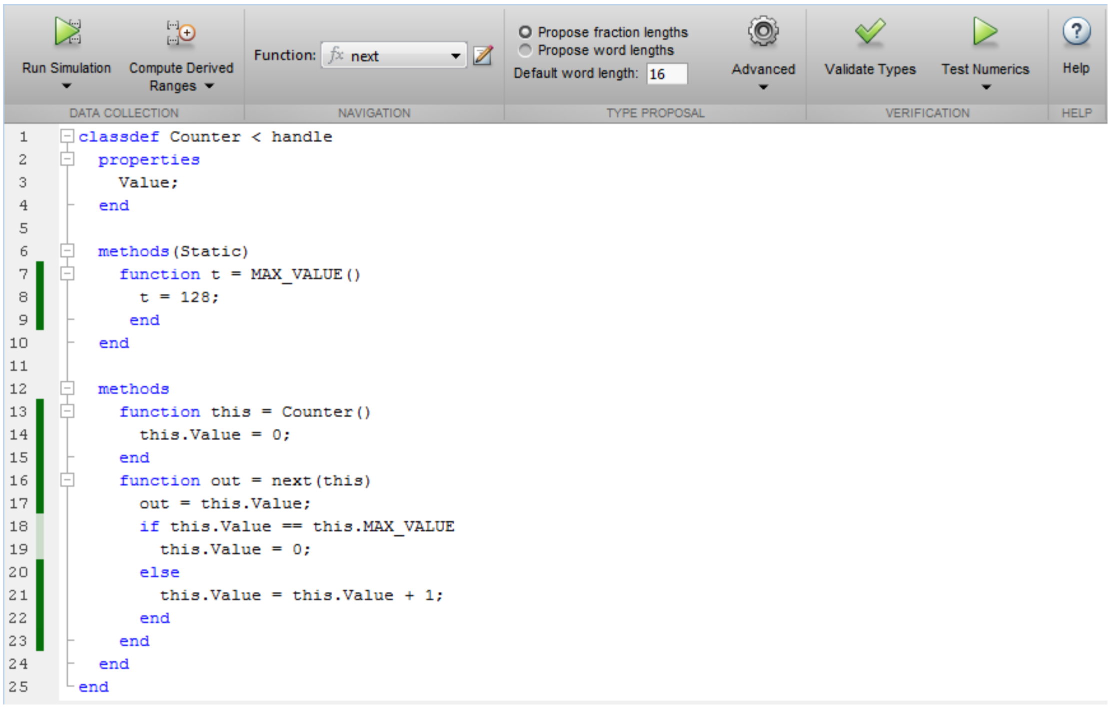
You can use `Ctrl+F` to search for variables in the MATLAB code and on the **Variables** tab. The tool highlights occurrences in the code and displays only the variable with the specified name on the **Variables** tab.

Viewing Information for MATLAB Classes

The tool displays:

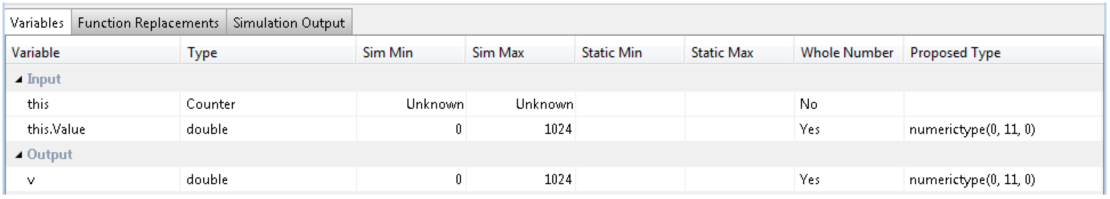
- Code for MATLAB classes and code coverage for class methods in the code window. Use the **Function** list in the Navigation bar to select which class or class method to view.





```
classdef Counter < handle
  properties
    Value;
  end

  methods(Static)
    function t = MAX_VALUE()
      t = 128;
    end
  end

  methods
    function this = Counter()
      this.Value = 0;
    end
    function out = next(this)
      out = this.Value;
      if this.Value == this.MAX_VALUE
        this.Value = 0;
      else
        this.Value = this.Value + 1;
      end
    end
  end
end
```

- Information about MATLAB classes on the **Variables** tab.



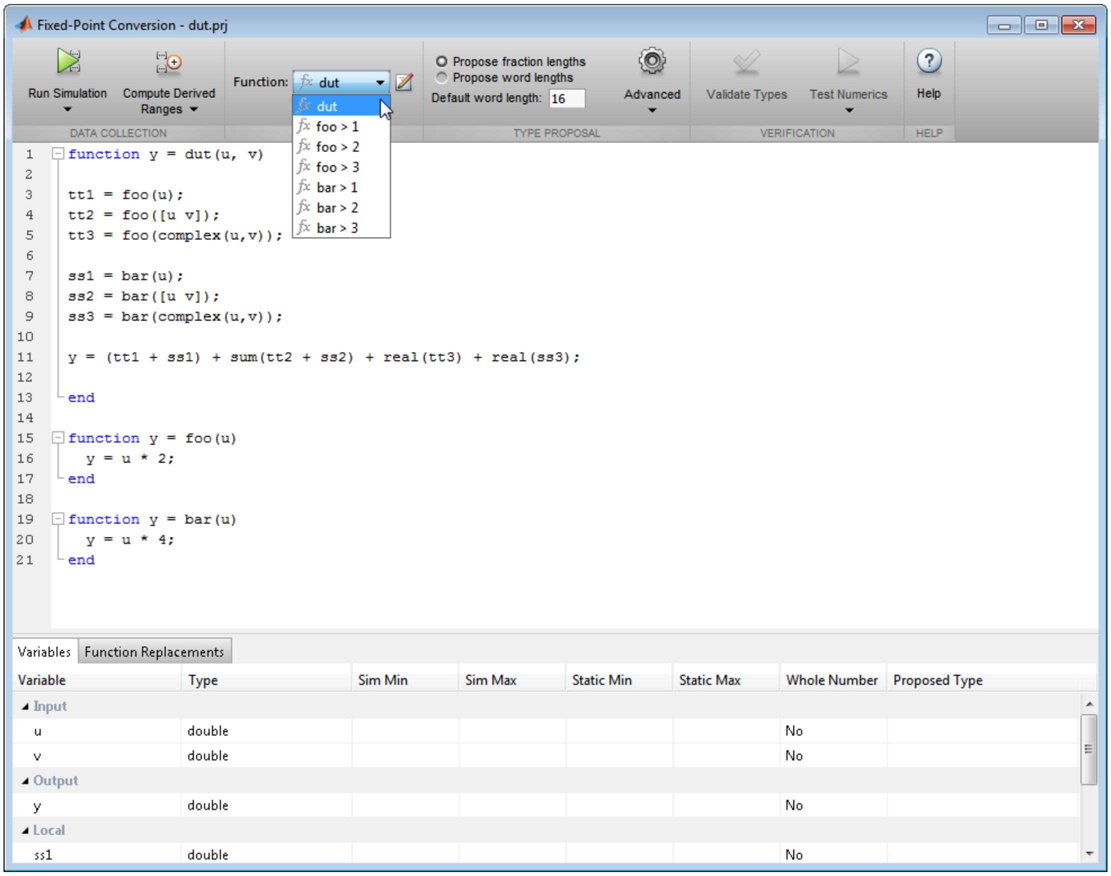
Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
this	Counter	Unknown	Unknown			No	
this.Value	double	0	1024			Yes	numerictype(0, 11, 0)
Output							
v	double	0	1024			Yes	numerictype(0, 11, 0)

Specializations

If a function is specialized, the tool lists each specialization and numbers them sequentially. For example, consider a function, `dut`, that calls subfunctions, `foo` and `bar`, multiple times with different input types.

```
function y = dut(u, v)
```

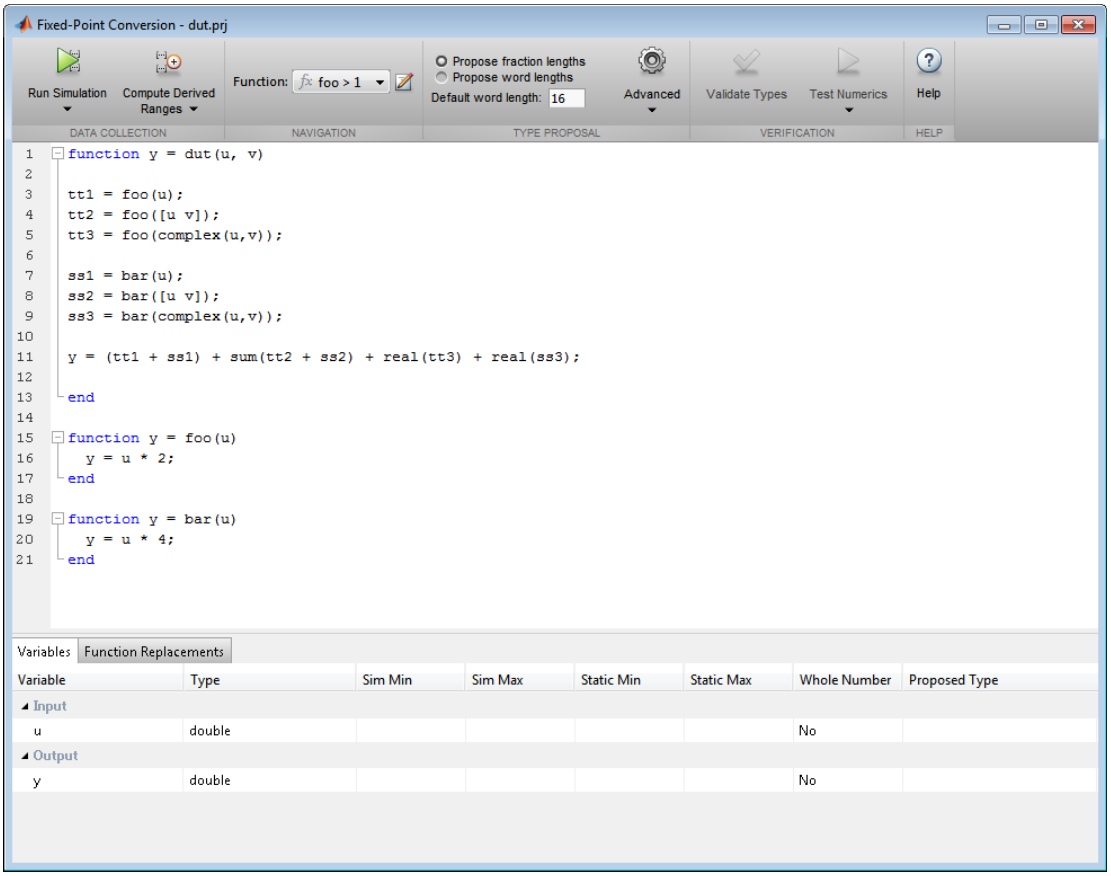
```
tt1 = foo(u);
tt2 = foo([u v]);
tt3 = foo(complex(u,v));

ss1 = bar(u);
ss2 = bar([u v]);
ss3 = bar(complex(u,v));

y = (tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3);

end

function y = foo(u)
  y = u * 2;
end

function y = bar(u)
  y = u * 4;
end
```



```
function y = dut(u, v)

tt1 = foo(u);
tt2 = foo([u v]);
tt3 = foo(complex(u,v));

ss1 = bar(u);
ss2 = bar([u v]);
ss3 = bar(complex(u,v));

y = (tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3);

end

function y = foo(u)
  y = u * 2;
end

function y = bar(u)
  y = u * 4;
end
```

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
u	double					No	
v	double					No	
Output							
y	double					No	
Local							
ss1	double					No	

If you select a specialization, the app displays only the variables used by the specialization.

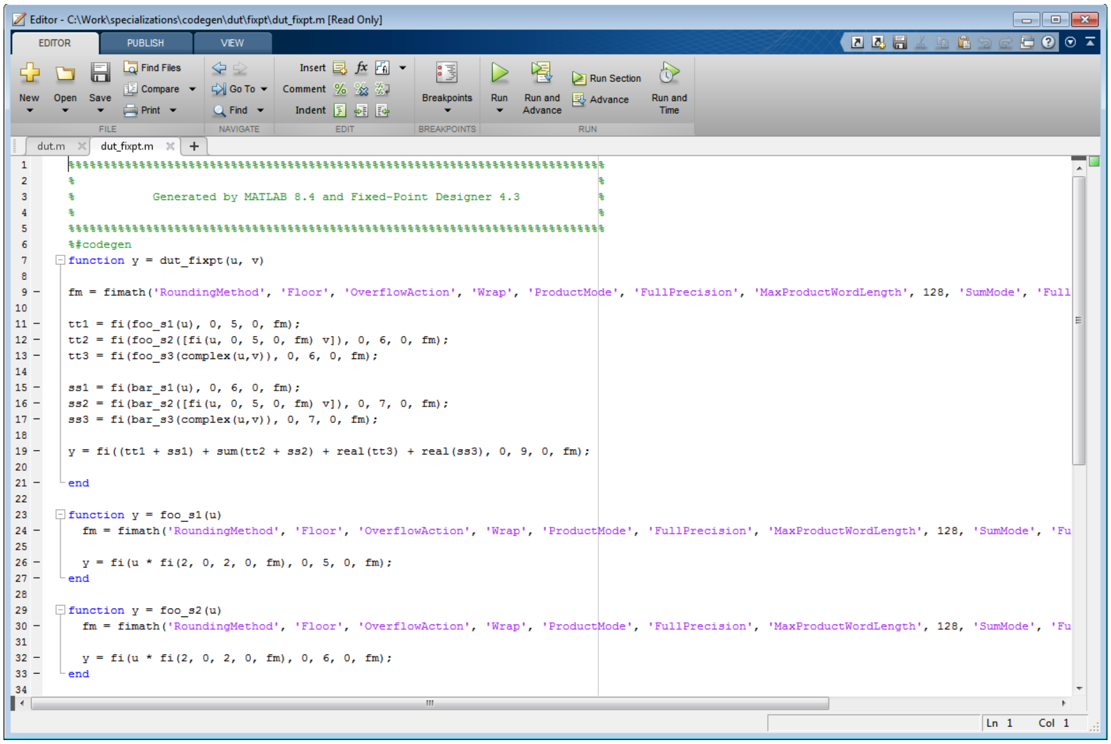


Propose fraction lengths [✓]
Propose word lengths
Default word length: 16

```
function y = dut(u, v)

tt1 = foo(u);
tt2 = foo([u v]);
tt3 = foo(complex(u,v));

ss1 = bar(u);
ss2 = bar([u v]);
ss3 = bar(complex(u,v));

y = (tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3);

end

function y = foo(u)
  y = u * 2;
end

function y = bar(u)
  y = u * 4;
end
```

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
u	double					No	
Output							
y	double					No	

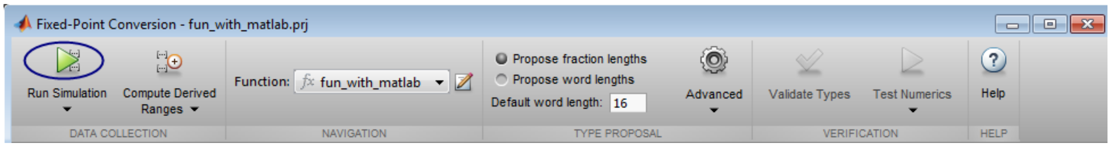
In the generated fixed-point code, the number of each fixed-point specialization matches the number in the Source Code list which makes it easy to trace between the floating-point and fixed-point versions of your code. For example, the generated fixed-point function for `foo > 1` is named `foo_s1`.



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                          %
%           Generated by MATLAB 8.4 and Fixed-Point Designer 4.3           %
%                                                                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%#codegen
function y = dut_fixpt(u, v)

fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'Full

tt1 = fi(foo_s1(u), 0, 5, 0, fm);
tt2 = fi(foo_s2([fi(u, 0, 5, 0, fm) v]), 0, 6, 0, fm);
tt3 = fi(foo_s3(complex(u,v)), 0, 6, 0, fm);

ss1 = fi(bar_s1(u), 0, 6, 0, fm);
ss2 = fi(bar_s2([fi(u, 0, 5, 0, fm) v]), 0, 7, 0, fm);
ss3 = fi(bar_s3(complex(u,v)), 0, 7, 0, fm);

y = fi((tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3), 0, 9, 0, fm);

end

function y = foo_s1(u)
  fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'Fu

  y = fi(u * fi(2, 0, 2, 0, fm), 0, 5, 0, fm);
end

function y = foo_s2(u)
  fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'Fu

  y = fi(u * fi(2, 0, 2, 0, fm), 0, 6, 0, fm);
end

```

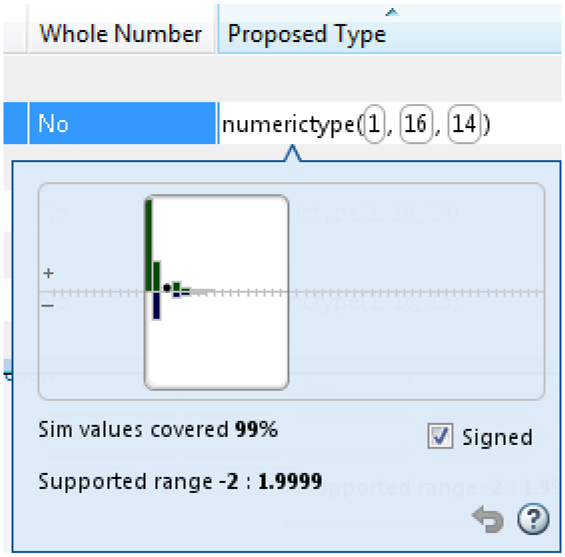
Histogram

To log data for histograms, in the Fixed-Point Conversion window, click **Run Simulation** and select `Log data for histogram`, and then click the Run Simulation button.



After simulation, to view the histogram for a variable, on the **Variables** tab, click the **Proposed Type** field for that variable.

The histogram provides the range of the proposed data type and the percentage of simulation values that the proposed data type covers. The bit weights are displayed along the X-axis, and the percentage of occurrences along the Y-axis. Each bin in the histogram corresponds to a bit in the binary word. For example, this histogram displays the range for a variable of type `numerictype(1,16,14)`.

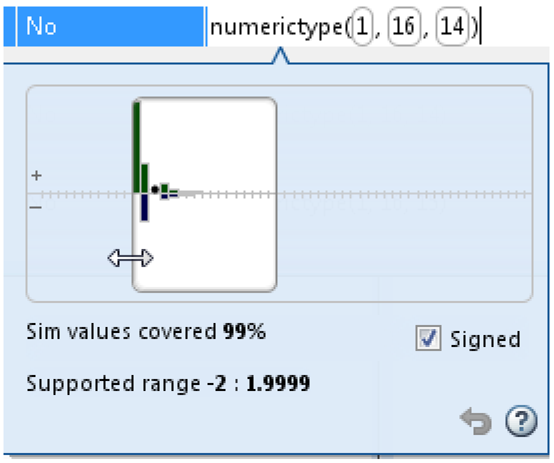


Sim values covered **99%**

Supported range **-2 : 1.9999**

You can view the effect of changing the proposed data types by:

- Dragging the edges of the bounding box in the histogram window to change the proposed data type.



Sim values covered 99%

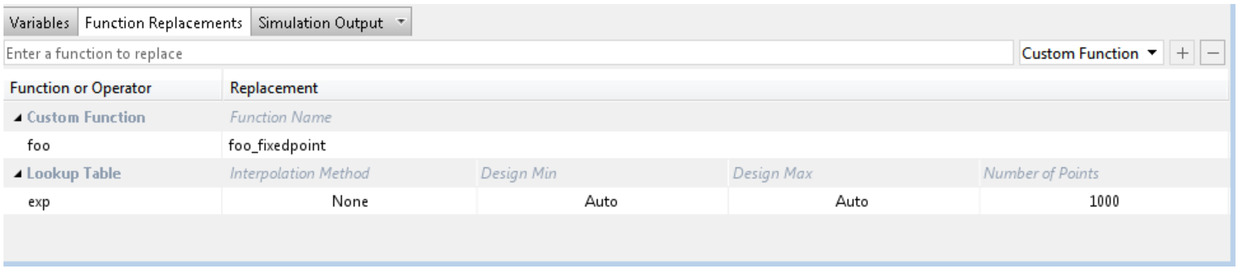
Supported range -2 : 1.9999

- Selecting or clearing **Signed**.

To revert to the types proposed by the automatic conversion, in the histogram window, click ↶.

Function Replacements

If your MATLAB code uses functions that do not have fixed-point support, the tool lists these functions on the **Function Replacements** tab. You can choose to replace unsupported functions with a custom function replacement or with a lookup table.



Function or Operator	Replacement			
Custom Function	Function Name			
foo	foo_fixedpoint			
Lookup Table	Interpolation Method	Design Min	Design Max	Number of Points
exp	None	Auto	Auto	1000

You can add and remove function replacements from this list. If you enter a function replacements for a function, the replacement function is used when you build the project. If you do not enter a replacement, the tool uses the type specified in the original MATLAB code for the function.

Note: Using this table, you can replace the names of the functions but you cannot replace argument patterns.

Validating Types

Selecting **Validate Types** validates the build using the proposed fixed-point data types. If the validation is successful, you are ready to test the numerical behavior of the fixed-point MATLAB algorithm.

If the errors or warnings occur during validation, they are displayed on the **Type Validation Output** tab. If errors or warning occur:

- On the **Variables** tab, inspect the proposed types and manually modified types to verify that they are valid.
- On the **Function Replacements** tab, verify that you have provided function replacements for unsupported functions.

Testing Numerics

After validating the proposed fixed-point data types, select **Test Numerics** to verify the behavior of the fixed-point MATLAB algorithm. By default, if you added a test bench to define inputs or run a simulation, the tool uses this test bench to test numerics. The tool compares the numerical behavior of the generated fixed-point MATLAB code with the original floating-point MATLAB code. If you select to log inputs and outputs for comparison plots, the tool generates an additional plot for each scalar output. This plot shows the floating-point and fixed-point results and the difference between them. For non-scalar outputs, only the error information is shown.

If the numerical results do not meet your desired accuracy after fixed-point simulation, modify fixed-point data type settings and repeat the type validation and numerical testing steps. You might have to iterate through these steps multiple times to achieve the desired results.

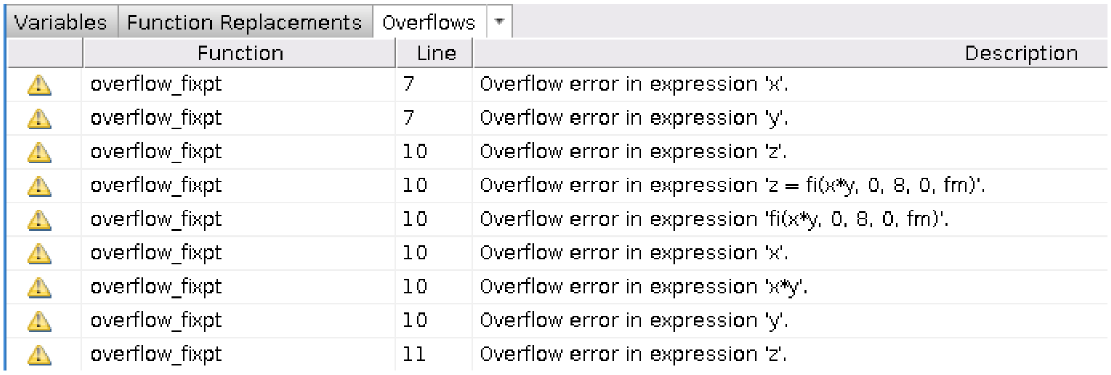
Detecting Overflows

When testing numerics, selecting **Use scaled doubles to detect overflows** enables overflow detection. When this option is selected, the conversion tool runs the simulation using scaled double versions of the proposed fixed-point types. Because scaled doubles store their data in double-precision floating-point, they carry out arithmetic in full range.

They also retain their fixed-point settings, so they are able to report when a computation goes out of the range of the fixed-point type. .

If the tool detects overflows, on its Overflow tab, it provides:

- A list of variables and expressions that overflowed
- Information on how much each variable overflowed
- A link to the variables or expressions in the code window



	Function	Line	Description
⚠	overflow_fixpt	7	Overflow error in expression 'x'.
⚠	overflow_fixpt	7	Overflow error in expression 'y'.
⚠	overflow_fixpt	10	Overflow error in expression 'z'.
⚠	overflow_fixpt	10	Overflow error in expression 'z = fi(x*y, 0, 8, 0, fm)'.
⚠	overflow_fixpt	10	Overflow error in expression 'fi(x*y, 0, 8, 0, fm)'.
⚠	overflow_fixpt	10	Overflow error in expression 'x'.
⚠	overflow_fixpt	10	Overflow error in expression 'x*y'.
⚠	overflow_fixpt	10	Overflow error in expression 'y'.
⚠	overflow_fixpt	11	Overflow error in expression 'z'.

If your original algorithm uses scaled doubles, the tool also provides overflow information for these expressions.

See Also

“Detect Overflows” (MATLAB Coder)

Custom Plot Functions

The Fixed-Point Conversion tool provides a default time series based plotting function. The conversion process uses this function at the test numerics step to show the floating-point and fixed-point results and the difference between them. However, during fixed-point conversion you might want to visualize the numerical differences in a view that is more suitable for your application domain. For example, plots that show eye diagrams and bit error differences are more suitable in the communications domain and histogram difference plots are more suitable in image processing designs.

You can choose to use a custom plot function at the test numerics step. The Fixed-Point Conversion tool facilitates custom plotting by providing access to the raw logged input and output data before and after fixed-point conversion. You supply a custom plotting function to visualize the differences between the floating-point and fixed-point results. If you specify a custom plot function, the fixed-point conversion process calls the function for each input and output variable, passes in the name of the variable and the function that uses it, and the results of the floating-point and fixed-point simulations.

Your function should accept three inputs:

- A structure that holds the name of the variable and the function that uses it.

  Use this information to:

  - Customize plot headings and axes.
  - Choose which variables to plot.
  - Generate different error metrics for different output variables.
- A cell array to hold the logged floating-point values for the variable.

  This cell array contains values observed during floating-point simulation of the algorithm during the test numerics phase. You might need to reformat this raw data.
- A cell array to hold the logged values for the variable after fixed-point conversion.

  This cell array contains values observed during fixed-point simulation of the converted design.

For example, `function customComparisonPlot(varInfo, floatVarVals, fixedPtVarVals)`.

To use a custom plot function, in the Fixed-Point Conversion tool, select **Advanced**, and then set **Custom plot function** to the name of your plot function.

In the programmatic workflow, set the coder.FixptConfig configuration object `PlotFunction` property to the name of your plot function. See “Visualize Differences Between Floating-Point and Fixed-Point Results” on page 4-65.

Visualize Differences Between Floating-Point and Fixed-Point Results

This example shows how to configure the `codegen` function to use a custom plot function to compare the behavior of the generated fixed-point code against the behavior of the original floating-point MATLAB code.

By default, when the `LogIOForComparisonPlotting` option is enabled, the conversion process uses a time series based plotting function to show the floating-point and fixed-point results and the difference between them. However, during fixed-point conversion you might want to visualize the numerical differences in a view that is more suitable for your application domain. This example shows how to customize plotting and produce scatter plots at the test numerics step of the fixed-point conversion.

Prerequisites

To complete this example, you must install the following products:

- MATLAB
- Fixed-Point Designer
- C compiler (for most platforms, a default C compiler is supplied with MATLAB) For a list of supported compilers, see `http://www.mathworks.com/support/compilers/current_release/`

  You can use `mex -setup` to change the default compiler. See “Change Default Compiler” (MATLAB).

Create a New Folder and Copy Relevant Files

1. Create a local working folder, for example, `c:\custom_plot`.
2. Change to the `docroot\toolbox\fixpoint\examples` folder. At the MATLAB command line, enter:

   ```
   cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
   ```
3. Copy the `myFilter.m`, `myFilterTest.m`, `plotDiff.m`, and `filterData.mat` files to your local working folder.

Type	Name	Description
Function code	`myFilter.m`	Entry-point MATLAB function

Type	Name	Description
Test file	`myFilterTest.m`	MATLAB script that tests `myFilter.m`
Plotting function	`plotDiff.m`	Custom plot function
MAT-fiile	`filterData.mat`	Data to filter.

The myFilter Function

```
function [y, ho] = myFilter(in)

persistent b h;
if isempty(b)
  b = complex(zeros(1,16));
  h = complex(zeros(1,16));
  h(8) = 1;
end

b = [in, b(1:end-1)];
y = b*h.';

errf = 1-sqrt(real(y)*real(y) + imag(y)*imag(y));
update = 0.001*conj(b)*y*errf;

h = h + update;
h(8) = 1;
ho = h;

end
```

The myFilterTest File

```
% load data
data = load('filterData.mat');
d = data.symbols;

for idx = 1:4000
    y = myFilter(d(idx));
end
```

The plotDiff Function

```
% varInfo - structure with information about the variable. It has the following fields
%             i) name
%              ii) functionName
% floatVals - cell array of logged original values for the 'varInfo.name' variable
% fixedVals - cell array of logged values for the 'varInfo.name' variable after
%                Fixed-Point conversion.
function plotDiff(varInfo, floatVals, fixedVals)
    varName = varInfo.name;
    fcnName = varInfo.functionName;

    % convert from cell to matrix
    floatVals = cell2mat(floatVals);
    fixedVals = cell2mat(fixedVals);

    % escape the '_'s because plot titles treat these as subscripts
    escapedVarName = regexprep(varName,'_','\\_');
    escapedFcnName = regexprep(fcnName,'_','\\_');

    % flatten the values
    flatFloatVals = floatVals(1:end);
    flatFixedVals = fixedVals(1:end);

    % build Titles
    floatTitle = [ escapedFcnName ' > ' 'float : ' escapedVarName ];
    fixedTitle = [ escapedFcnName ' > ' 'fixed : ' escapedVarName ];

    data = load('filterData.mat');

    switch varName
        case 'y'
            x_vec = data.symbols;

            figure('Name', 'Comparison plot', 'NumberTitle', 'off');

            % plot floating point values
            y_vec = flatFloatVals;
            subplot(1, 2, 1);
            plotScatter(x_vec, y_vec, 100, floatTitle);

            % plot fixed point values
            y_vec = flatFixedVals;
            subplot(1, 2, 2);
            plotScatter(x_vec, y_vec, 100, fixedTitle);

        otherwise
            % Plot only output 'y' for this example, skip the rest
    end

end

function plotScatter(x_vec, y_vec, n, figTitle)
    % plot the last n samples
    x_plot = x_vec(end-n+1:end);
    y_plot = y_vec(end-n+1:end);

    hold on
    scatter(real(x_plot),imag(x_plot), 'bo');

    hold on
    scatter(real(y_plot),imag(y_plot), 'rx');

    title(figTitle);
end
```

Set Up Configuration Object

1 Create a `coder.FixptConfig` object.

```
fxptcfg = coder.config('fixpt');
```

2 Specify the test file name and custom plot function name. Enable logging and numerics testing.

```
fxptcfg.TestBenchName = 'myFilterTest';
fxptcfg.PlotFunction = 'plotDiff';
fxptcfg.TestNumerics = true;
fxptcfg. LogIOForComparisonPlotting = true;
fxptcfg.DefaultWordLength = 16;
```

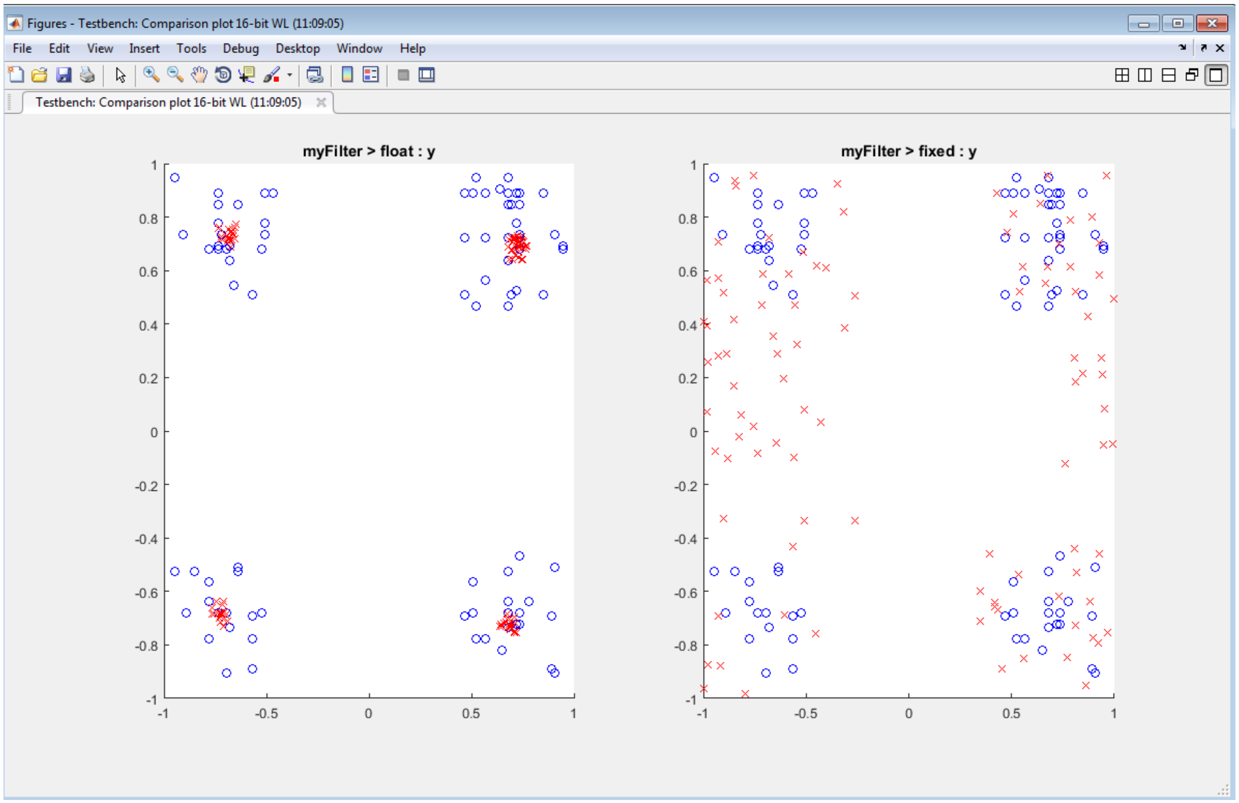
Convert to Fixed Point

Convert the floating-point MATLAB function, `myFilter`, to fixed-point MATLAB code. You do not need to specify input types for the `codegen` command because it infers the types from the test file.

```
codegen -args {complex(0, 0)} -float2fixed fxptcfg myFilter
```

The conversion process generates fixed-point code using a default word length of `16` and then runs a fixed-point simulation by running the `myFilterTest.m` function and calling the fixed-point version of `myFilter.m`.

Because you selected to log inputs and outputs for comparison plots and to use the custom plotting function, `plotDiff.m`, for these plots, the conversion process uses this function to generate the comparison plot.

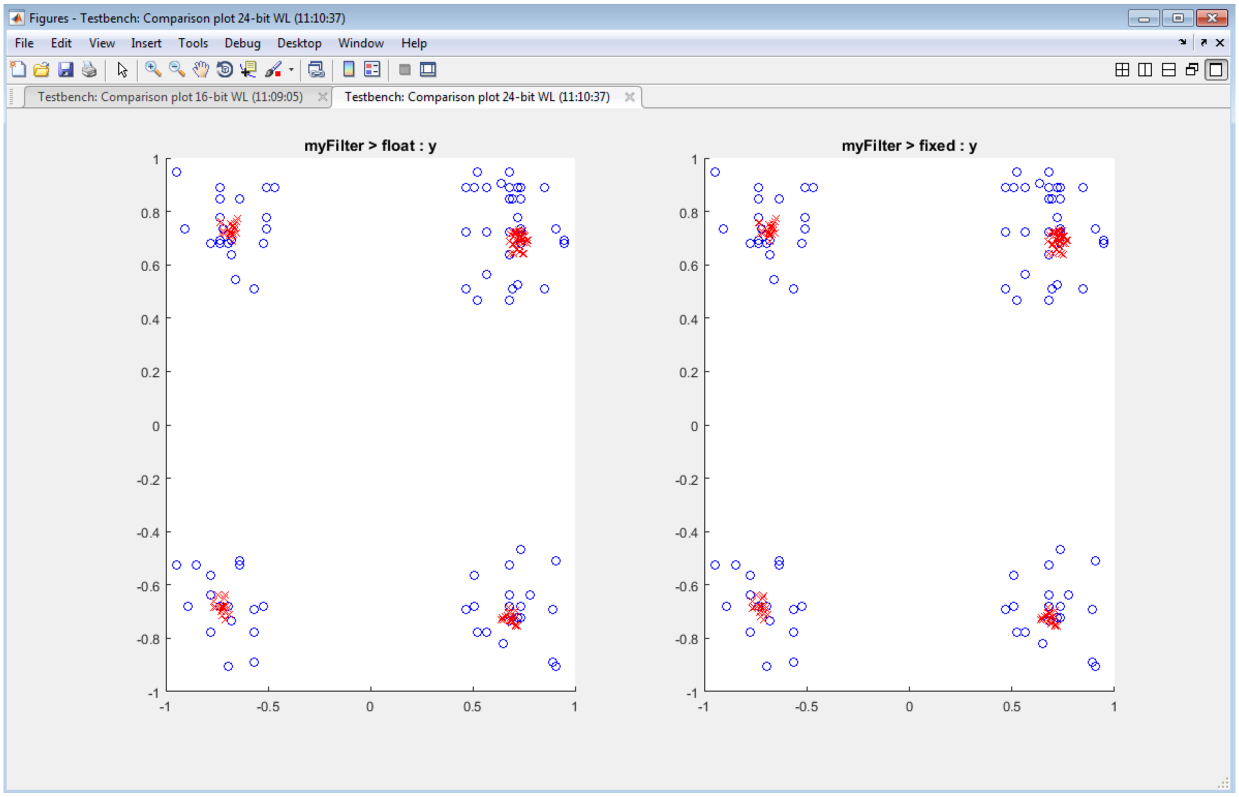

Figures - Testbench: Comparison plot 16-bit WL (11:09:05)
File Edit View Insert Tools Debug Desktop Window Help
Testbench: Comparison plot 16-bit WL (11:09:05)
myFilter > float : y
myFilter > fixed : y
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.5
0
0.5
1

The plot shows that the fixed-point results do not closely match the floating-point results.

Increase the word length to `24` and then convert to fixed point again.

```
fxptcfg.DefaultWordLength = 24;
codegen -args {complex(0, 0)} -float2fixed fxptcfg myFilter
```

The increased word length improved the results. This time, the plot shows that the fixed-point results match the floating-point results.


Figures - Testbench: Comparison plot 24-bit WL (11:10:37)
File Edit View Insert Tools Debug Desktop Window Help
Testbench: Comparison plot 16-bit WL (11:09:05)
Testbench: Comparison plot 24-bit WL (11:10:37)
myFilter > float : y
myFilter > fixed : y
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.5
0
0.5
1

Inspecting Data Using the Simulation Data Inspector

In this section...

What Is the Simulation Data Inspector?

The Simulation Data Inspector allows you to view data logged during the fixed-point conversion process. You can use it to inspect and compare the inputs and outputs to the floating-point and fixed-point versions of your algorithm.

For fixed-point conversion, there is no programmatic interface for the Simulation Data Inspector.

Import Logged Data

Before importing data into the Simulation Data Inspector, you must have previously logged data to the base workspace or to a MAT-file.

Export Logged Data

The Simulation Data Inspector provides the capability to save data collected by the fixed-point conversion process to a MAT-file that you can later reload. The format of the MAT-file is different from the format of a MAT-file created from the base workspace.

Group Signals

You can customize the organization of your logged data in the Simulation Data Inspector **Runs** pane. By default, data is first organized by run. You can then organize your data by logged variable or no hierarchy.

Run Options

You can configure the Simulation Data Inspector to:

- Append New Runs

  In the Run Options dialog box, the default is set to add new runs to the bottom of the run list. To append new runs to the top of the list, select **Add new runs at top**.
- Specify a Run Naming Rule

  To specify run naming rules, in the Simulation Data Inspector toolbar, click **Run Options**.

Create Report

You can create a report of the runs or comparison plots. Specify the name and location of the report file. By default, the Simulation Data Inspector overwrites existing files. To preserve existing reports, select **If report exists, increment file name to prevent overwriting**.

Comparison Options

To change how signals are matched when runs are compared, specify the **Align by** and **Then by** parameters and then click **OK**.

Enabling Plotting Using the Simulation Data Inspector

To enable the Simulation Data Inspector, see “Enable Plotting Using the Simulation Data Inspector” on page 4-74.

Save and Load Simulation Data Inspector Sessions

If you have data in the Simulation Data Inspector and you want to archive or share the data to view in the Simulation Data Inspector later, save the Simulation Data Inspector session. When you save a Simulation Data Inspector session, the MAT-file contains:

- All runs, data, and properties from the **Runs** and **Comparisons** panes.
- Check box selection state for data in the **Runs** pane.

Save a Session to a MAT-File

1. On the **Visualize** tab, click **Save**.
2. Browse to where you want to save the MAT-file to, name the file, and click **Save**.

Load a Saved Simulation Data Inspector Simulation

1. On the **Visualize** tab, click **Open**.
2. Browse, select the MAT-file saved from the Simulation Data Inspector, and click **Open**.
3. If data in the session is plotted on multiple subplots, on the **Format** tab, click **Subplots** and select the subplot layout.

Enable Plotting Using the Simulation Data Inspector

In this section...
"From the UI" on page 4-74
"From the Command Line" on page 4-74

From the UI

You can use the Simulation Data Inspector to inspect and compare floating-point and fixed-point logged input and output data. In the Fixed-Point Conversion tool:

1 Click **Advanced**.
2 In the Advanced Settings dialog box, set **Plot with Simulation Data Inspector** to `Yes`.
3 At the Test Numerics stage in the conversion process, click **Test Numerics**, select `Log inputs and outputs for comparison plots`, and then click ▶.

For an example, see "Propose Fixed-Point Data Types Based on Derived Ranges" (MATLAB Coder).

From the Command Line

You can use the Simulation Data Inspector to inspect and compare floating-point and fixed-point input and output data logged using the function. At the MATLAB command line:

1 Create a fixed-point configuration object and configure the test file name.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'dti_test';
```

2 Select to run the test file to verify the generated fixed-point MATLAB code. Log inputs and outputs for comparison plotting and select to use the Simulation Data Inspector to plot the results.

```
fixptcfg.TestNumerics = true;
fixptcfg.LogIOForComparisonPlotting = true;
fixptcfg.PlotWithSimulationDataInspector = true;
```

3 Generate fixed-point MATLAB code using `codegen`.

```
codegen -float2fixed fixptcfg -config cfg dti
```

For an example, see “Propose Fixed-Point Data Types Based on Derived Ranges” (MATLAB Coder).

Replacing Functions Using Lookup Table Approximations

The software provides an option to generate lookup table approximations for continuous and stateless single-input, single-output functions in your original MATLAB code. These functions must be on the MATLAB path.

You can use this capability to handle functions that are not supported for fixed point and to replace your own custom functions. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. You can control the interpolation method and number of points in the lookup table. By adjusting these settings, you can tune the behavior of replacement function to match the behavior of the original function as closely as possible.

The fixed-point conversion process generates one lookup table approximation per call site of the function that needs replacement.

To use lookup table approximations, see:

- `coder.approximation`
- "Replace the exp Function with a Lookup Table" on page 4-85
- "Replace a Custom Function with a Lookup Table" on page 4-77

Replace a Custom Function with a Lookup Table

In this section...

With HDL Coder, you can generate lookup table approximations for functions that do not support fixed-point types, and replace your own functions. To replace a custom function with a Lookup Table, use the HDL Coder app, or the `fiaccel codegen` function.

Using the HDL Coder App

This example shows how to replace a custom function with a Lookup Table using the **HDL Coder** app.

Create Algorithm and Test Files

In a local, writable folder:

1. Create a MATLAB function, `custom_fcn`, which is the function that you want to replace.

```
function y = custom_fcn(x)
  y = 1./(1+exp(-x));
end
```

2. Create a wrapper function that calls `custom_fcn`.

```
function y = call_custom_fcn(x)
  y = custom_fcn(x);
end
```

3. Create a test file, `custom_test`, which uses `call_custom_fcn`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
   y(itr) = call_custom_fcn( x(itr) );
end
plot( x, y );
```

Create and Set up a HDL Coder Project

1 Navigate to the work folder that contains the file for this example.

2 To open the **HDL Coder** app, in the MATLAB command prompt, enter `hdlcoder`. Set **Name** to `custom_project.prj` and click **OK**. The project opens in the MATLAB workspace.

3 In the project window, on the **MATLAB Function** tab, click the **Add MATLAB function** link. Browse to the file `call_custom_fcn.m`, and then click **OK** to add the file to the project.

Define Input Types

1 To define input types for `call_custom_fcn.m`, on the **MATLAB Function** tab, click **Autodefine types**.

2 Add `custom_test` as a test file, and then click **Run**.

   From the test file, HDL Coder determines that `x` is a scalar double.

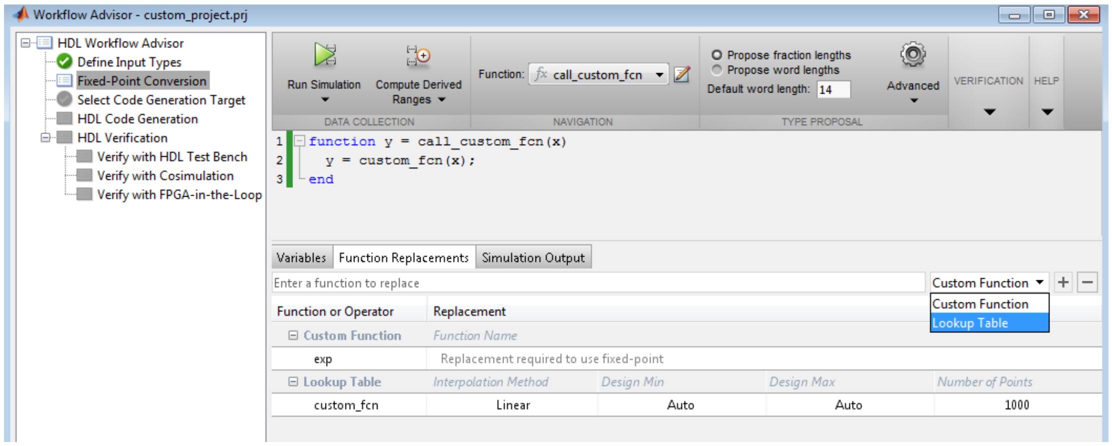
3 Click **Use These Types**.

Replace custom_fcn with Lookup Table

1 To open the HDL Workflow Advisor, click **Workflow Advisor**, and in the Workflow Advisor window, click **Fixed-Point Conversion**.

2 To replace `custom_fcn` with a Lookup Table, on the **Function Replacements** tab, enter `custom_fcn`, select `Lookup Table`, and then click `+`.

   By default, the lookup table uses linear interpolation, 1000 points, and design minimum and maximum values that the app detects by running a simulation or computing derived ranges.



Function: call_custom_fcn

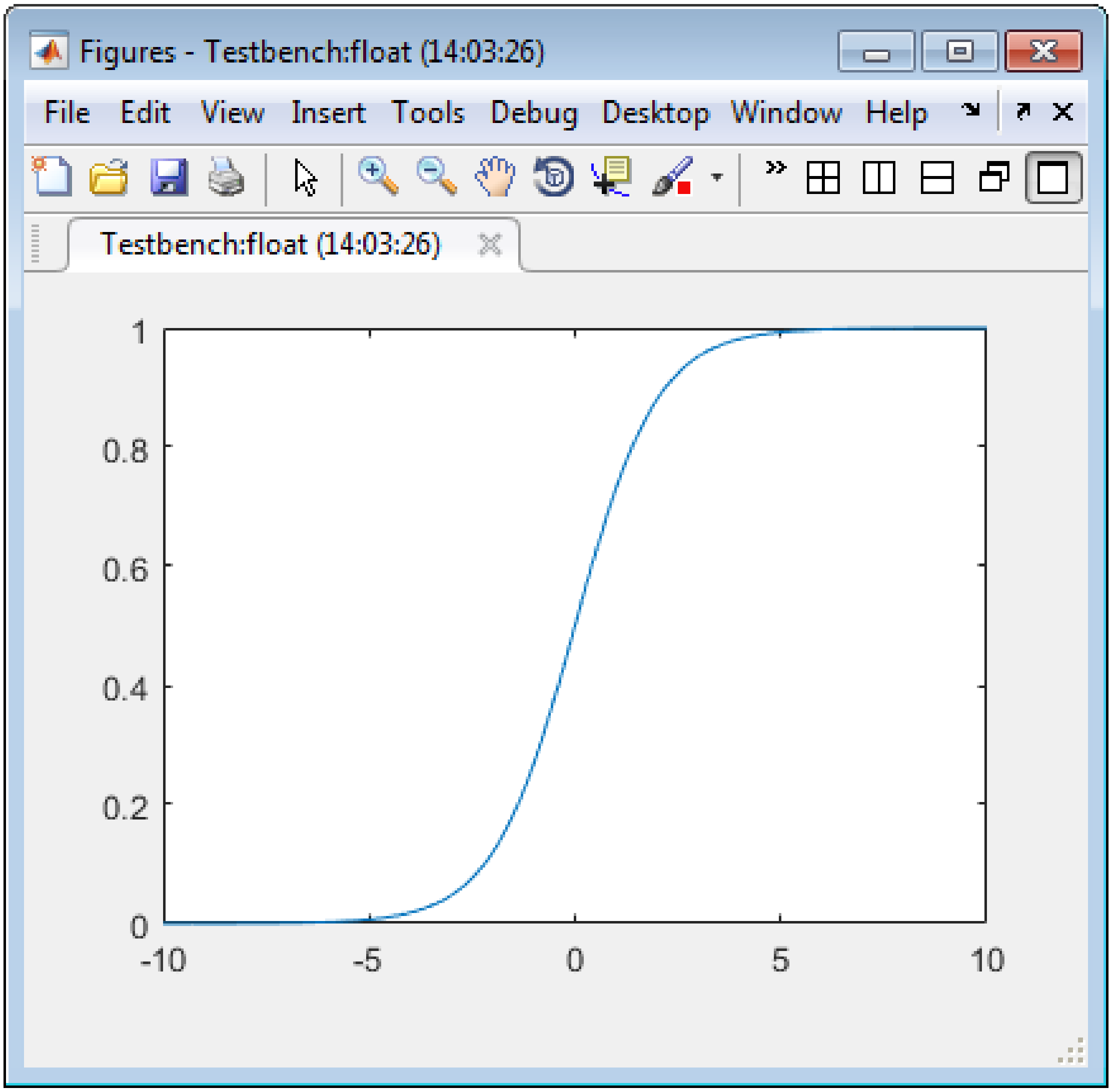
Propose fraction lengths
Propose word lengths
Default word length: 14

```
function y = call_custom_fcn(x)
  y = custom_fcn(x);
end
```

Function or Operator	Replacement			
⊟ Custom Function	Function Name			
exp	Replacement required to use fixed-point			
⊟ Lookup Table	Interpolation Method	Design Min	Design Max	Number of Points
custom_fcn	Linear	Auto	Auto	1000

3 Under **Run Simulation**, select `Log data for histogram`, and then click **Run Simulation**. Verify that `custom_test` file is selected as the test file.

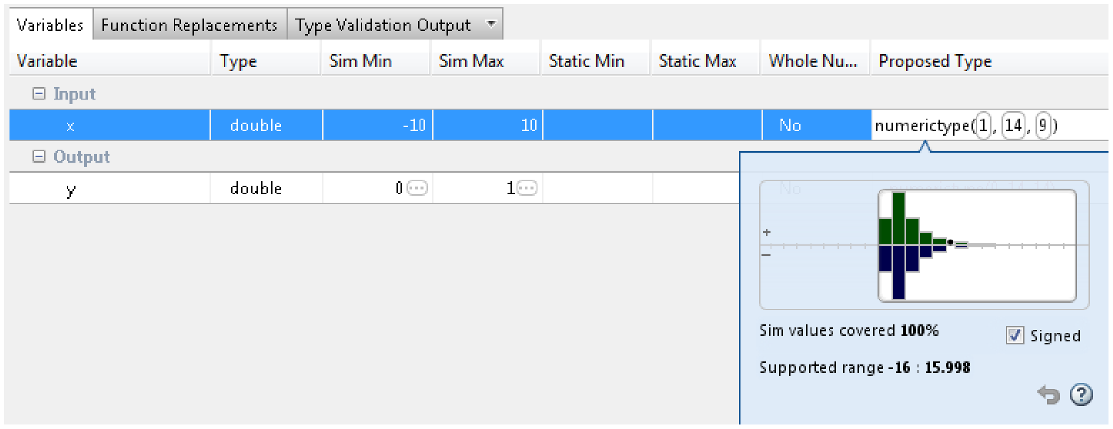
The simulation runs and the tool displays simulation minimum and maximum ranges on the **Variables** tab. HDL Coder plots the simulation results in the MATLAB Editor.


Figures - Testbench:float (14:03:26)
File
Edit
View
Insert
Tools
Debug
Desktop
Window
Help
Testbench:float (14:03:26)
1
0.8
0.6
0.4
0.2
0
-10
-5
0
5
10

Validate Fixed-Point Types

1 In the **Proposed Type** column, verify that the fixed-point types proposed by software cover the full simulation range. To view logged histogram data for a variable, click its **Proposed Type** field.

The histogram provides range information and the percentage of simulation range that the proposed data type covers.



Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
x	double	-10	10			No	numerictype(1, 14, 9)
Output							
y	double	0	1				

Sim values covered **100%**

☑ Signed

Supported range **-16 : 15.998**

2 To validate the build by using the proposed types, click **Validate Types**.

The software validates the proposed types and generates a fixed-point code, `call_custom_fcn_fixpt`.

3 To view the generated fixed-point code, click the `call_custom_fcn_fixpt` link.

The generated fixed-point function, `call_custom_fcn_fixpt.m`, calls this approximation instead of calling `custom_fcn`.

```
function y = call_custom_fcn_fixpt(x)
  fm = get_fimath();

  y = fi(replacement_custom_fcn(x), 0, 14, 14, fm);
end

function fm = get_fimath()
 fm = fimath('RoundingMethod', 'Floor',...
      'OverflowAction', 'Wrap',...
      'ProductMode','FullPrecision',...
```

```
            'MaxProductWordLength', 128,...
            'SumMode','FullPrecision',...
            'MaxSumWordLength', 128);
    end
```

From the Command Line

Prerequisites

To complete this example, you must install the following products:

- MATLAB
- Fixed-Point Designer
- C compiler (for most platforms, a default C compiler is supplied with MATLAB) For a list of supported compilers, see `http://www.mathworks.com/support/compilers/current_release/`

  You can use `mex -setup` to change the default compiler. See “Change Default Compiler” (MATLAB).

Create a MATLAB function, `custom_fcn.m`. This is the function that you want to replace.

```
function y = custom_fcn(x)
  y = 1./(1+exp(-x));
end
```

Create a wrapper function that calls `custom_fcn.m`.

```
function y = call_custom_fcn(x)
  y = custom_fcn(x);
end
```

Create a test file, `custom_test.m`, that uses `call_custom_fcn.m`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
   y(itr) = call_custom_fcn( x(itr) );
end
```

```
plot( x, y );
```

Create a function replacement configuration object to approximate `custom_fcn`. Specify the function handle of the custom function and set the number of points to use in the lookup table to 50.

```
q = coder.approximation('Function','custom_fcn',...
                        'CandidateFunction',@custom_fcn, 'NumberOfPoints',50);
```

Create a `coder.FixptConfig` object, `fixptcfg`. Specify the test file name and enable numerics testing. Associate the function replacement configuration object with the fixed-point configuration object.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'custom_test';
fixptcfg.TestNumerics = true;
fixptcfg.addApproximation(q);
```

Generate fixed-point MATLAB code.

```
codegen -float2fixed fixptcfg call_custom_fcn
```

`codegen` generates fixed-point MATLAB code in `call_custom_fcn_fixpt.m`.

To view the generated fixed-point code, click the link to `call_custom_fcn_fixpt`.

The generated code contains a lookup table approximation, `replacement_custom_fcn`, for the `custom_fcn` function. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. The lookup table uses 50 points as specified. By default, it uses linear interpolation and the minimum and maximum values detected by running the test file.

The generated fixed-point function, `call_custom_fcn_fixpt`, calls this approximation instead of calling `custom_fcn`.

```
function y = call_custom_fcn_fixpt(x)
  fm = get_fimath();

  y = fi(replacement_custom_fcn(x), 0, 14, 14, fm);
end
```

You can now test the generated fixed-point code and compare the results against the original MATLAB function. If the behavior of the generated fixed-point code does not match the behavior of the original code closely enough, modify the interpolation method or number of points used in the lookup table and then regenerate code.

See Also

`coder.approximation`

Related Examples

- "Replace the exp Function with a Lookup Table" on page 4-85

More About

- "Replacing Functions Using Lookup Table Approximations" on page 4-76

Replace the exp Function with a Lookup Table

With HDL Coder, you can handle functions that are not supported for fixed point and replace your own functions. To replace a custom function with a Lookup Table, use the HDL Coder App, or the `fiaccel codegen` function.

In this section...
"From the UI" on page 4-85
"From the Command Line" on page 4-90

From the UI

This example shows how to replace a custom function with a Lookup Table using the **HDL Coder** app.

Create Algorithm and Test Files

In a local, writable folder:

1 Create a MATLAB function, `custom_fcn`, which is the function that you want to replace.

```
function y = custom_fcn(x)
  y = 1./(1+exp(-x));
end
```

2 Create a wrapper function that calls `custom_fcn`.

```
function y = call_custom_fcn(x)
  y = custom_fcn(x);
end
```

3 Create a test file, `custom_test`, which uses `call_custom_fcn`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
   y(itr) = call_custom_fcn( x(itr) );
end
plot( x, y );
```

Create and Set up a HDL Coder Project

1. Navigate to the work folder that contains the file for this example.
2. To open the **HDL Coder** app, in the MATLAB command prompt, enter `hdlcoder`. Set **Name** to `custom_project.prj` and click **OK**. The project opens in the MATLAB workspace.
3. In the project window, on the **MATLAB Function** tab, click the **Add MATLAB function** link. Browse to the file `call_custom_fcn.m`, and then click **OK** to add the file to the project.

Define Input Types

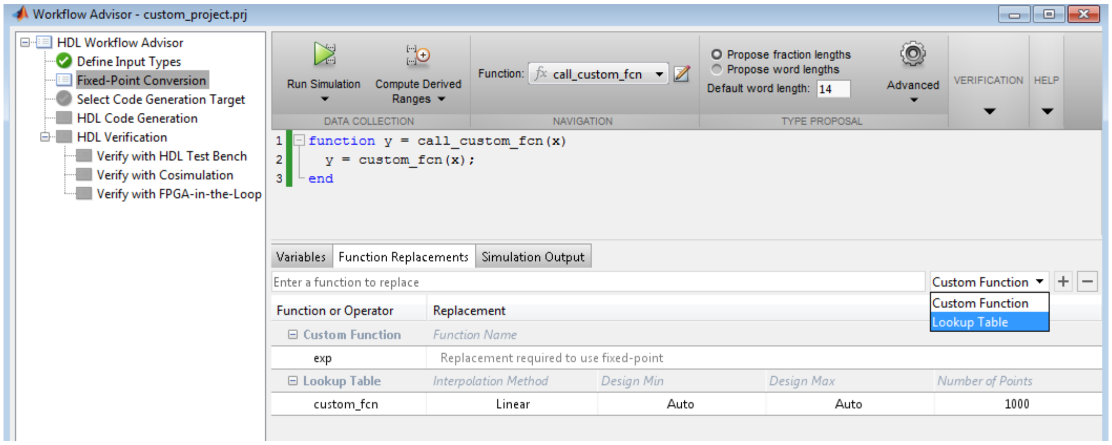
1. To define input types for `call_custom_fcn.m`, on the **MATLAB Function** tab, click **Autodefine types**.
2. Add `custom_test` as a test file, and then click **Run**.

   From the test file, HDL Coder determines that `x` is a scalar double.
3. Click **Use These Types**.

Replace custom_fcn with Lookup Table

1. To open the HDL Workflow Advisor, click **Workflow Advisor**, and in the Workflow Advisor window, click **Fixed-Point Conversion**.
2. To replace `custom_fcn` with a Lookup Table, on the **Function Replacements** tab, enter `custom_fcn`, select `Lookup Table`, and then click `+`.

   By default, the lookup table uses linear interpolation, 1000 points, and design minimum and maximum values that the app detects by running a simulation or computing derived ranges.



Function: call_custom_fcn

Default word length: 14

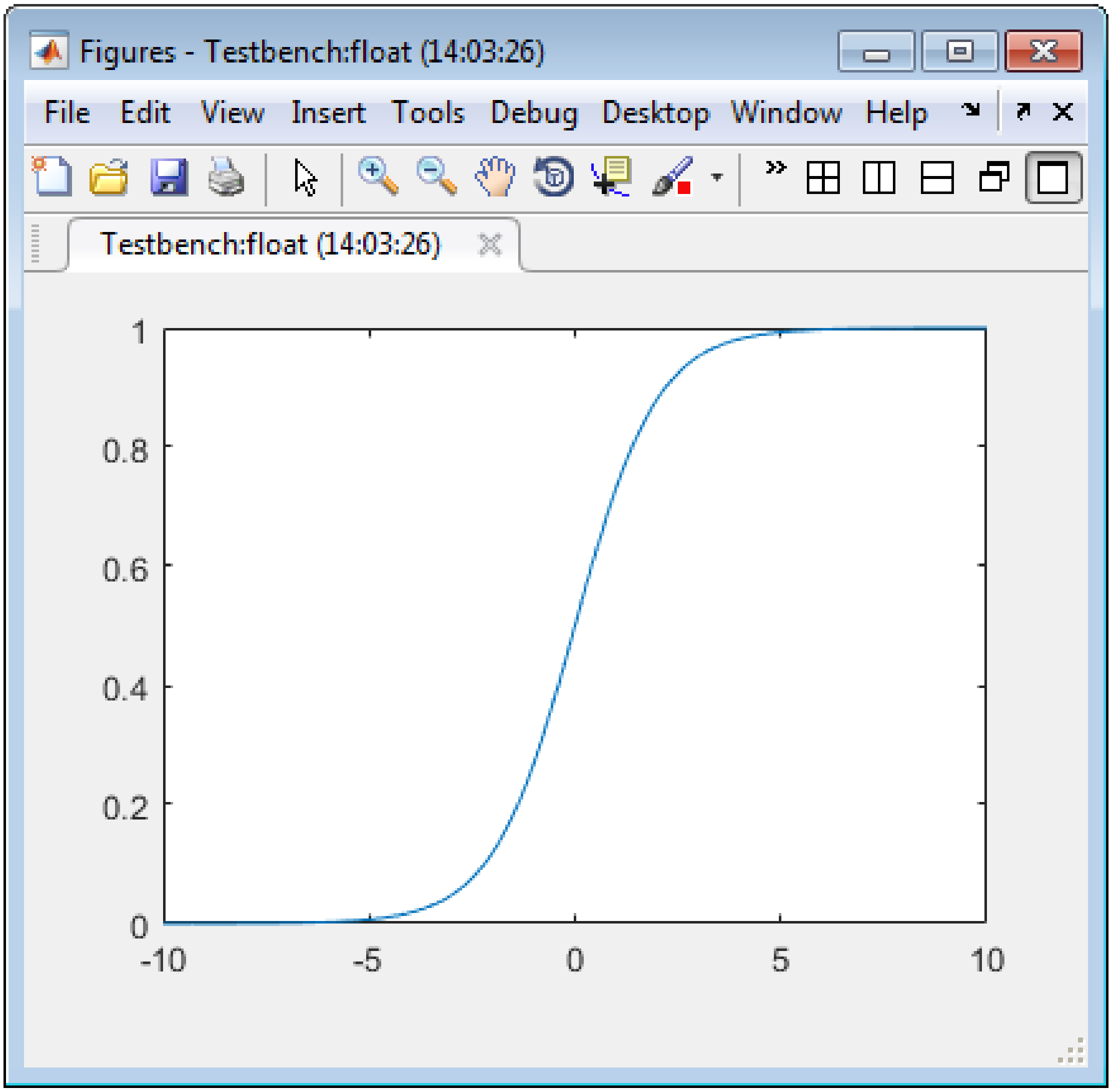
```
function y = call_custom_fcn(x)
  y = custom_fcn(x);
end
```

Enter a function to replace

Function or Operator	Replacement			
Custom Function	Function Name			
exp	Replacement required to use fixed-point			
Lookup Table	Interpolation Method	Design Min	Design Max	Number of Points
custom_fcn	Linear	Auto	Auto	1000

3 Under **Run Simulation**, select `Log data for histogram`, and then click **Run Simulation**. Verify that `custom_test` file is selected as the test file.

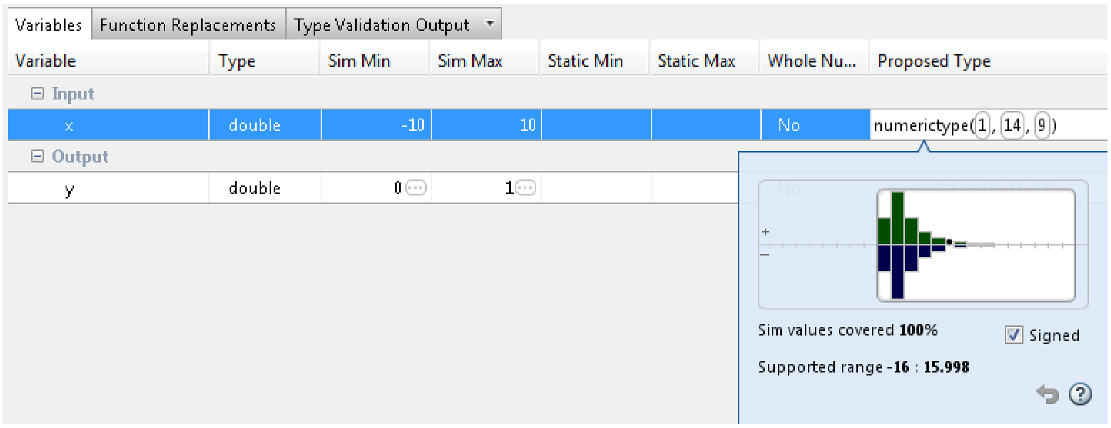
The simulation runs and the tool displays simulation minimum and maximum ranges on the **Variables** tab. HDL Coder plots the simulation results in the MATLAB Editor.


Figures - Testbench:float (14:03:26)
File Edit View Insert Tools Debug Desktop Window Help
Testbench:float (14:03:26)
1
0.8
0.6
0.4
0.2
0
-10
-5
0
5
10

Validate Fixed-Point Types

1 In the **Proposed Type** column, verify that the fixed-point types proposed by software cover the full simulation range. To view logged histogram data for a variable, click its **Proposed Type** field.

The histogram provides range information and the percentage of simulation range that the proposed data type covers.



Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
x	double	-10	10			No	numerictype(1, 14, 9)
Output							
y	double	0	1				

Sim values covered **100%**

✓ Signed

Supported range **-16 : 15.998**

2 To validate the build by using the proposed types, click **Validate Types**.

The software validates the proposed types and generates a fixed-point code, `call_custom_fcn_fixpt`.

3 To view the generated fixed-point code, click the `call_custom_fcn_fixpt` link.

The generated fixed-point function, `call_custom_fcn_fixpt.m`, calls this approximation instead of calling `custom_fcn`.

```
function y = call_custom_fcn_fixpt(x)
  fm = get_fimath();

  y = fi(replacement_custom_fcn(x), 0, 14, 14, fm);
end

function fm = get_fimath()
 fm = fimath('RoundingMethod', 'Floor',...
      'OverflowAction', 'Wrap',...
      'ProductMode','FullPrecision',...
```

```
            'MaxProductWordLength', 128,...
            'SumMode','FullPrecision',...
            'MaxSumWordLength', 128);
    end
```

From the Command Line

This example shows how to replace the `exp` function with a lookup table approximation in the generated fixed-point code using the function.

Prerequisites

To complete this example, you must install the following products:

- MATLAB
- Fixed-Point Designer
- C compiler (for most platforms, a default C compiler is supplied with MATLAB). For a list of supported compilers, see `http://www.mathworks.com/support/compilers/current_release/`.

  You can use `mex -setup` to change the default compiler. See “Change Default Compiler” (MATLAB).

Create Algorithm and Test Files

1. Create a MATLAB function, `my_fcn.m`, that calls the `exp` function.

   ```
   function y = my_fcn(x)
     y = exp(x);
   end
   ```

2. Create a test file, `my_fcn_test.m`, that uses `my_fcn.m`.

   ```
   close all

   x = linspace(-10,10,1e3);
   for itr = 1e3:-1:1
      y(itr) = my_fcn( x(itr) );
   end
   plot( x, y );
   ```

Configure Approximation

Create a function replacement configuration object to approximate the `exp` function, using the default settings of linear interpolation and 1000 points in the lookup table.

```
q = coder.approximation('exp');
```

Set Up Configuration Object

Create a `coder.FixptConfig` object, `fixptcfg`. Specify the test file name and enable numerics testing. Associate the function replacement configuration object with the fixed-point configuration object.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'my_fcn_test';
fixptcfg.TestNumerics = true;
fixptcfg.DefaultWordLength = 16;
fixptcfg.addApproximation(q);
```

Convert to Fixed Point

Generate fixed-point MATLAB code.

```
codegen -float2fixed fixptcfg my_fcn
```

View Generated Fixed-Point Code

To view the generated fixed-point code, click the link to `my_fcn_fixpt`.

The generated code contains a lookup table approximation, `replacement_exp`, for the `exp` function. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. By default, the lookup table uses linear interpolation, 1000 points, and the minimum and maximum values detected by running the test file.

The generated fixed-point function, `my_fcn_fixpt`, calls this approximation instead of calling `exp`.

```
function y = my_fcn_fixpt(x)
  fm = get_fimath();

  y = fi(replacement_exp(x), 0, 16, 1, fm);
end
```

You can now test the generated fixed-point code and compare the results against the original MATLAB function. If the behavior of the generated fixed-point code does not match the behavior of the original code closely enough, modify the interpolation method or number of points used in the lookup table and then regenerate code.

See Also

`coder.approximation`

Related Examples

- “Replace a Custom Function with a Lookup Table” on page 4-77

More About

- “Replacing Functions Using Lookup Table Approximations” on page 4-76

Data Type Issues in Generated Code

Within the fixed-point conversion HTML report, you have the option to highlight MATLAB code that results in double, single, or expensive fixed-point operations. Consider enabling these checks when trying to achieve a strict single, or fixed-point design.

These checks are disabled by default.

Enable the Highlight Option in a Project

1 Open the **Settings** menu.
2 Under **Plotting and Reporting**, set **Highlight potential data type issues** to `Yes`.

Enable the Highlight Option at the Command Line

1 Create a fixed-point code configuration object:

```
cfg = coder.config('fixpt');
```

2 Set the `HighlightPotentialDataTypeIssues` property of the configuration object to `true`.

```
cfg.HighlightPotentialDataTypeIssues = true;
```

Stowaway Doubles

When trying to achieve a strict-single or fixed-point design, manual inspection of code can be time-consuming and error prone. This check highlights all expressions that result in a double operation.

For a strict-single precision design, specify a standard math library that supports single-precision implementations. To change the library for a project, during the Generate Code step, in the project settings dialog box, on the **Custom Code** tab, set the **Standard math library** to `C99 (ISO)`.

Stowaway Singles

This check highlights all expressions that result in a single operation.

Expensive Fixed-Point Operations

The expensive fixed-point operations check identifies optimization opportunities for fixed-point code. It highlights expressions in the MATLAB code that require cumbersome multiplication or division, expensive rounding, expensive comparison, or multiword operations. For more information on optimizing generated fixed-point code, see “Tips for Making Generated Code More Efficient” (Fixed-Point Designer).

Cumbersome Operations

Cumbersome operations most often occur due to insufficient range of output. Avoid inputs to a multiply or divide operation that has word lengths larger than the base integer type of your processor. Operations with larger word lengths can be handled in software, but this approach requires much more code and is much slower.

Expensive Rounding

Traditional handwritten code, especially for control applications, almost always uses "no effort" rounding. For example, for unsigned integers and two's complement signed integers, shifting right and dropping the bits is equivalent to rounding to floor. To get results comparable to, or better than, what you expect from traditional handwritten code, use the `floor` rounding method. This check identifies expensive rounding operations in multiplication and division.

Expensive Comparison Operations

Comparison operations generate extra code when a casting operation is required to do the comparison. For example, when comparing an unsigned integer to a signed integer, one of the inputs must first be cast to the signedness of the other before the comparison operation can be performed. Consider optimizing the data types of the input arguments so that a cast is not required in the generated code.

Multiword Operations

Multiword operations can be inefficient on hardware. When an operation has an input or output data type larger than the largest word size of your processor, the generated code contains multiword operations. You can avoid multiword operations in the generated code by specifying local `fimath` properties for variables. You can also manually specify input and output word lengths of operations that generate multiword code.

Code Generation

Create and Set Up Your Project

In this section...
"Create a New Project" on page 5-2
"Open an Existing Project" on page 5-4
"Add Files to the Project" on page 5-4

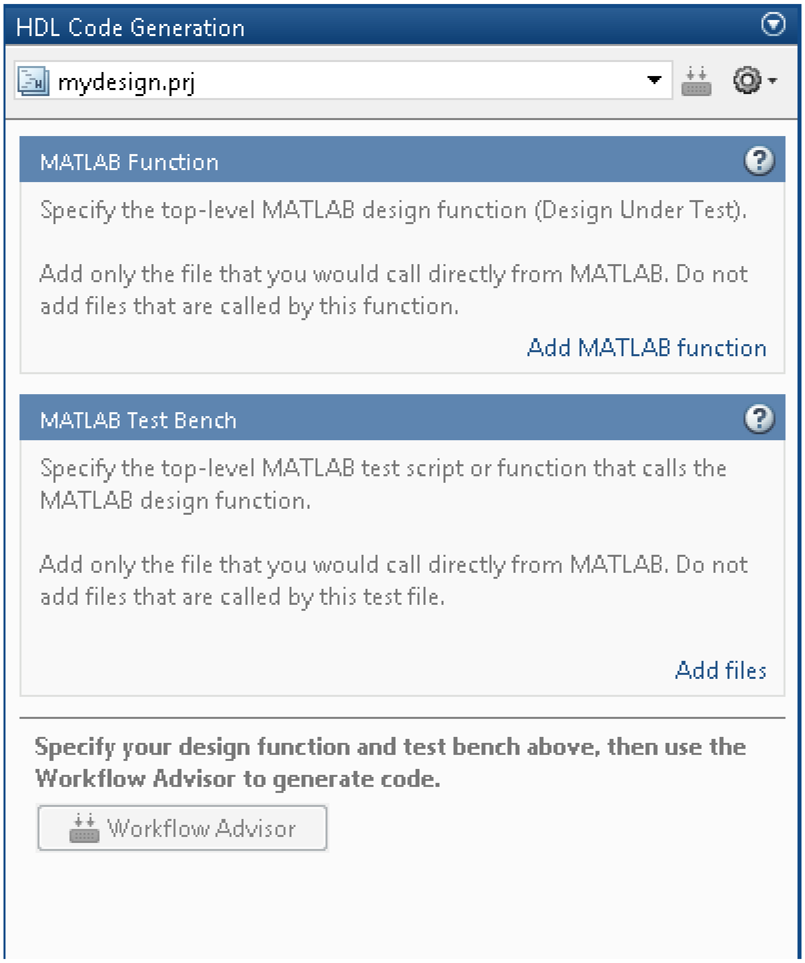
Create a New Project

1 At the MATLAB command line, enter:

```
hdlcoder
```

2 Enter a project name in the project dialog box and click **OK**.

HDL Coder creates the project in the local working folder, and, by default, opens the project in the right side of the MATLAB workspace.



Specify the top-level MATLAB design function (Design Under Test).

Add only the file that you would call directly from MATLAB. Do not add files that are called by this function.

Specify the top-level MATLAB test script or function that calls the MATLAB design function.

Add only the file that you would call directly from MATLAB. Do not add files that are called by this test file.

Specify your design function and test bench above, then use the Workflow Advisor to generate code.

Alternatively, you can create a new HDL Coder project from the apps gallery:

1 On the **Apps** tab, on the far right of the **Apps** section, click the arrow ▾.
2 Under **Code Generation**, click **HDL Coder**.
3 Enter a project name in the project dialog box and click **OK**.

Open an Existing Project

At the MATLAB command line, enter:

```
open project_name
```
where `project_name` specifies the full path to the project file.

Alternatively, navigate to the folder that contains your project and double-click the `.prj` file.

Add Files to the Project

Add the MATLAB Function (Design Under Test)

First, you must add the MATLAB file from which you want to generate code to the project. Add only the top-level function that you call from MATLAB (the Design Under Test). Do not add files that are called by this file. Do not add files that have spaces in their names. The path must not contain spaces, as spaces can lead to code generation failures in certain operating system configurations.

To add a file, do one of the following:

- In the project pane, under **MATLAB Function** , click the **Add MATLAB function** link and browse to the file.
- Drag a file from the current folder and drop it in the project pane under **MATLAB Function**.

If the functions that you added have inputs, and you do not specify a test bench, you must define these inputs. See “Specify Properties of Entry-Point Function Inputs” on page 5-6.

Add a MATLAB Test Bench

You must add a MATLAB test bench unless your design does not need fixed-point conversion and you do not want to generate an RTL test bench. If you do not add a test bench, you must define the inputs to your top-level MATLAB function. For more information, see “Specify Properties of Entry-Point Function Inputs” on page 5-6.

To add a test bench, do one of the following:

- In the project panel, under **MATLAB Test Bench**, click the **Add MATLAB test bench** link and browse to the file.

- Drag a file from the current folder and drop it in the project pane under **MATLAB Test Bench**.

Specify Properties of Entry-Point Function Inputs

When to Specify Input Properties

If you supply a test bench for your MATLAB algorithm, you do not need to specify the primary function inputs manually. The HDL Coder software uses the test bench to infer the data types.

Why You Must Specify Input Properties

HDL Coder must determine the properties of all variables in the MATLAB files at compile time. To infer variable properties in MATLAB files, HDL Coder must be able to identify the properties of the inputs to the *primary* function, also known as the *top-level* or *entry-point* function. Therefore, if your primary function has inputs, you must specify the properties of these inputs, to HDL Coder. If your primary function has no input parameters, HDL Coder can compile your MATLAB file without modification. You do not need to specify properties of inputs to local functions or external functions called by the primary function.

If you use the tilde (~) character to specify unused function inputs in an HDL Coder project, and you want a different type to appear in the generated code, specify the type. Otherwise, the inputs default to real, scalar doubles.

Properties to Specify

If your primary function has inputs, you must specify the following properties for each input.

For	Specify properties				
	Class	**Size**	**Complexity**	**numerictype**	**fimath**

For	Specify properties				
Fixed-point inputs	✓	✓	✓	✓	✓
Other inputs	✓	✓	✓		

The following data types are not supported for primary function inputs, although you can use them within the primary function:

- structure
- matrix

Variable-size data is not supported in the test bench or the primary function.

Default Property Values

HDL Coder assigns the following default values for properties of primary function inputs.

Property	Default
class	`double`
size	`scalar`
complexity	`real`
`numerictype`	No default
`fimath`	`hdlfimath`

Supported Classes

The following table presents the class names supported by HDL Coder.

Class Name	Description
`logical`	Logical array of true and false values
`char`	Character array
`int8`	8-bit signed integer array
`uint8`	8-bit unsigned integer array
`int16`	16-bit signed integer array

Class Name	Description
`uint16`	16-bit unsigned integer array
`int32`	32-bit signed integer array
`uint32`	32-bit unsigned integer array
`single`	Single-precision floating-point or fixed-point number array
`double`	Double-precision floating-point or fixed-point number array
`embedded.fi`	Fixed-point number array

Rules for Specifying Properties of Primary Inputs

When specifying the properties of primary inputs, follow these rules:

- You must specify the class of all primary inputs. If you do not specify the size or complexity of primary inputs, they default to real scalars.
- For each primary function input whose class is fixed point (`fi`), you must specify the input `numerictype` and `fimath` properties.

Methods for Defining Properties of Primary Inputs

Method	Advantages	Disadvantages
Note: If you define input properties programmatically in the MATLAB file, you cannot use this method	• Easy to use • Does not alter original MATLAB code • Designed for prototyping a function that has a few primary inputs	• Must be specified at the command line every time you invoke (unless you use a script) • Not efficient for specifying memory-intensive inputs such as large structures and arrays
"Define Input Properties Programmatically in	• Integrated with MATLAB code; no need to redefine properties each time you invoke HDL Coder	• Uses complex syntax • HDL Coder project files do not currently recognize properties defined programmatically. If you

Method	Advantages	Disadvantages
the MATLAB File" (MATLAB Coder)	• Provides documentation of property specifications in the MATLAB code • Efficient for specifying memory-intensive inputs such as large structures	are using a project, you must reenter the input types in the project.

Basic HDL Code Generation with the Workflow Advisor

This example shows how to work with MATLAB® HDL Coder™ projects to generate HDL from MATLAB designs.

Introduction

This example helps you familiarize yourself with the following aspects of HDL code generation:

1. Generating HDL code from MATLAB design.
2. Generating a HDL test bench from a MATLAB test bench.
3. Verifying the generated HDL code using a HDL simulator.
4. Synthesizing the generated HDL code using a HDL synthesis tool.

MATLAB Design

The MATLAB code used in this example implements a simple symmetric FIR filter. This example also shows a MATLAB test bench that exercises the filter.

```
design_name = 'mlhdlc_sfir';
testbench_name = 'mlhdlc_sfir_tb';
```

1. MATLAB Design: mlhdlc_sfir
2. MATLAB testbench: mlhdlc_sfir_tb

Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_sfir'];

% Create a temporary folder and copy the MATLAB files.
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name,'.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name,'.m*']), mlhdlc_temp_dir);
```

Simulate the Design

Simulate the design with the test bench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_sfir_tb
```

Create a New HDL Coder Project

To create a new project, enter the following command:

```
coder -hdlcoder -new mlhdlc_sfir
```

Next, add the file 'mlhdlc_sfir.m' to the project as the MATLAB Function and 'mlhdlc_sfir_tb.m' as the MATLAB Test Bench.

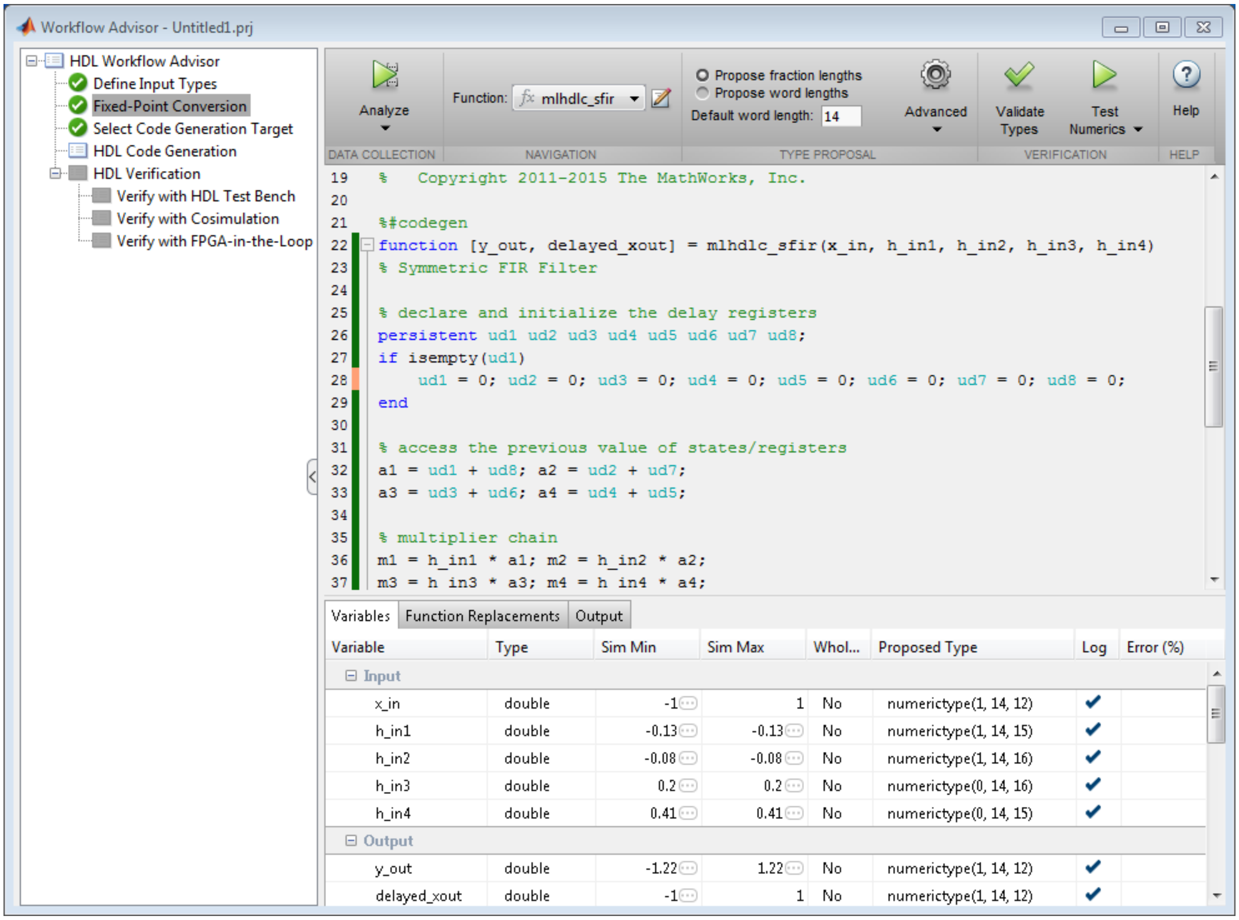
You can refer to Getting Started with MATLAB to HDL Workflow tutorial for a more complete introduction to creating and populating HDL Coder projects.

Step 1: Generate Fixed-Point MATLAB Code

Right-click the 'Float-to-Fixed Workflow' step and choose the option 'Run this task' to run all the steps to generate fixed-point MATLAB code.

Examine the generated fixed-point MATLAB code by clicking the links in the log window to open the MATLAB code in the editor.

For more details on fixed-point conversion, refer to the Floating-Point to Fixed-Point Conversion tutorial.



Function: mlhdlc_sfir

[✓ Propose fraction lengths] Propose word lengths

Default word length: 14

```
%   Copyright 2011-2015 The MathWorks, Inc.

%#codegen
function [y_out, delayed_xout] = mlhdlc_sfir(x_in, h_in1, h_in2, h_in3, h_in4)
% Symmetric FIR Filter

% declare and initialize the delay registers
persistent ud1 ud2 ud3 ud4 ud5 ud6 ud7 ud8;
if isempty(ud1)
    ud1 = 0; ud2 = 0; ud3 = 0; ud4 = 0; ud5 = 0; ud6 = 0; ud7 = 0; ud8 = 0;
end

% access the previous value of states/registers
a1 = ud1 + ud8; a2 = ud2 + ud7;
a3 = ud3 + ud6; a4 = ud4 + ud5;

% multiplier chain
m1 = h_in1 * a1; m2 = h_in2 * a2;
m3 = h_in3 * a3; m4 = h_in4 * a4;
```

Variable	Type	Sim Min	Sim Max	Whol...	Proposed Type	Log	Error (%)
Input							
x_in	double	-1	1	No	numerictype(1, 14, 12)	✓	
h_in1	double	-0.13	-0.13	No	numerictype(1, 14, 15)	✓	
h_in2	double	-0.08	-0.08	No	numerictype(1, 14, 16)	✓	
h_in3	double	0.2	0.2	No	numerictype(0, 14, 16)	✓	
h_in4	double	0.41	0.41	No	numerictype(0, 14, 15)	✓	
Output							
y_out	double	-1.22	1.22	No	numerictype(1, 14, 12)	✓	
delayed_xout	double	-1	1	No	numerictype(1, 14, 12)	✓	

Step 2: Generate HDL Code

This step generates Verilog code from the generated fixed-point MATLAB design, and a Verilog test bench from the MATLAB test bench wrapper.

To set code generation options and generate HDL code:

1 Click the 'Code Generation' step to view the HDL code generation options panel.

2 In the Target tab, choose 'Verilog' as the 'Language' option.

3 Select the 'Generate HDL' and 'Generate HDL test bench' options.

4. In the 'Optimizations' tab, choose '1' as the Input and Output pipeline length, and enable the 'Distribute pipeline registers' option.
5. In the 'Coding style' tab, choose 'Include MATLAB source code as comments' and 'Generate report' to generate a code generation report with comments and traceability links.
6. Click the 'Run' button to generate both the Verilog design and testbench with reports.

Examine the log window and click the links to explore the generated code and the reports.

Step 3: Simulate the Generated Code

In the 'HDL Verification' step, select 'Verify with HDL Test Bench' substep and choose the 'Multi-file test bench' option in 'Test Bench Options' sub-tab. This option helps to generate HDL test bench code and test bench data (stimulus and response) in separate files.

HDL Coder automates the process of generating a HDL test bench and running the generated HDL test bench using the ModelSim® or ISIM™ simulator, and reports if the generated HDL simulation matches the numerics and latency with respect to the fixed-point MATLAB simulation.

Step 4: Synthesize the Generated Code

HDL Coder also creates a Xilinx® ISE™ or Altera® Quartus™ project with the selected options and runs the selected logic synthesis and place-and-route steps for the generated HDL code.

Examine the log window to view the results of synthesis steps.

Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_sfir'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

HDL Code Generation from System Objects

This example shows how to generate HDL code from MATLAB® code that contains System objects.

MATLAB Design

The MATLAB code used in this example implements a simple symmetric FIR filter and uses the dsp.Delay System object to model state. This example also shows a MATLAB test bench that exercises the filter.

```
design_name = 'mlhdlc_sysobj_ex';
testbench_name = 'mlhdlc_sysobj_ex_tb';
```

Let us take a look at the MATLAB design.

```
type(design_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB design: Symmetric FIR Filter
%
% Design pattern covered in this example:
%  Filter states modeled using DSP System object (dsp.Delay)
%  Filter coefficients passed in as parameters to the design
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%   Copyright 2011-2015 The MathWorks, Inc.

%#codegen
function [y_out, delayed_xout] = mlhdlc_sysobj_ex(x_in, h_in1, h_in2, h_in3, h_in4)
% Symmetric FIR Filter

persistent h1 h2 h3 h4 h5 h6 h7 h8;
if isempty(h1)
    h1 = dsp.Delay;
    h2 = dsp.Delay;
    h3 = dsp.Delay;
    h4 = dsp.Delay;
    h5 = dsp.Delay;
    h6 = dsp.Delay;
    h7 = dsp.Delay;
    h8 = dsp.Delay;
```

```
end

h1p = step(h1, x_in);
h2p = step(h2, h1p);
h3p = step(h3, h2p);
h4p = step(h4, h3p);
h5p = step(h5, h4p);
h6p = step(h6, h5p);
h7p = step(h7, h6p);
h8p = step(h8, h7p);

a1 = h1p + h8p;
a2 = h2p + h7p;
a3 = h3p + h6p;
a4 = h4p + h5p;

m1 = h_in1 * a1;
m2 = h_in2 * a2;
m3 = h_in3 * a3;
m4 = h_in4 * a4;

a5 = m1 + m2;
a6 = m3 + m4;

% filtered output
y_out = a5 + a6;
% delayout input signal
delayed_xout = h8p;

end
```

```
type(testbench_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB test bench for the FIR filter
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%   Copyright 2011-2015 The MathWorks, Inc.

clear mlhdlc_sysobj_ex;

x_in = cos(2.*pi.*(0:0.001:2).*(1+(0:0.001:2).*75)).';
```

```
h1 = -0.1339;
h2 = -0.0838;
h3 = 0.2026;
h4 = 0.4064;

len = length(x_in);
y_out_sysobj = zeros(1,len);
x_out_sysobj = zeros(1,len);
a = 10;

for ii=1:len
    data = x_in(ii);
    % call to the design 'sfir' that is targeted for hardware
    [y_out_sysobj(ii), x_out_sysobj(ii)] = mlhdlc_sysobj_ex(data, h1, h2, h3, h4);
end


figure('Name', [mfilename, '_plot']);
subplot(2,1,1);
plot(1:len,x_in); title('Input signal with noise');
subplot(2,1,2);
plot(1:len,y_out_sysobj); title('Filtered output signal');
```

Create a New Folder and Copy Relevant Files

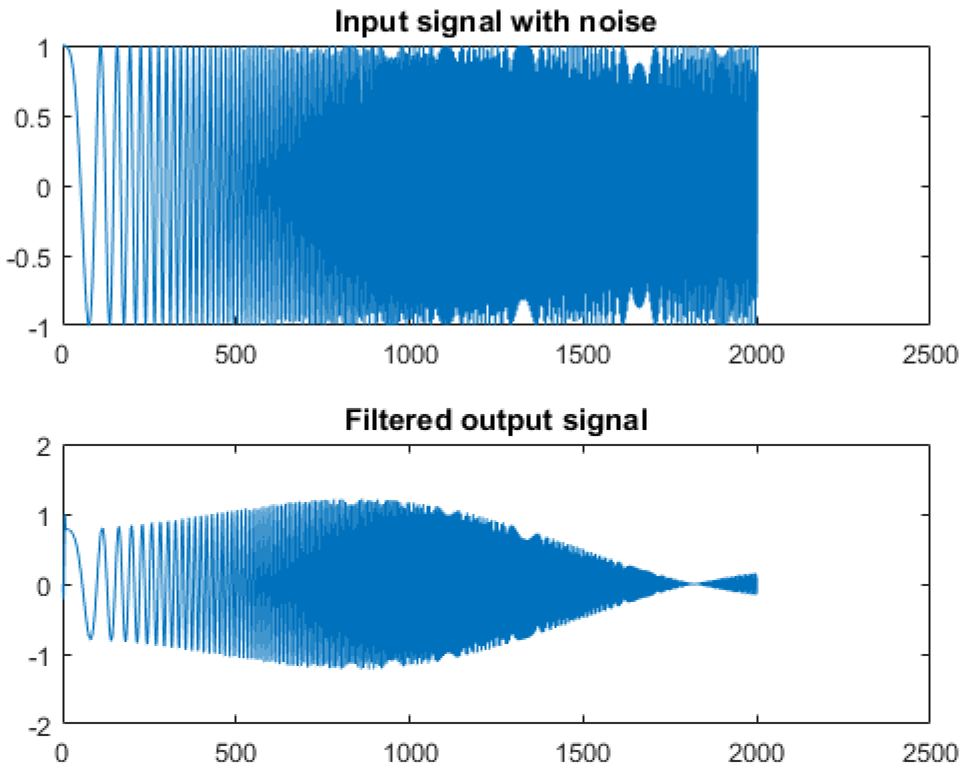
Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_sysobj_intro'];

% Create a temporary folder and copy the MATLAB files.
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name,'.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name,'.m*']), mlhdlc_temp_dir);
```

Simulate the Design

Simulate the design with the test bench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_sysobj_ex_tb
```



Create a New HDL Coder™ Project

To create a new project, enter the following command:

```
coder -hdlcoder -new mlhdlc_sysobj_prj
```

Next, add the file 'mlhdlc_sysobj_ex.m' to the project as the MATLAB Function and 'mlhdlc_sysobj_ex_tb.m' as the MATLAB Test Bench.

You can refer to the Getting Started with MATLAB to HDL Workflow tutorial for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Run Fixed-Point Conversion and HDL Code Generation

Launch the Workflow Advisor. In the Workflow Advisor, right-click the 'Code Generation' step. Choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

Examine the generated HDL code by clicking the links in the log window.

Supported System objects

Refer to the documentation for a list of System objects supported for HDL code generation.

Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_sysobj_intro'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

Generate Instantiable Code for Functions

In this section...

You can use the **Generate instantiable code for functions** option to generate a VHDL `entity` or Verilog `module` for each function. The software generates code for each entity or module in a separate file.

How to Generate Instantiable Code for Functions

To enable instantiable code generation for functions in the UI:

1. In the HDL Workflow Advisor, select the **HDL Code Generation** task.
2. In the **Advanced** tab, select **Generate instantiable code for functions**.

To enable instantiable code generation for functions programmatically, in your coder.HdlConfig object, set the `InstantiateFunctions` property to true. For example, to create a `coder.HdlConfig` object and enable instantiable code generation for functions:

```
hdlcfg = coder.config('hdl');
hdlcfg.InstantiateFunctions = true;
```

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.

If you enable `InstantiateFunctions` , `UseMatrixTypesInHDL` has no effect.

Integrate Custom HDL Code Into MATLAB Design

hdl.BlackBox provides a way to include custom HDL code, such as legacy or handwritten HDL code, in a MATLAB design intended for HDL code generation.

When you create a user-defined System object that inherits from `hdl.BlackBox`, you specify a port interface and simulation behavior that matches your custom HDL code.

HDL Coder simulates the design in MATLAB using the behavior you define in the System object. During code generation, instead of generating code for the simulation behavior, the coder instantiates a module with the port interface you specify in the System object.

To use the generated HDL code in a larger system, you include the custom HDL source files with the rest of the generated code.

In this section...

Define the `hdl.BlackBox` System object

1 Create a user-defined System object that inherits from hdl.BlackBox.
2 Configure the black box interface to match the port interface for your custom HDL code by setting `hdl.BlackBox` properties in the System object.
3 Define the `step` method such that its simulation behavior matches the custom HDL code.

   Alternatively, the System object you define can inherit from both `hdl.BlackBox` and the `matlab.system.mixin.Nondirect` class, and you can define `output` and `update` methods to match the custom HDL code simulation behavior.

Example Code

For example, the following code defines a System object, `CounterBbox`, that inherits from `hdl.BlackBox` and represents custom HDL code for a counter that increments until it reaches a threshold. The `CounterBbox` `reset` and `step` methods model the custom HDL code behavior.

```
classdef CounterBbox < hdl.BlackBox % derive from hdl.BlackBox class
    %Counter: Count up to a threshold.
    %
    % This is an example of a discrete-time System object with state
    % variables.
    %
    properties (Nontunable)
        Threshold = 1
    end

    properties (DiscreteState)
        % Define discrete-time states.
        Count
    end

    methods
        function obj = CounterBbox(varargin)
            % Support name-value pair arguments
            setProperties(obj,nargin,varargin{:});
            obj.NumInputs = 1;   % define number of inputs
            obj.NumOutputs = 1;  % define number of inputs
        end
    end

    methods (Access=protected)
        % Define simulation behavior.
        % For code generation, the coder uses your custom HDL code instead.
        function resetImpl(obj)
            % Specify initial values for DiscreteState properties
            obj.Count = 0;
        end

        function myout = stepImpl(obj, myin)
            % Implement algorithm. Calculate y as a function of
            % input u and state.
            if (myin > obj.Threshold)
                obj.Count = obj.Count + 1;
            end
            myout = obj.Count;
        end
    end
```

```
end
```

Use System object In MATLAB Design Function

After you define your System object, use it in the MATLAB design function by creating an instance and calling its `step` method.

To generate code, you also need to create a test bench function that exercises the top-level design function.

Example Code

The following example code shows a top-level design function that creates an instance of the `CounterBbox` and calls its `step` method.

```
function [y1, y2] = topLevelDesign(u)

persistent mybboxObj myramObj
if isempty(mybboxObj)
    mybboxObj = CounterBbox; % instantiate the black box
    myramObj = hdl.RAM('RAMType', 'Dual port');
end

y1 = step(mybboxObj, u); % call the system object step method
[~, y2] = step(myramObj, uint8(10), uint8(0), true, uint8(20));
```

The following example code shows a test bench function for the `topLevelDesign` function.

```
clear topLevelDesign
y1 = zeros(1,200);
y2 = zeros(1,200);
for ii=1:200
    [y1(ii), y2(ii)] = topLevelDesign(ii);
end
plot([1:200], y2)
```

Generate HDL Code

Generate HDL code using the design function and test bench code.

When you use the generated HDL code, include your custom HDL code with the generated HDL files.

Example Code

In the following generated VHDL code for the `CounterBbox` example, you can see that the `CounterBbox` instance in the MATLAB code maps to an HDL component definition and instantiation, but HDL code is not generated for the `step` method.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY foo IS
  PORT( clk               :   IN    std_logic;
        reset             :   IN    std_logic;
        clk_enable        :   IN    std_logic;
        u                 :   IN    std_logic_vector(7 DOWNTO 0);  -- uint8
        ce_out            :   OUT   std_logic;
        y1                :   OUT   real;  -- double
        y2                :   OUT   std_logic_vector(7 DOWNTO 0)  -- uint8
        );
END foo;


ARCHITECTURE rtl OF foo IS

  -- Component Declarations
  COMPONENT CounterBbox
    PORT( clk             :   IN    std_logic;
          clk_enable      :   IN    std_logic;
          reset           :   IN    std_logic;
          myin            :   IN    std_logic_vector(7 DOWNTO 0);  -- uint8
          myout           :   OUT   real  -- double
          );
  END COMPONENT;

  COMPONENT DualPortRAM_Inst0
    PORT( clk             :   IN    std_logic;
          enb             :   IN    std_logic;
          wr_din          :   IN    std_logic_vector(7 DOWNTO 0);  -- uint8
          wr_addr         :   IN    std_logic_vector(7 DOWNTO 0);  -- uint8
          wr_en           :   IN    std_logic;
          rd_addr         :   IN    std_logic_vector(7 DOWNTO 0);  -- uint8
          wr_dout         :   OUT   std_logic_vector(7 DOWNTO 0);  -- uint8
          rd_dout         :   OUT   std_logic_vector(7 DOWNTO 0)  -- uint8
          );
  END COMPONENT;

  -- Component Configuration Statements
  FOR ALL : CounterBbox
    USE ENTITY work.CounterBbox(rtl);

  FOR ALL : DualPortRAM_Inst0
    USE ENTITY work.DualPortRAM_Inst0(rtl);

  -- Signals
  SIGNAL enb              : std_logic;
  SIGNAL varargout_1      : real := 0.0;  -- double
  SIGNAL tmp              : unsigned(7 DOWNTO 0);  -- uint8
  SIGNAL tmp_1            : unsigned(7 DOWNTO 0);  -- uint8
  SIGNAL tmp_2            : std_logic;
  SIGNAL tmp_3            : unsigned(7 DOWNTO 0);  -- uint8
  SIGNAL varargout_1_1    : std_logic_vector(7 DOWNTO 0);  -- ufix8
  SIGNAL varargout_2      : std_logic_vector(7 DOWNTO 0);  -- ufix8

BEGIN
  u_CounterBbox : CounterBbox
    PORT MAP( clk => clk,
              clk_enable => enb,
              reset => reset,
              myin => u,  -- uint8
              myout => varargout_1  -- double
              );

  u_DualPortRAM_Inst0 : DualPortRAM_Inst0
    PORT MAP( clk => clk,
              enb => enb,
              wr_din => std_logic_vector(tmp),  -- uint8
              wr_addr => std_logic_vector(tmp_1),  -- uint8
              wr_en => tmp_2,
              rd_addr => std_logic_vector(tmp_3),  -- uint8
              wr_dout => varargout_1_1,  -- uint8
              rd_dout => varargout_2  -- uint8
              );

  enb <= clk_enable;

  y1 <= varargout_1;

  --y2 = u;
  tmp <= to_unsigned(2#00001010#, 8);

  tmp_1 <= to_unsigned(2#00000000#, 8);

  tmp_2 <= '1';

  tmp_3 <= to_unsigned(2#00010100#, 8);

  ce_out <= clk_enable;

  y2 <= varargout_2;

END rtl;
```

Limitations for `hdl.BlackBox`

You cannot use `hdl.BlackBox` to assign values to a VHDL `generic` or Verilog `parameter` in your custom HDL code.

See Also

hdl.BlackBox

Related Examples

- “Generate a Board-Independent IP Core from MATLAB” on page 5-50

Enable MATLAB Function Block Generation

In this section...

Requirements for MATLAB Function Block Generation

During HDL code generation, your MATLAB algorithm must go through the floating-point to fixed-point conversion process, even if it is already a fixed-point algorithm.

Enable MATLAB Function Block Generation

Using the GUI

To enable MATLAB Function block generation using the HDL Workflow Advisor:

1. In the HDL Workflow Advisor, on the left, click **Code Generation**.
2. In the **Advanced** tab, select the **Generate MATLAB Function Black Box** option.

Using the Command Line

To enable MATLAB Function block generation, at the command line, enter:

```
hdlcfg = coder.config('hdl');
hdlcfg.GenerateMLFcnBlock = true;
```

Restrictions for MATLAB Function Block Generation

The top-level MATLAB design function cannot have input or output arguments with the `struct` data type.

Results of MATLAB Function Block Generation

After you generate HDL code, an untitled model opens containing a MATLAB Function block.

You can use the MATLAB Function block as part of a larger model in Simulink for simulation and further HDL code generation.

To learn more about generating a MATLAB Function block from a MATLAB algorithm, see “System Design with HDL Code Generation from MATLAB and Simulink” on page 5-29.

System Design with HDL Code Generation from MATLAB and Simulink

This example shows how to generate a MATLAB Function block from a MATLAB® design for system simulation, code generation, and FPGA programming in Simulink®.

Introduction

HDL Coder can generate HDL code from both MATLAB® and Simulink®. The coder can also generate a Simulink® component, the MATLAB Function block, from your MATLAB code.

This capability enables you to:

1. Design an algorithm in MATLAB;
2. Generate a MATLAB Function block from your MATLAB design;
3. Use the MATLAB component in a Simulink model of the system;
4. Simulate and optimize the system model;
5. Generate HDL code; and
6. Program an FPGA with the entire system design.

In this example, you will generate a MATLAB Function block from MATLAB code that implements a FIR filter.

MATLAB Design

The MATLAB code used in the example is a simple FIR filter. The example also shows a MATLAB testbench that exercises the filter.

```
design_name = 'mlhdlc_fir';
testbench_name = 'mlhdlc_fir_tb';
```

1. Design: mlhdlc_fir
2. Test Bench: mlhdlc_fir_tb

Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_fir'];

% Create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name,'.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name,'.m*']), mlhdlc_temp_dir);
```

Simulate the Design

To simulate the design with the test bench prior to code generation to make sure there are no runtime errors, enter the following command:

mlhdlc_fir_tb

Create a New Project

To create a new HDL Coder project, enter the following command:

```
coder -hdlcoder -new fir_project
```

Next, add the file 'mlhdlc_fir.m' to the project as the MATLAB Function and 'mlhdlc_fir_tb.m' as the MATLAB Test Bench.

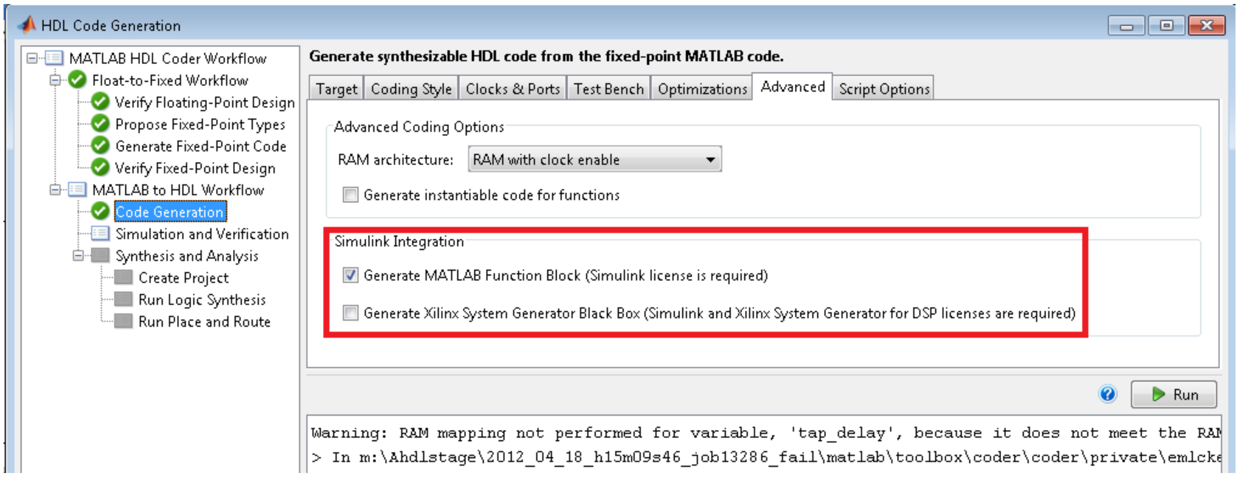
Click the Workflow Advisor button to launch the HDL Workflow Advisor.

Enable the MATLAB Function Block Option

To generate a MATLAB Function block from a MATLAB HDL design, you must have a Simulink license. If the following command returns '1', Simulink is available:

```
license('test', 'Simulink')
```

In the HDL Workflow Advisor Advanced tab, enable the Generate MATLAB Function Block option.



Generate synthesizable HDL code from the fixed-point MATLAB code.

Advanced Coding Options

RAM architecture: RAM with clock enable

☐ Generate instantiable code for functions

Simulink Integration

☑ Generate MATLAB Function Block (Simulink license is required)

☐ Generate Xilinx System Generator Black Box (Simulink and Xilinx System Generator for DSP licenses are required)

```
Warning: RAM mapping not performed for variable, 'tap_delay', because it does not meet the RAM
> In m:\Ahdlstage\2012_04_18_h15m09s46_job13286_fail\matlab\toolbox\coder\coder\private\emlcke
```

Run Floating-Point to Fixed-Point Conversion and Generate Code

To generate a MATLAB Function block, you must also convert your design from floating-point to fixed-point.

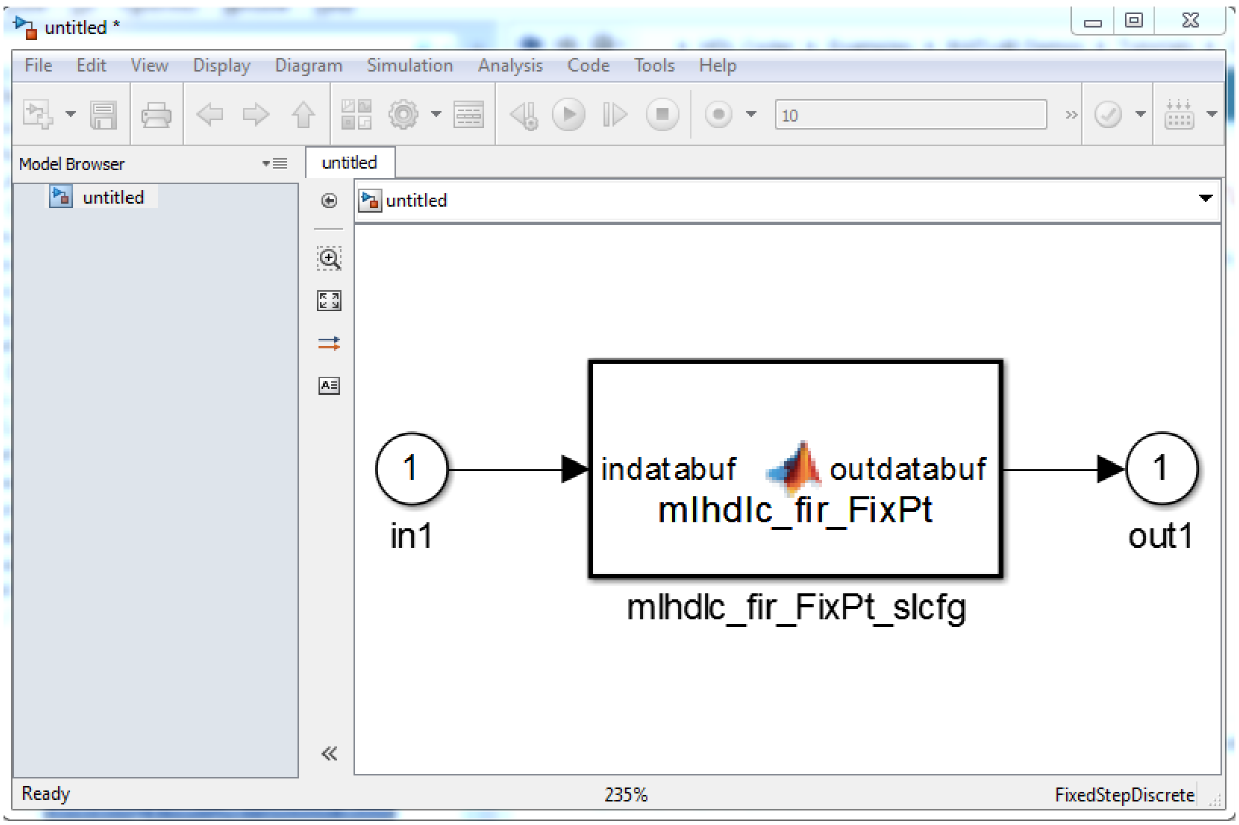
Right-click the 'Code Generation' step and choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

Examine the Generated MATLAB Function Block

An untitled model opens after HDL code generation. It has a MATLAB Function block containing the fixed-point MATLAB code from your MATLAB HDL design. HDL Coder automatically applies settings to the model and MATLAB Function block so that they can simulate in Simulink and generate HDL code.

To generate HDL code from the MATLAB Function block, enter the following command:

```
makehdl('untitled');
```



You can rename and save the new block to use in a larger Simulink design.

Clean Up the Generated Files

You can run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_fir'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

Generate Xilinx System Generator Black Box Block

In this section...
"Requirements for System Generator Black Box Block Generation" on page 5-33
"Enable System Generator Black Box Block Generation" on page 5-33
"Results of System Generator Black Box Block Generation" on page 5-34

Requirements for System Generator Black Box Block Generation

You must have Xilinx® ISE Design Suite 13.4 or later to generate a System Generator Black Box block.

To verify your System Generator setup, at the command line, enter:

```
xlVersion
```

Enable System Generator Black Box Block Generation

Using the GUI

To enable System Generator Black Box block generation using the HDL Workflow Advisor:

1. In the HDL Workflow Advisor, on the left, click **Code Generation**.
2. In the **Advanced** tab, select the **Generate Xilinx System Generator Black Box** option.
3. In the **Clocks & Ports** tab, set the following fields:
   - For **Clock input port**, enter `clk`.
   - For **Clock enable input port**, enter `ce`.
   - For **Drive clock enable at**, select **DUT base rate**.

Using the Command Line

To enable System Generator Black Box block generation, at the command line, enter:

```
hdlcfg = coder.config('hdl');
hdlcfg.GenerateXSGBlock = true;
hdlcfg.ClockInputPort = 'clk';
hdlcfg.ClockEnableInputPort = 'ce';
hdlcfg.EnableRate = 'DutBaseRate';
```

Results of System Generator Black Box Block Generation

After you generate HDL code, you have:

- An XSG subsystem.
- A System Generator Black Box block within the XSG subsystem.
- A System Generator Black Box configuration M-function.

You can use the XSG subsystem in a Simulink model, or use the Black Box block and Black Box configuration M-function in a Xilinx System Generator design.

To learn more about generating a System Generator Black Box block, see “Using Xilinx System Generator for DSP with HDL Coder” on page 18-33.

Generate HDL Code from MATLAB Code Using the Command Line Interface

This example shows how to use the HDL Coder™ command line interface to generate HDL code from MATLAB® code, including floating-point to fixed-point conversion and FPGA programming file generation.

Overview

HDL code generation with the command line interface has the following basic steps:

1. Create a 'fixpt' coder config object. (Optional)
2. Create an 'hdl' coder config object.
3. Set config object parameters. (Optional)
4. Run the codegen command to generate code.

The HDL Coder™ command line interface can use two coder config objects with the codegen command. The optional 'fixpt' coder config object configures the floating-point to fixed-point conversion of your MATLAB® code. The 'hdl' coder config object configures HDL code generation and FPGA programming options.

In this example, we explore different ways you can configure your floating-point to fixed-point conversion and code generation.

The example code implements a discrete-time integrator and its test bench.

Copy the Design and Test Bench Files Into a Temporary Folder

Execute the following code to copy the design and test bench files into a temporary folder:

```
close all;
design_name = 'mlhdlc_dti';
testbench_name = 'mlhdlc_dti_tb';

mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_dti'];

cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);
```

```
copyfile(fullfile(mlhdlc_demo_dir, [design_name,'.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name,'.m*']), mlhdlc_temp_dir);
```

Basic Code Generation With Floating-Point to Fixed-Point Conversion

You can generate HDL code and convert the design from floating-point to fixed-point using the default settings.

You need only your design name, 'mlhdlc_dti', and test bench name, 'mlhdlc_dti_tb':

```
close all;
% Create a 'fixpt' config with default settings
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'mlhdlc_dti_tb';
% Create an 'hdl' config with default settings
hdlcfg = coder.config('hdl'); %#ok<NASGU>
```

After creating 'fixpt' and 'hdl' config objects set up, run the following codegen command to perform floating-point to fixed-point conversion, generate HDL code.

```
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_dti
```

Alternatively, if your design already uses fixed-point types and functions, you can skip fixed-point conversion:

```
hdlcfg = coder.config('hdl'); % Create an 'hdl' config with default settings
hdlcfg.TestBenchName = 'mlhdlc_dti_tb';
codegen -config hdlcfg mlhdlc_dti
```

The rest of this example describes how to configure code generation using the 'hdl' and 'fixpt' objects.

Create a Floating-Point to Fixed-Point Conversion Config Object

To perform floating-point to fixed-point conversion, you need a 'fixpt' config object.

Create a 'fixpt' config object and specify your test bench name:

```
close all;
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'mlhdlc_dti_tb';
```

Set Fixed-Point Conversion Type Proposal Options

The coder can propose fixed-point types based on your choice of either word length or fraction length. These two options are mutually exclusive.

Base the proposed types on a word length of 24:

```
fixptcfg.DefaultWordLength = 24;
fixptcfg.ProposeFractionLengthsForDefaultWordLength = true;
```

Alternatively, you can base the proposed fixed-point types on fraction length. The following code configures the coder to propose types based on a fraction length of 10:

```
fixptcfg.DefaultFractionLength = 10;
fixptcfg.ProposeWordLengthsForDefaultFractionLength = true;
```

Set the Safety Margin

The coder increases the simulation data range on which it bases its fixed-point type proposal by the safety margin percentage. For example, the default safety margin is 4, which increases the simulation data range used for fixed-point type proposal by 4%.

Set the SafetyMargin to 10%:

```
fixptcfg.SafetyMargin = 10;
```

Enable Data Logging

The coder runs the test bench with the design before and after floating-point to fixed-point conversion. You can enable simulation data logging to plot the data differences introduced by fixed-point conversion.

Enable data logging in the 'fixpt' config object:

```
fixptcfg.LogIOForComparisonPlotting = true;
```

View the Numeric Type Proposal Report

Configure the coder to launch the type proposal report after the coder has proposed fixed-point types:

```
fixptcfg.LaunchNumericTypesReport = true;
```

Specify a Type For a Design Variable

If you want to specify the fixed-point data type for a variable in your design, you can create a type specification, set its fields, and associate it with the variable.

The type specification has the following fields:

- IsInteger: Can be true or false

- ProposedType: A type string, like 'ufix15' or 'int32'.
- RoundingMethod: Can be 'ceil', 'convergent', 'fix', 'floor', 'nearest', or 'round'.
- OverflowAction: Can be 'saturate' or 'wrap'.

Create a type specification and associate it with the 'delayed_xout' variable:

Create a type specification object.

```
  typeSpec = coder.FixPtTypeSpec;
```

Set fields in the typeSpec object.

```
  typeSpec.ProposedType = 'ufix15';
  typeSpec.RoundingMethod = 'nearest';
  typeSpec.OverflowAction = 'saturate';
```

Associate the type specification with the variable, 'yt'.

```
  fixptcfg.addTypeSpecification('mlhdlc_dti', 'yt', typeSpec)
```

Create an HDL Code Generation Config Object

To generate code, you must create an 'hdl' config object and set your test bench name:

```
hdlcfg = coder.config('hdl');
hdlcfg.TestBenchName = 'mlhdlc_dti_tb';
```

Set the Target Language

You can generate either VHDL or Verilog code. The coder generates VHDL code by default.

To generate Verilog code:

```
hdlcfg.TargetLanguage = 'Verilog';
```

Generate HDL Test Bench Code

Generate an HDL test bench from your MATLAB® test bench:

```
hdlcfg.GenerateHDLTestBench = true;
```

Simulate the Generated HDL Code Using an HDL Simulator

If you want to simulate your generated HDL code using an HDL simulator, you must also generate the HDL test bench.

Enable HDL simulation and use the ModelSim simulator:

```
hdlcfg.SimulateGeneratedCode = true;
```

```
hdlcfg.SimulationTool = 'ModelSim'; %  or 'ISIM'
```

Generate an FPGA Programming File

You can generate an FPGA programming file if you have a synthesis tool set up.

Enable synthesis, specify a synthesis tool, and specify an FPGA:

```
% Enable Synthesis.
hdlcfg.SynthesizeGeneratedCode = true;

% Configure Synthesis tool.
hdlcfg.SynthesisTool = 'Xilinx ISE'; %  or 'Altera Quartus II';
hdlcfg.SynthesisToolChipFamily = 'Virtex7';
hdlcfg.SynthesisToolDeviceName = 'xc7vh580t';
hdlcfg.SynthesisToolPackageName = 'hcg1155';
hdlcfg.SynthesisToolSpeedValue = '-2G';
```

Run Code Generation

Now that you have your 'fixpt' and 'hdl' config objects set up, run the codegen command to perform floating-point to fixed-point conversion, generate HDL code, and generate an FPGA programming file:

```
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_dti
```

Specify the Clock Enable Rate

In this section...
"Why Specify the Clock Enable Rate?" on page 5-40
"How to Specify the Clock Enable Rate" on page 5-40

Why Specify the Clock Enable Rate?

When HDL Coder performs area optimizations, it might upsample parts of your design (DUT), and thereby introduce an increase in your required DUT clock frequency.

If the coder upsamples your design, it generates a message indicating the ratio between the new clock frequency and your original clock frequency. For example, the following message indicates that your design's new required clock frequency is 4 times higher than the original frequency:

```
The design requires 4 times faster clock with respect to the base rate = 1
```

This frequency increase introduces a rate mismatch between your input clock enable and output clock enable, because the output clock enable runs at the slower original clock frequency.

With the **Drive clock enable at** option, you can choose whether to drive the input clock enable at the faster rate (**DUT base rate**) or at a rate that is less than or equal to the original clock enable rate (**Input data rate**).

How to Specify the Clock Enable Rate

1 In the HDL Workflow Advisor, select **MATLAB to HDL Workflow** > **Code Generation**. Click the **Clocks & Ports** tab.

2 For the **Drive clock enable at** option, select **Input data rate** or **DUT base rate**.

Drive clock enable at Option	Clock Enable Behavior
Input data rate (default)	Each assertion of the input clock enable produces an output clock enable assertion. You can assert the input clock enable at a maximum rate of once every N clocks.

Drive clock enable at Option	Clock Enable Behavior
	N = the upsampled clock rate / original clock rate. For example, if you see the message, “`The design requires 4 times faster clock with respect to the base rate = 1`”, your maximum input clock enable rate is once every 4 clocks.
DUT base rate	Input clock enable rate does not match the output clock enable rate. You must assert the input clock enable with your input data N times to get 1 output clock enable assertion. N = the upsampled clock rate / original clock rate. For example, if you see the message, “`The design requires 4 times faster clock with respect to the base rate = 1`”, you must assert the input clock enable 4 times to get 1 output clock enable assertion.

Specify Test Bench Clock Enable Toggle Rate

In this section...
"When to Specify Test Bench Clock Enable Toggle Rate" on page 5-42
"How to Specify Test Bench Clock Enable Toggle Rate" on page 5-42

When to Specify Test Bench Clock Enable Toggle Rate

When you want the test bench to drive your input data at a slower rate than the maximum input clock enable rate, specify the test bench clock enable toggle rate.

This specification can help you to achieve better test coverage, and to simulate the real world input data rate.

Note: The maximum input clock enable rate is once every N clock cycles. N = the upsampled clock rate / original clock rate. Refer to the clock enable behavior for **Input data rate**, in "Specify the Clock Enable Rate" on page 5-40.

How to Specify Test Bench Clock Enable Toggle Rate

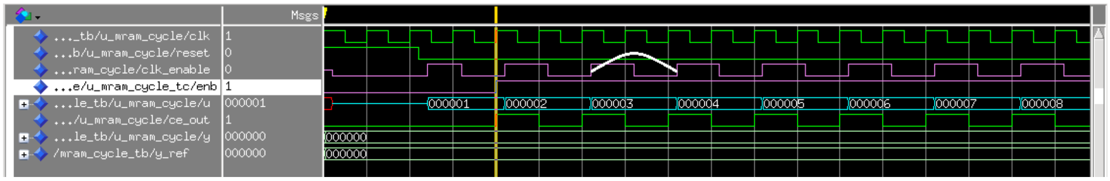
To set your test bench clock enable toggle rate:

1 In the HDL Workflow Advisor, select **MATLAB to HDL Workflow** > **Code Generation**.
2 In the **Clocks & Ports** tab, for the **Drive clock enable at** option, select **Input data rate**.
3 In the **Test Bench** tab, for **Input data interval**, enter 0 or an integer greater than the maximum input clock enable interval.

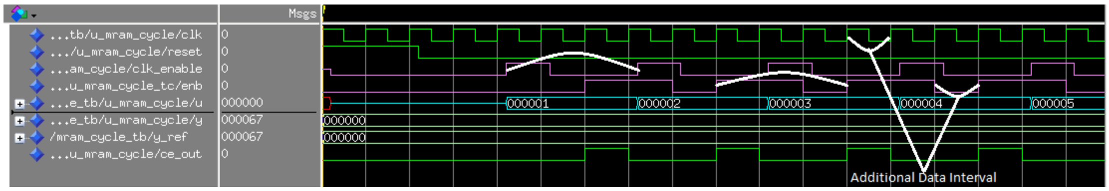
Input data interval, I	Test Bench Clock Enable Behavior
I = 0 (default)	Asserts at the maximum input clock enable rate, or once every N cycles. N = the upsampled clock rate / original clock rate.
I < N	Not valid; generates an error.

Input data interval, I	Test Bench Clock Enable Behavior
I = N	Same as I = 0.
I > N	Asserts every I clock cycles.

For example, this timing diagram shows clock enable behavior with **Input data interval** = 0. Here, the maximum input clock enable rate is once every 2 cycles.


Msgs
..._tb/u_mram_cycle/clk 1
...b/u_mram_cycle/reset 0
...ram_cycle/clk_enable 0
...e/u_mram_cycle_tc/enb 1
...le_tb/u_mram_cycle/u 000001
.../u_mram_cycle/ce_out 1
...le_tb/u_mram_cycle/y 000000
/mram_cycle_tb/y_ref 000000
000001
000002
000003
000004
000005
000006
000007
000008
000000
000000

The following timing diagram shows the same test bench and DUT with **Input data interval** = 3.



Generate an HDL Coding Standard Report from MATLAB

In this section...
"Using the HDL Workflow Advisor" on page 5-44
"Using the Command Line" on page 5-46

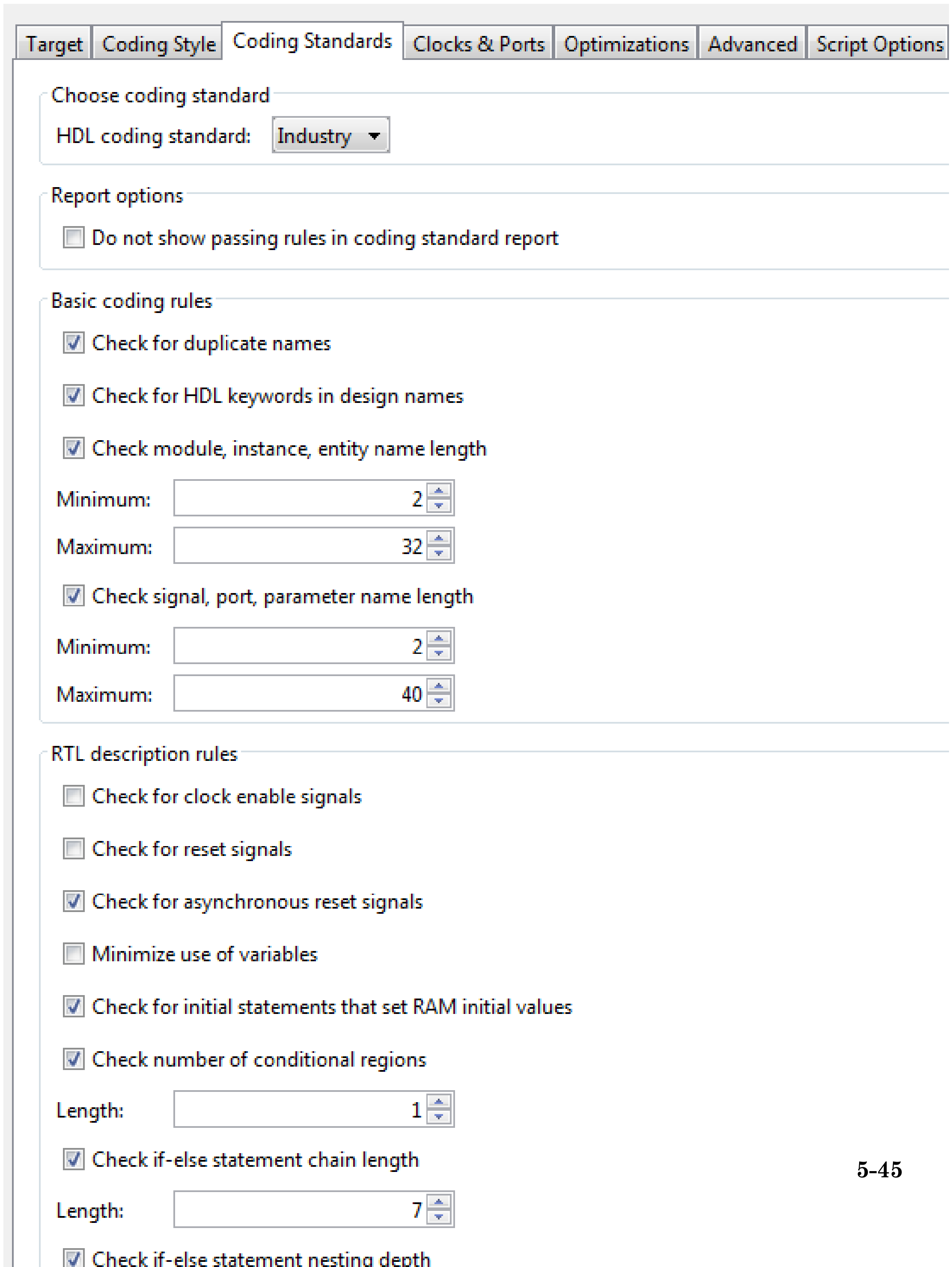
You can generate an HDL coding standard report that shows how well your generated code follows industry standards. You can optionally customize the coding standard report and the coding standard rules.

Using the HDL Workflow Advisor

To generate an HDL coding standard report using the HDL Workflow Advisor:

1 In the **HDL Code Generation** task, select the **Coding Standards** tab.

2 For **HDL coding standard**, select **Industry**.



Choose coding standard

HDL coding standard: Industry

Report options

- [ ] Do not show passing rules in coding standard report

Basic coding rules

- [x] Check for duplicate names
- [x] Check for HDL keywords in design names
- [x] Check module, instance, entity name length

Minimum: 2

Maximum: 32

- [x] Check signal, port, parameter name length

Minimum: 2

Maximum: 40

RTL description rules

- [ ] Check for clock enable signals
- [ ] Check for reset signals
- [x] Check for asynchronous reset signals
- [ ] Minimize use of variables
- [x] Check for initial statements that set RAM initial values
- [x] Check number of conditional regions

Length: 1

- [x] Check if-else statement chain length

Length: 7

- [x] Check if-else statement nesting depth

3 Optionally, using the other options in the **Coding Standards** tab, customize the coding standard rules.

4 Click **Run** to generate code.

   After you generate code, the message window shows a link to the HTML compliance report.

Using the Command Line

To generate an HDL coding standard report using the command line interface, set the `HDLCodingStandard` property to `Industry` in the coder.HdlConfig object.

For example, to generate HDL code and an HDL coding standard report for a design, `mlhdlc_sfir`, with a testbench, `mlhdlc_sfir_tb`, enter the following commands:

```
hdlcfg = coder.config('hdl');
hdlcfg.TestBenchName = 'mlhdlc_sfir_tb';
hdlcfg.HDLCodingStandard='Industry';
codegen -config hdlcfg mlhdlc_sfir

### Generating Resource Utilization Report resource_report.html
### Generating default Industry script file mlhdlc_sfir_mlhdlc_sfir_default.prj
### Industry Compliance report with 0 errors, 8 warnings, 4 messages.
### Generating Industry Compliance Report mlhdlc_sfir_Industry_report.html
```

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, suppose you have a design, `mlhdlc_sfir`, and testbench, `mlhdlc_sfir_tb`. 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:

```
hdlcfg = coder.config('hdl');
hdlcfg.TestBenchName = 'mlhdlc_sfir_tb';
hdlcfg.HDLCodingStandard='Industry';
cso = hdlcoder.CodingStandard('Industry');
cso.IfElseChain.length = 5;
hdlcfg.HDLCodingStandardCustomizations = cso;
codegen -config hdlcfg mlhdlc_sfir
```

See Also

Properties
HDL Coding Standard Customization Properties

More About

- “HDL Coding Standard Report” on page 17-2
- “Basic Coding Practices” on page 17-9
- “RTL Description Techniques” on page 17-20
- “RTL Design Methodology Guidelines” on page 17-48

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 HDL Workflow Advisor

1. In the HDL Workflow Advisor, select the **HDL Code Generation** task.
2. In the **Script Options** tab, select **Lint**.
3. For **Choose lint tool**, select **Ascent Lint**, **HDL Designer**, **Leda**, **SpyGlass**, or **Custom**.
4. Optionally, enter text to customize the **Lint script initialization**, **Lint script command**, and **Lint script termination** fields. For a custom tool, you must specify these fields.

After you generate code, the command 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` property to `AscentLint`, `HDLDesigner`, `Leda`, `SpyGlass` or `Custom` in your coder.HdlConfig object.

To disable HDL lint tool script generation, set the `HDLLintTool` property to `None`.

For example, to generate a default SpyGlass lint script using a `coder.HdlConfig` object, *`hdlcfg`*, enter:

```
hdlcfg.HDLLintTool = 'SpyGlass';
```

After you generate code, the command window shows a link to the lint tool script.

To generate an HDL lint tool script with custom initialization, command, and termination strings, use the `HDLLintTool`, `HDLLintInit`, `HDLLintCmd`, and `HDLLintTerm` properties.

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, termination, and command strings:

```
hdlcfg.HDLLintTool = 'Leda';
hdlcfg.HDLLintInit = 'myInitialization';
hdlcfg.HDLLintCmd = 'myCommand %s';
hdlcfg.HDLLintTerm = 'myTermination';
```

After you generate code, the command window shows a link to the lint tool script.

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.

For **Lint script command** or `HDLLintCmd`, specify the lint command in the following format:

```
custom_lint_tool_command -option1 -option2 %s
```

For example, to set the `HDLLintCmd` for a `coder.HdlConfig` object, *`hdlcfg`*, where the lint command is *`custom_lint_tool_command -option1 -option2`*, enter:

```
hdlcfg.HDLLintCmd = 'custom_lint_tool_command -option1 -option2 %s';
```

Generate a Board-Independent IP Core from MATLAB

In this section...
"Generate a Board-Independent IP Core" on page 5-50
"Requirements and Limitations for IP Core Generation" on page 5-52

Generate a Board-Independent IP Core

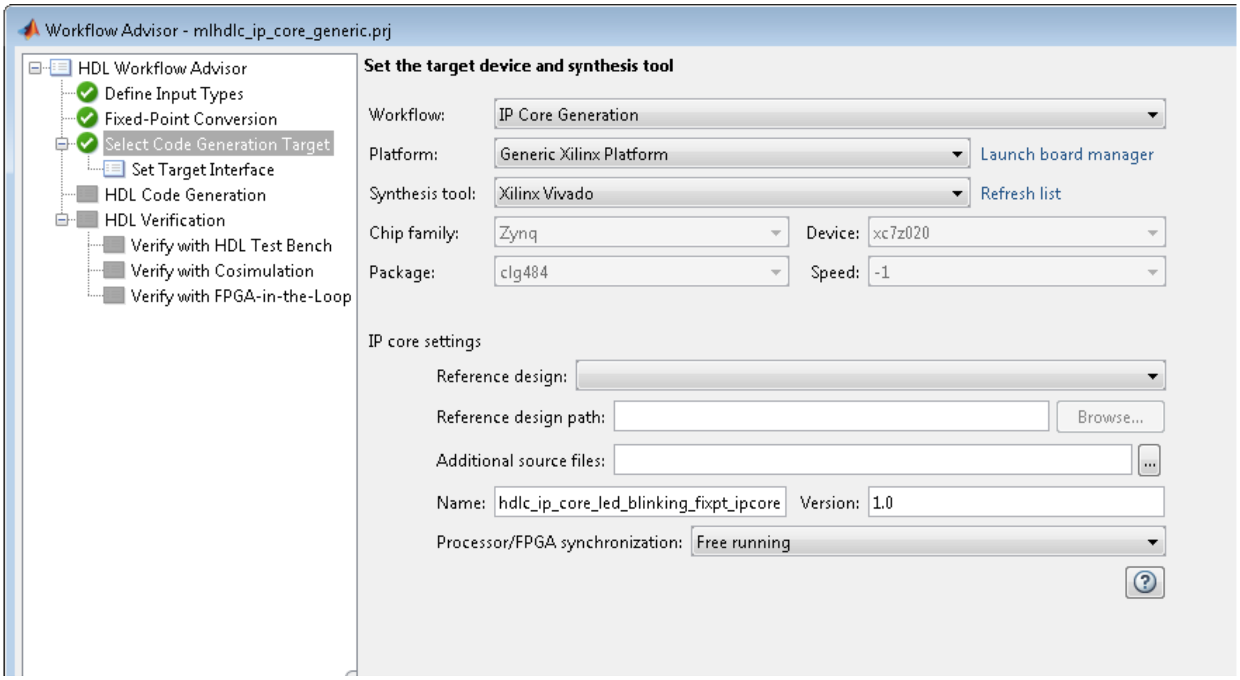
To generate a board-independent IP core to use in an embedded system integration environment, such as Altera® Qsys, Xilinx EDK, or Xilinx IP Integrator:

1. Create an HDL Coder project containing your MATLAB design and test bench, or open an existing project.
2. In the HDL Workflow Advisor, define input types and perform fixed-point conversion.

   To learn how to convert your design to fixed-point, see "HDL Code Generation from a MATLAB Algorithm".
3. In the HDL Workflow Advisor, in the **Select Code Generation Target** task:

   - **Workflow**: Select `IP Core Generation`.
   - **Platform**: Select `Generic Xilinx Platform` or `Generic Altera Platform`.

     Depending on your selection, the coder automatically sets **Synthesis tool**. For example, if you select `Generic Xilinx Platform`, **Synthesis tool** automatically changes to `Xilinx Vivado`. You can change the **Synthesis tool** to `Xilinx ISE`.
   - **Additional source files**: If you are using an `hdl.BlackBox` System object to include existing Verilog or VHDL code, enter the file names. Enter each file name manually, separated with a semicolon (`;`), or by using the **...** button. The source file language must match your target language.


Workflow Advisor - mlhdlc_ip_core_generic.prj

Set the target device and synthesis tool

Workflow: IP Core Generation

Platform: Generic Xilinx Platform

Synthesis tool: Xilinx Vivado

Chip family: Zynq

Device: xc7z020

Package: clg484

Speed: -1

IP core settings

Reference design:

Reference design path:

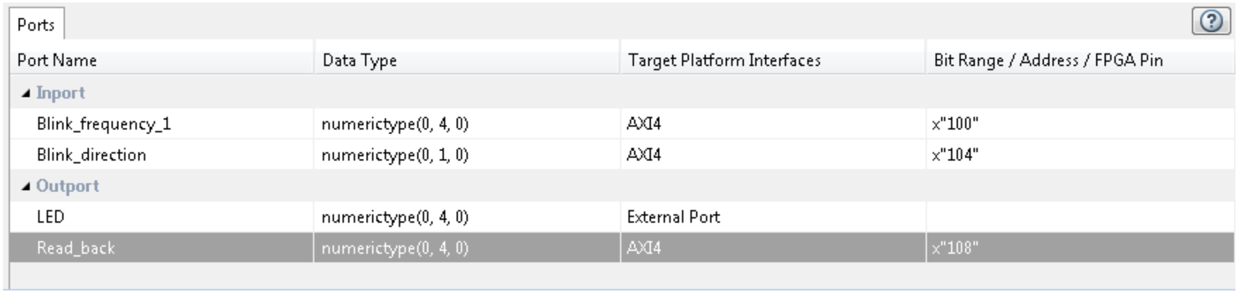
Additional source files:

Name: hdlc_ip_core_led_blinking_fixpt_ipcore

Version: 1.0

Processor/FPGA synchronization: Free running

4 In the **Set Target Interface** step, for each port, select an option from the **Target Platform Interfaces** drop-down list.


Ports

Port Name	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
Inport			
Blink_frequency_1	numerictype(0, 4, 0)	AXI4	x"100"
Blink_direction	numerictype(0, 1, 0)	AXI4	x"104"
Outport			
LED	numerictype(0, 4, 0)	External Port	
Read_back	numerictype(0, 4, 0)	AXI4	x"108"

5 In the **HDL Code Generation** step, optionally specify code generation options, then click **Run**.

6 In the HDL Workflow Advisor message pane, click the IP core report link to view detailed documentation for your generated IP core.

Requirements and Limitations for IP Core Generation

You cannot map to both an AXI4 interface and AXI4-Lite interface in the same IP core.

To map your design function inputs or outputs to an AXI4-Lite interface, the input and outputs must:

- Have a bit width less than or equal to 32 bits.
- Be scalar.

When mapping design function inputs or outputs to an AXI4-Stream Video interface, the following requirements apply:

- Ports must have a 32-bit width.
- Ports must be scalar.
- You can have a maximum of one input video port and one output video port.

The AXI4-Stream Video interface is not supported in **Coprocessing – blocking** processor/FPGA synchronization mode.

Minimize Clock Enables

In this section...

By default, HDL Coder generates code in a style that is intended to map to registers with clock enables, and the DUT has a top-level clock enable port.

If you do not want to generate registers with clock enables, you can minimize the clock enable logic. For example, if your target hardware contains registers without clock enables, you can save hardware resources by minimizing the clock enable logic.

The following VHDL code shows the default style of generated code, which uses clock enables. The `enb` signal is the clock enable:

```
Unit_Delay_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      Unit_Delay_out1 <= to_signed(0, 32);
    ELSIF clk'EVENT AND clk = '1' THEN
      IF enb = '1' THEN
        Unit_Delay_out1 <= In1_signed;
      END IF;
    END IF;
  END PROCESS Unit_Delay_process;
```

The following VHDL code shows the style of code you generate if you minimize clock enables:

```
Unit_Delay_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      Unit_Delay_out1 <= to_signed(0, 32);
    ELSIF clk'EVENT AND clk = '1' THEN
      Unit_Delay_out1 <= In1_signed;
    END IF;
  END PROCESS Unit_Delay_process;
```

Using the GUI

To minimize clock enables, in the HDL Workflow Advisor, on the **HDL Code Generation** > **Set Code Generation Options** > **Set Optimization Options** > **General** tab, select **Minimize clock enables**.

Using the Command Line

To minimize clock enables, in the `coder.HdlConfig` configuration object, set the `MinimizeClockEnables` property to `true`. For example:

```
hdlCfg = coder.config('hdl')
hdlCfg.MinimizeClockEnables = true;
```

Limitations

If you specify area optimizations that the coder implements by increasing the clock rate in certain regions of the design, you cannot minimize clock enables. The following optimizations prevent clock enable minimization:

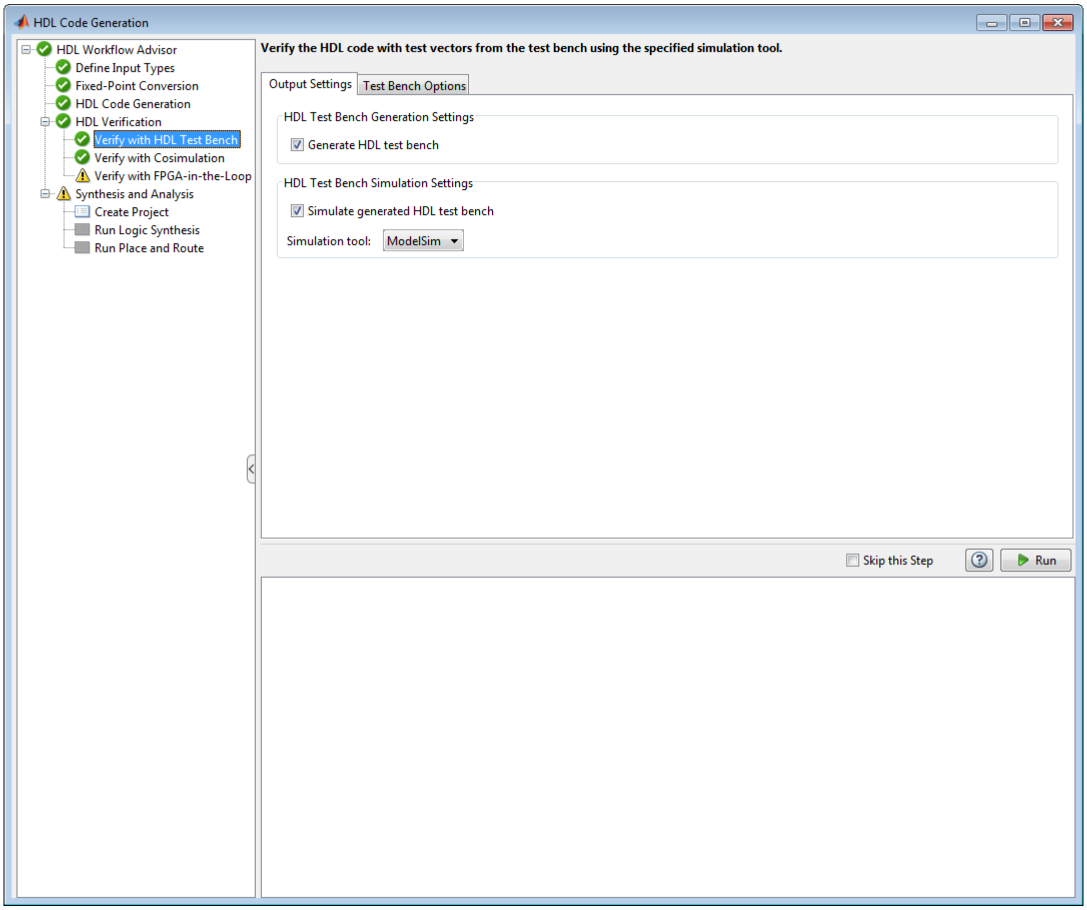
- Resource sharing
- RAM mapping
- Loop streaming

Verification

Verify Code with HDL Test Bench

Simulate the generated HDL design under test (DUT) with test vectors from the test bench using the specified simulation tool.

1 Start the MATLAB to HDL Workflow Advisor.



Verify the HDL code with test vectors from the test bench using the specified simulation tool.

HDL Test Bench Generation Settings

☑ Generate HDL test bench

HDL Test Bench Simulation Settings

☑ Simulate generated HDL test bench

Simulation tool: ModelSim

2 At step **HDL Verification**, click **Verify with HDL Test Bench**.

3 Select **Generate HDL test bench**.

This option enables HDL Coder to generate HDL test bench code from your MATLAB test script.

4 Optionally, select **Simulate generated HDL test bench**. This option enables MATLAB to simulate the HDL test bench with the HDL DUT.

   If you select this option, you must also select the **Simulation tool**.

5 For **Test Bench Options**, select and set the optional parameters according to the descriptions in the following table.

HDL Test Bench Parameter	Description
Test bench name postfix	Specify the postfix for the test bench name.
Force clock	Enable for test bench to force clock input signals.
Clock high time (ns)	Specify the number of nanoseconds the clock is high.
Clock low time (ns)	Specify the number of nanoseconds the clock is low.
Hold time (ns)	Specify the hold time for input signals and forced reset signals.
Force clock enable	Enable to force clock enable.
Clock enable delay (in clock cycles)	Specify time (in clock cycles) between deassertion of reset and assertion of clock enable.
Force reset	Enable for test bench to force reset input signals.
Reset length (in clock cycles)	Specify time (in clock cycles) between assertion and deassertion of reset.
Hold input data between samples	Enable to hold subrate signals between clock samples.
Input data interval	Specifies the number of clock cycles between assertions of clock enable. For more information, see “Specify Test

HDL Test Bench Parameter	Description
	Bench Clock Enable Toggle Rate” on page 5-42.
Initialize test bench inputs	Enable to initialize values on inputs to test bench before test bench drives data to DUT.
Multi file test bench	Enable to divide generated test bench into helper functions, data, and HDL test bench code.
Test bench data file name postfix	Specify the character vector to append to name of test bench data file when generating multi-file test bench.
Test bench reference postfix	Specify the character vector to append to names of reference signals in test bench code.
Ignore data checking (number of samples)	Specify the number of samples at the beginning of simulation during which output data checking is suppressed.
Simulation iteration limit	Specify the maximum number of test samples to use during simulation of generated HDL code.

6 Optionally, select **Skip this step** if you don’t want to use the HDL test bench to verify the HDL DUT.

7 Click **Run**.

If the test bench and simulation is successful, you should see messages similar to these in the message pane:

```
### Begin TestBench generation.
### Collecting data...
### Begin HDL test bench file generation with logged samples
### Generating test bench: mlhdlc_sfir_fixpt_tb.vhd
### Creating stimulus vectors...
### Simulating the design 'mlhdlc_sfir_fixpt' using 'ModelSim'.
### Generating Compilation Report mlhdlc_sfir_fixpt_vsim_log_compile.txt
### Generating Simulation Report mlhdlc_sfir_fixpt_vsim_log_sim.txt
### Simulation successful.
### Elapsed Time: 113.0315 sec(s)
```

If there are errors, those messages appear in the message pane. Fix errors and click **Run**.

Test Bench Generation

In this section...

How Test Bench Generation Works

HDL Coder writes the DUT stimulus and reference data from your MATLAB or Simulink simulation to data files (`.dat`).

During HDL simulation, the HDL test bench reads the saved stimulus from the `.dat` files. The test bench compares the actual DUT output with the expected output, which is also saved in .dat files. After you generate code, the message window displays links to the test bench data files.

Reference data is delayed by one clock cycle in the waveform viewer compared to default test bench generation due to the delay in reading data from files.

Test Bench Data Files

The coder saves stimulus and reference data for each DUT input and output in a separate test bench data file (`.dat`), with the following exceptions:

- Two files are generated for the real and imaginary parts of complex data.
- Constant DUT input data is written to the test bench as constants.

Vector input or output data is saved as a single file.

Test Bench Data Type Limitations

If you have double, single, or enumeration data types at the DUT inputs and outputs, the simulation data is generated as constants in the test bench code, instead of writing the simulation data to files.

Use Constants Instead of File I/O

You can generate test bench stimulus and reference data as constants in the test bench code instead of using file I/O. However, simulating a long running test bench that uses constants requires more memory than a test bench that uses file I/O.

Test bench generation automatically generates data as constants if your DUT inputs or outputs use data types that are not supported for file I/O. For details, see “Test Bench Data Type Limitations” on page 6-6.

To generate a test bench that uses constants instead of file I/O:

1. In the HDL Workflow Advisor, select the **HDL Verification** > **Verify with HDL Test Bench** task.
2. In the **Test bench Options** tab, disable the **Use file I/O for test bench** option.

7

Deployment

Generate Synthesis Scripts

You can generate customized synthesis scripts for the following tools:

- Xilinx Vivado®
- Xilinx ISE
- Microsemi Libero
- Mentor Graphics® Precision
- Altera Quartus II
- Synopsys® Synplify Pro®

You can also generate a synthesis script for a custom tool by specifying the fields manually.

To generate a synthesis script:

1. In the HDL Workflow Advisor, select the **HDL Code Generation** task.
2. In the **Script Options** tab, select **Synthesis**.
3. For **Choose synthesis tool**, select a tool option.
4. If you want to customize your script, use the **Synthesis file postfix**, **Synthesis initialization**, **Synthesis command**, and **Synthesis termination** text fields to do so.

After you generate code, your synthesis Tcl script (`.tcl`) is in the same folder as your generated HDL code.

Optimization

RAM Mapping

RAM mapping is an area optimization that maps storage and delay elements in your MATLAB code to RAM. Without this optimization, storage and delay elements are mapped to registers. RAM mapping can therefore reduce the area of your design in the target hardware.

You can map the following MATLAB code elements to RAM:

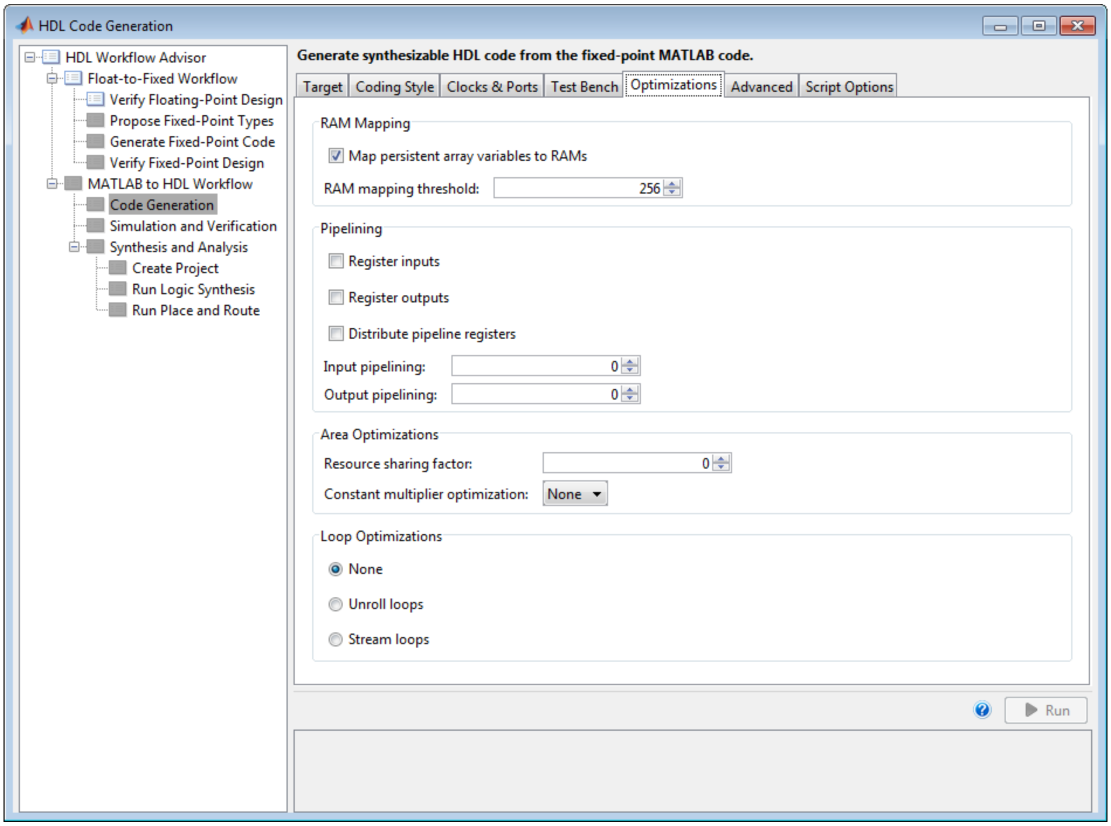
- persistent array variable
- `dsp.Delay` System object
- `hdl.RAM` System object

Map Persistent Arrays and dsp.Delay to RAM

In this section...

How To Enable RAM Mapping

1. In the HDL Workflow Advisor, select **MATLAB to HDL Workflow** > **Code Generation** > **Optimizations** tab.
2. Select the **Map persistent array variables to RAMs** option.
3. Set the **RAM mapping threshold** to the size (in bits) of the smallest persistent array, user-defined System object private property, or `dsp.Delay` that you want to map to RAM.



Generate synthesizable HDL code from the fixed-point MATLAB code.

RAM Mapping

[✓] Map persistent array variables to RAMs

RAM mapping threshold: 256

Pipelining

[ ] Register inputs

[ ] Register outputs

[ ] Distribute pipeline registers

Input pipelining: 0

Output pipelining: 0

Area Optimizations

Resource sharing factor: 0

Constant multiplier optimization: None

Loop Optimizations

[● None] Unroll loops Stream loops

RAM Mapping Requirements for Persistent Arrays and System object Properties

The following table shows a summary of the RAM mapping behavior for persistent arrays and private properties of a user-defined System object.

Map Persistent Array Variables to RAMs Setting	Mapping Behavior
on	Map to RAM. For restrictions, see “RAM Mapping Restrictions” on page 8-5.
off	Map to registers in the generated HDL code.

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.

RAM Mapping Requirements for `dsp.Delay` System Objects

A summary of the mapping behavior for a `dsp.Delay` System object is in the following table.

Map Persistent Array Variables to RAMs Option	Mapping Behavior
on	A `dsp.Delay` System object maps to a block RAM when all of the following conditions are true: • `Length` property is greater than 4. • `InitialConditions` property is 0. • Delay input data type is one of the following: • Real scalar with a non-floating-point data type. • Complex scalar with real and imaginary parts that are non-floating-point. • Vector where each element is either a non-floating-point real scalar or complex scalar.

Map Persistent Array Variables to RAMs Option	Mapping Behavior
	• `RAMSize` is greater than or equal to the **RAM Mapping Threshold** value. • `RAMSize` is the product `Length * InputWordLength`. • `InputWordLength` is the number of bits that represent the input data type. If any of the conditions are false, the `dsp.Delay` System object maps to registers in the HDL code.
off	A `dsp.Delay` System object maps to registers in the generated HDL code.

RAM Mapping Comparison for MATLAB Code

`hdl.RAM`, `dsp.Delay`, persistent array variables, and user-definedSystem object private properties can map to RAM, but have different attributes. The following table summarizes the differences.

Attribute	`hdl.RAM`	`dsp.Delay`	**Persistent Arrays and User-Defined System object Properties**
RAM mapping criteria	Unconditionally maps to RAM	Maps to RAM in HDL code under specific conditions. See “RAM Mapping Requirements for dsp.Delay System Objects” on page 8-6.	Maps to RAM in HDL code under specific conditions. See “RAM Mapping Requirements for Persistent Arrays and System object Properties” on page 8-4.
Address generation and port mapping	User specified	Automatic	Automatic
Access scheduling	User specified	Automatically inferred	Automatically inferred
Overclocking	None	None	Local multirate if access schedule requires it.
Latency with respect to simulation in MATLAB.	0	0	2 cycles if local multirate; 1 cycle otherwise.
RAM type	User specified	Dual port	Dual port

Pipelining MATLAB Code

Pipelining helps achieve a higher maximum clock rate by inserting registers at strategic points in the hardware to break the critical path. However, the higher clock rate comes at the expense of increased chip area and increased initial latency.

Port Registers

Input and output port registers for modules help partition a larger design so the critical path does not extend across module boundaries. Having a port register at each input and output port is a good design practice for synchronous interfaces.

Distributed pipelining does not affect port registers.

To learn how to insert port registers, see “Register Inputs and Outputs” on page 8-10.

Input and Output Pipeline Registers

You can insert multiple input and output pipeline stages. Distributed pipelining can move these input and output pipeline registers to help reduce your critical path within the module.

If you insert input and output pipeline stages without applying distributed pipelining, the registers stay at the DUT inputs and outputs.

For more information, see “Insert Input and Output Pipeline Registers” on page 8-11.

Operation Pipelining

Operation pipelining inserts one or more registers at the output of a specific expression in your MATLAB code. If you know a specific expression is part of the critical path, you can add a pipeline register at its output to reduce your critical path.

To learn how to insert a pipeline register at the output of a MATLAB expression, see “Pipeline MATLAB Expressions” on page 8-12.

Register Inputs and Outputs

To insert input or output port registers:

1. In the HDL Workflow Advisor, select the **HDL Code Generation** task and select the **Optimizations** tab.
2. Enable **Register inputs**, **Register outputs**, or both.

To learn more about input and output port registers, see "Port Registers" on page 8-9.

Insert Input and Output Pipeline Registers

To insert input or output pipeline register stages:

1. In the HDL Workflow Advisor, select the **HDL Code Generation** task and select the **Optimizations** tab.
2. For **Input pipelining**, **Output pipelining**, or both, enter the number of pipeline register stages.

To learn more about input and output pipeline registers, see "Input and Output Pipeline Registers" on page 8-9.

Pipeline MATLAB Expressions

In this section...

With the `coder.hdl.pipeline` pragma, you can specify the placement and number of pipeline registers in the HDL code generated for a MATLAB expression.

If you insert pipeline registers and enable distributed pipelining, HDL Coder automatically moves the pipeline registers to break the critical path.

How To Pipeline a MATLAB Expression

To insert pipeline registers at the output of an expression in MATLAB code, place the expression in the `coder.hdl.pipeline` pragma. Specify the number of registers.

You can insert pipeline registers in the generated HDL code:

- At the output of the entire right side of an assignment statement.

  The following code inserts three pipeline registers at the output of a MATLAB expression, `a + b * c`:

  ```
  y = coder.hdl.pipeline(a + b * c, 3);
  ```
- At an intermediate stage within a longer MATLAB expression.

  The following code inserts five pipeline registers after the computation of `b * c` within a longer expression, `a + b * c`:

  ```
  y = a + coder.hdl.pipeline(b * c, 5);
  ```
- By nesting multiple instances of the pragma.

  The following code inserts five pipeline registers after the computation of `b * c`, and two pipeline registers at the output of the whole expression, `a + b * c`:

  ```
  y = coder.hdl.pipeline(a + coder.hdl.pipeline(b * c, 5),2);
  ```

Alternatively, to insert one pipeline register instead of multiple pipeline registers, you can omit the second argument in the pragma:

```
y = coder.hdl.pipeline(a + b * c);
```

```
y = a + coder.hdl.pipeline(b * c);
```

```
y = coder.hdl.pipeline(a + coder.hdl.pipeline(b * c));
```

Limitations of Pipelining for MATLAB Expressions

HDL Coder cannot insert a pipeline register at the output of a MATLAB expression if any of the variables in the expression are:

- In a loop.
- A persistent variable that maps to a state element, like a state register or RAM.
- An output of a function. For example, in the following code, you cannot add a pipeline register for an expression containing `y`:

  ```
  function [y] = myfun(x)
  y = x + 5;
  end
  ```
- In a data feedback loop. For example, in the following code, you cannot pipeline an expression containing the `t` or `pvar` variables:

  ```
  persistent pvar;
  t = u + pvar;
  pvar = t + v;
  ```

You cannot use `coder.hdl.pipeline` to insert a pipeline register for a single variable or other no-op expression. To learn how to insert a pipeline register for a function input variable, see “Register Inputs and Outputs” on page 8-10.

See Also

`coder.hdl.pipeline`

More About

- “Pipelining MATLAB Code” on page 8-9

Distributed Pipelining

In this section...

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.

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.

Selected Bibliography

Leiserson, C.E, and James B. Saxe. "Retiming Synchronous Circuitry." *Algorithmica*. Vol. 6, Number 1, 1991, pp. 5-35.

Optimize MATLAB Loops

In this section...

With loop optimization, you can stream or unroll loops in generated code. Loop streaming is an area optimization, and loop unrolling is a speed optimization.

Loop Streaming

HDL Coder streams a loop by instantiating the loop body once and using that instance for each loop iteration. The coder oversamples the loop body instance to keep the generated loop functionally equivalent to the original loop.

If you stream a loop, the advantage is decreased hardware resource usage because the loop body is instantiated fewer times. The disadvantage is the hardware implementation runs at a lower speed.

You can partially stream a loop. A partially streamed loop instantiates the loop body more than once, so it uses more area than a fully streamed loop. However, a partially streamed loop also uses less oversampling than a fully streamed loop.

Loop Unrolling

HDL Coder unrolls a loop by instantiating multiple instances of the loop body in the generated code. You can also partially unroll a loop. The generated code uses a loop statement that contains multiple instances of the original loop body and fewer iterations than the original loop.

The distributed pipelining and resource sharing can optimize the unrolled code. Distributed pipelining can increase speed. Resource sharing can decrease area.

When loop unrolling creates multiple instances, these instances are likely to increase area. Loop unrolling also makes the code harder to read.

How to Optimize MATLAB Loops

You can specify a global loop optimization by using the HDL Workflow Advisor, or at the command line.

You can also specify a local loop optimization for a specific loop by using the `coder.hdl.loopspec` pragma in the MATLAB code. If you specify both a global and local loop optimization, the local loop optimization overrides the global setting.

Global Loop Optimization

To specify a loop optimization in the Workflow Advisor:

1. In the HDL Workflow Advisor left pane, select **HDL Workflow Advisor** > **HDL Code Generation**.
2. In the **Optimizations** tab, for **Loop Optimizations**, select **None**, **Unroll Loops**, or **Stream Loops**.

To specify a loop optimization at the command line in the MATLAB to HDL workflow, specify the `LoopOptimization` property of the `coder.HdlConfig` object. For example, for a `coder.HdlConfig` object, `hdlcfg`, enter one of the following commands:

```
hdlcfg.LoopOptimization = 'UnrollLoops'; % unroll loops
```

```
hdlcfg.LoopOptimization = 'StreamLoops'; % stream loops
```

```
hdlcfg.LoopOptimization = 'LoopNone'; % no loop optimization
```

Local Loop Optimization

To learn how to optimize a specific MATLAB loop, see `coder.hdl.loopspec`.

Note: If you specify the `coder.unroll` pragma, this pragma takes precedence over `coder.hdl.loopspec`. `coder.hdl.loopspec` has no effect.

Limitations for MATLAB 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 two 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.
- A persistent variable that is initialized to a nonzero value is updated inside the 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.

You cannot use the `coder.hdl.loopspec('stream')` pragma:

- In a subfunction. You must specify it in the top-level MATLAB design function.
- For a loop that is nested within another loop.
- For a loop containing a nested loop, unless the streaming factor is equal to the number of iterations.

See Also

`coder.hdl.loopspec`

Constant Multiplier Optimization

The **Constant multiplier optimization** option enables you to specify use of canonical signed digit (CSD) or factored CSD (FCSD) optimizations for processing coefficient multiplier operations.

The following table shows the **Constant multiplier optimization** values.

Constant Multiplier Optimization Value	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. HDL 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).

To learn how to specify constant multiplier optimization, see “Specify Constant Multiplier Optimization” on page 8-20.

Specify Constant Multiplier Optimization

To specify constant multiplier optimization:

1. In the HDL Workflow Advisor, select the **HDL Code Generation** task and select the **Optimizations** tab.
2. For **Constant multiplier optimization**, select **CSD**, **FCSD**, or **Auto**.

To learn more about the constant multiplier optimization options, see “Constant Multiplier Optimization” on page 8-18.

Distributed Pipelining for Clock Speed Optimization

This example shows how to use the distributed pipelining and loop unrolling optimizations in HDL Coder to optimize clock speed.

Introduction

Distributed pipelining is a design-wide optimization supported by HDL Coder for improving clock frequency. When you turn on the 'Distribute Pipeline Registers' option in HDL Coder, the coder redistributes the input and output pipeline registers of the top level function along with other registers in the design in order to minimize the combinatorial logic between registers and thus maximize the clock speed of the chip synthesized from the generated HDL code.

Consider the following example design of a FIR filter. The combinatorial logic from an input or a register to an output or another register contains a sum of products. Loop unrolling and distributed pipelining moves the output registers at the design level to reduce the amount of combinatorial logic, thus increasing clock speed.

MATLAB® Design

The MATLAB code used in the example is a simple FIR filter. The example also shows a MATLAB test bench that exercises the filter.

```
design_name = 'mlhdlc_fir';
testbench_name = 'mlhdlc_fir_tb';
```

1 Design: mlhdlc_fir
2 Test Bench: mlhdlc_fir_tb

Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

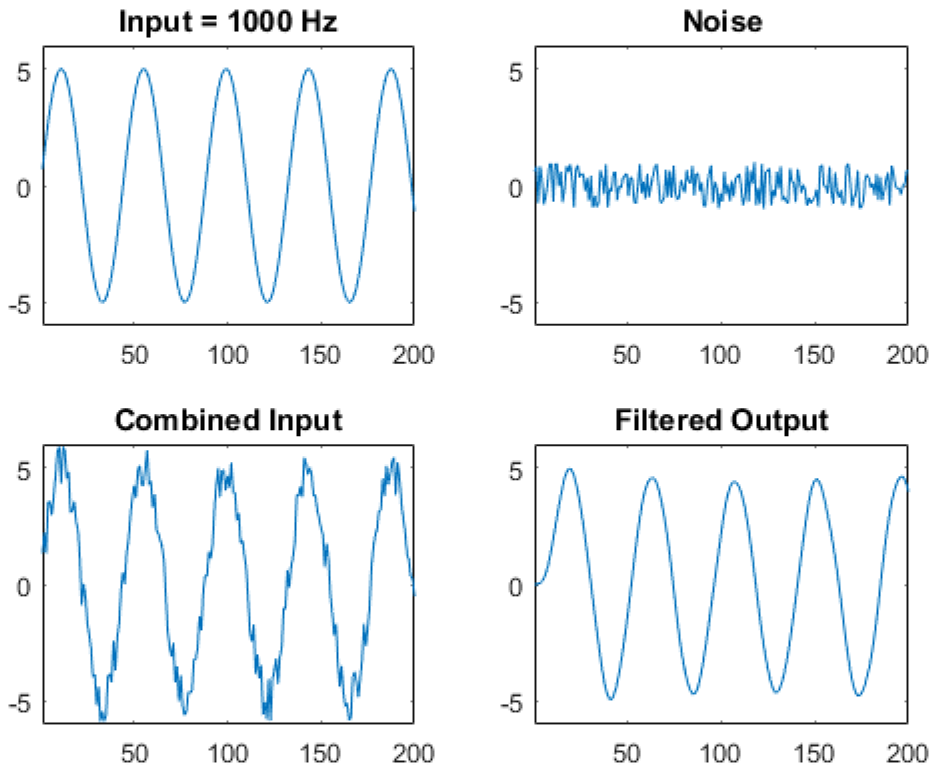
```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_fir'];

% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
```

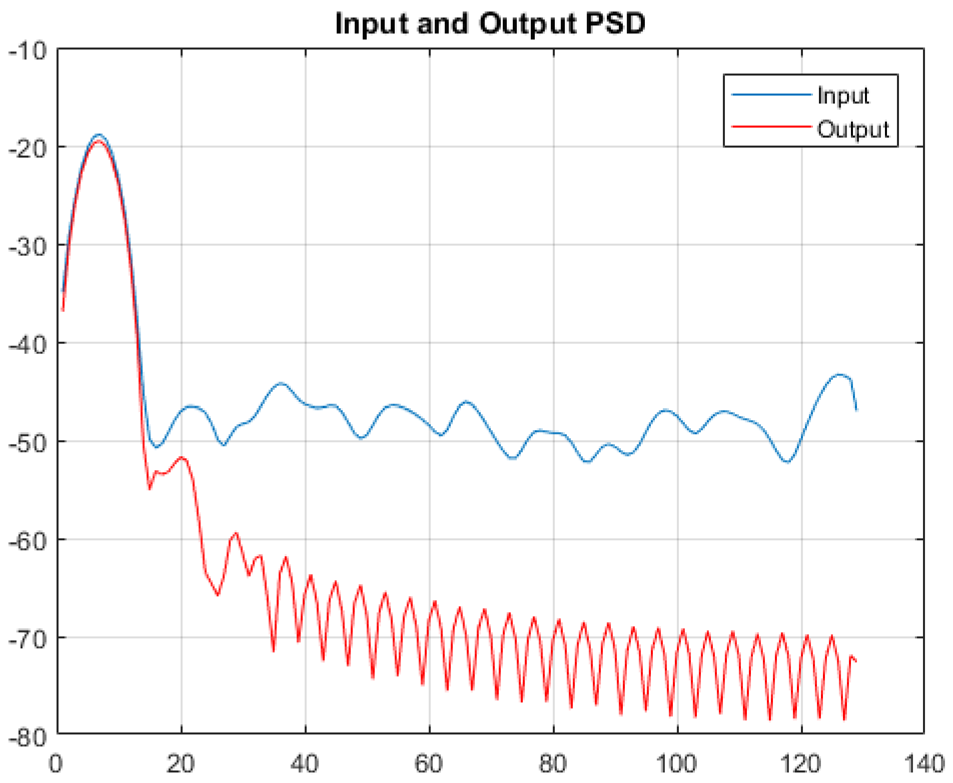
```
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name,'.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name,'.m*']), mlhdlc_temp_dir);
```

Simulate the Design

Simulate the design with the testbench prior to code generation to make sure there are no run-time errors.

```
mlhdlc_fir_tb
```





Create a Fixed-Point Conversion Config Object

To perform fixed-point conversion, you need a 'fixpt' config object.

Create a 'fixpt' config object and specify your test bench name:

```
close all;
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'mlhdlc_fir_tb';
```

Create an HDL Code Generation Config Object

To generate code, you must create an 'hdl' config object and set your test bench name:

```
hdlcfg = coder.config('hdl');
```

```
hdlcfg.TestBenchName = 'mlhdlc_fir_tb';
```

Distributed Pipelining

To increase the clock speed, the user can set a number of input and output pipeline stages for any design. In this particular example Input pipelining option is set to '1' and Output pipelining option is set to '20'. Without any additional options turned on these settings will add one input pipeline register at all input ports of the top level design and 20 output pipeline registers at each of the output ports.

If the option 'Distribute pipeline registers' is enabled, HDL Coder tries to reposition the registers to achieve the best clock frequency.

In addition to moving the input and output pipeline registers, HDL Coder also tries to move the registers modeled internally in the design using persistent variables or with system objects like dsp.Delay.

Additional opportunities for improvements become available if you unroll loops. The 'Unroll Loops' option unrolls explicit for-loops in MATLAB code in addition to implicit for-loops that are inferred for vector and matrix operations. 'Unroll Loops' is necessary for this example to do distributed pipelining.

```
hdlcfg.InputPipeline = 1;
hdlcfg.OutputPipeline = 20;
hdlcfg.DistributedPipelining = true;
hdlcfg.LoopOptimization = 'UnrollLoops';
```

Examine the Synthesis Results

If you have ISE installed on your machine, run the logic synthesis step

```
hdlcfg.SynthesizeGeneratedCode = true;
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_fir
```

View the result report

```
edit codegen/mlhdlc_fir/hdlsrc/ise_prj/mlhdlc_fir_fixpt_syn_results.txt
```

In the synthesis report, note the clock frequency reported by the synthesis tool. When you synthesize the design with the loop unrolling and distributed pipelining options enabled, you see a significant clock frequency increase with pipelining options turned on.

Clean Up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_fir'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

Map Matrices to Block RAMs to Reduce Area

This example shows how to use the RAM mapping optimization in HDL Coder™ to map persistent matrix variables to block RAMs in hardware.

Introduction

One of the attractive features of writing MATLAB code is the ease of creating, accessing, modifying and manipulating matrices in MATLAB.

When processing such MATLAB code, HDL Coder maps these matrices to wires or registers in HDL. For example, local temporary matrix variables are mapped to wires, whereas persistent matrix variables are mapped to registers.

The latter tends to be an inefficient mapping when the matrix size is large, since the number of register resources available is limited. It also complicates synthesis, placement and routing.

Modern FPGAs feature block RAMs that are designed to have large matrices. HDL Coder takes advantage of this feature and automatically maps matrices to block RAMs to improve area efficiency. For certain designs, mapping these persistent matrices to RAMs is mandatory if the design is to be realized. State-of-the-art synthesis tools may not be able to synthesize designs when large matrices are mapped to registers, whereas the problem size is more manageable when the same matrices are mapped to RAMs.

MATLAB Design

```
design_name = 'mlhdlc_sobel';
testbench_name = 'mlhdlc_sobel_tb';
```

- MATLAB Design: mlhdlc_sobel
- MATLAB Testbench: mlhdlc_sobel_tb
- Input Image: stop_sign

Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the example files into a temporary folder.

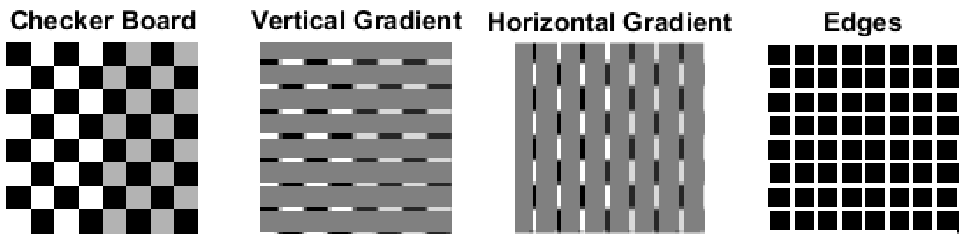
```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_sobel'];
```

```
% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

% copy the design files to the temporary directory
copyfile(fullfile(mlhdlc_demo_dir, [design_name,'.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name,'.m*']), mlhdlc_temp_dir);
```

Simulate the Design

Simulate the design with the test bench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_sobel_tb
```



Create a New HDL Coder™ Project

Run the following command to create a new project.

```
coder -hdlcoder -new mlhdlc_ram
```

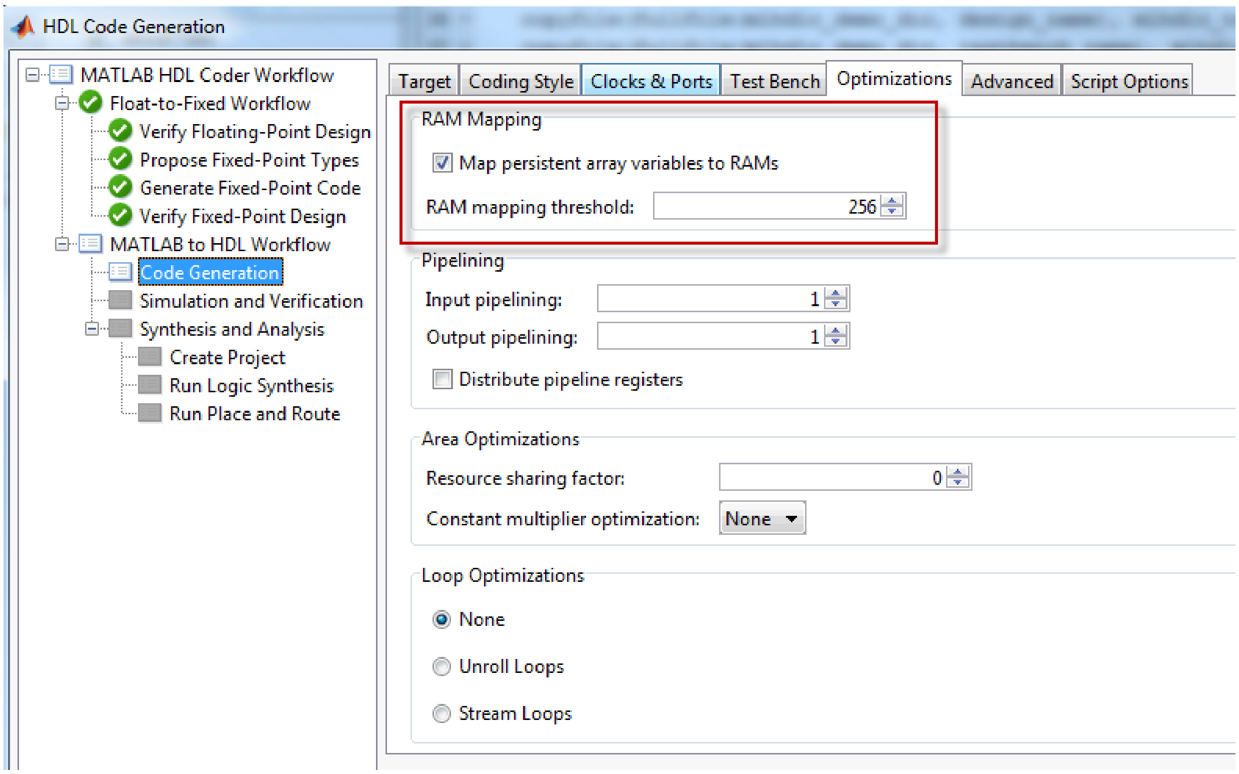
Next, add the file 'mlhdlc_sobel.m' to the project as the MATLAB function, and 'mlhdlc_sobel_tb.m' as the MATLAB test bench.

You can refer to Getting Started with MATLAB to HDL Workflow tutorial for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Turn On the RAM Mapping Optimization

Launch the Workflow Advisor.

The checkbox 'Map persistent array variables to RAMs' needs to be turned on to map persistent variables to block RAMs in the generated code.



RAM Mapping

[✓] Map persistent array variables to RAMs

RAM mapping threshold: 256

Pipelining

Input pipelining: 1

Output pipelining: 1

[ ] Distribute pipeline registers

Area Optimizations

Resource sharing factor: 0

Constant multiplier optimization: None

Loop Optimizations

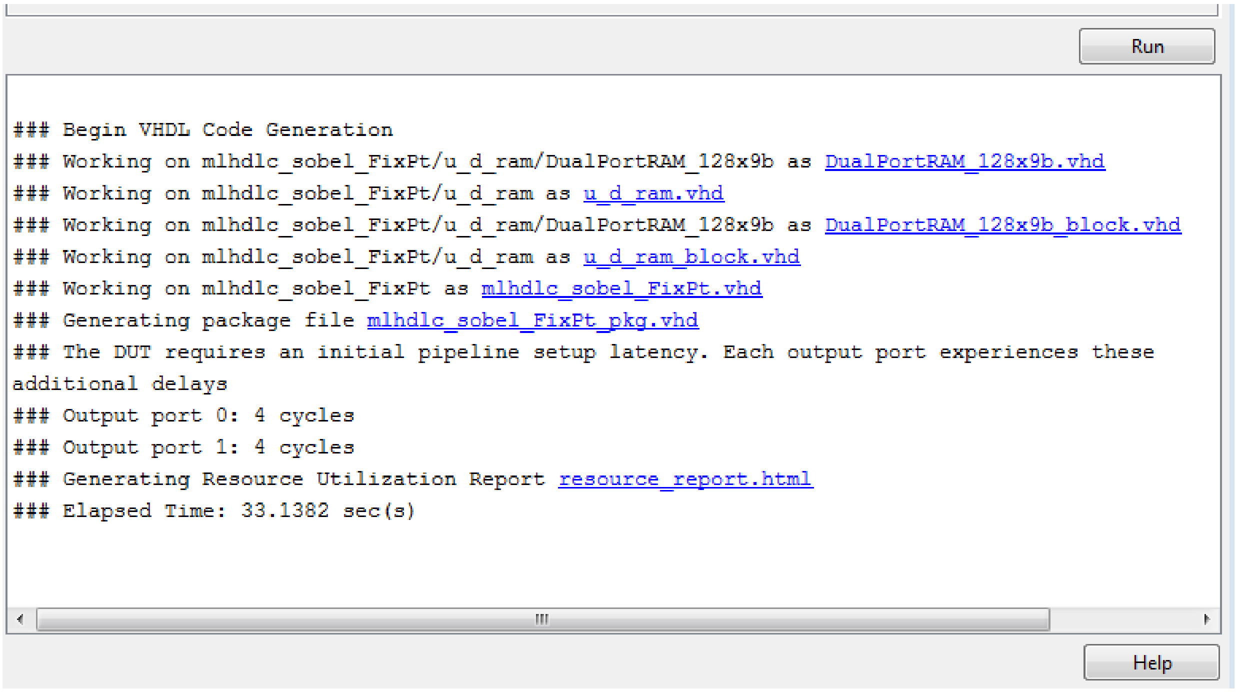
[✓ None] Unroll Loops Stream Loops

Run Fixed-Point Conversion and HDL Code Generation

In the Workflow Advisor, right-click the 'Code Generation' step. Choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

Examine the Generated Code

Examine the messages in the log window to see the RAM files generated along with the design.

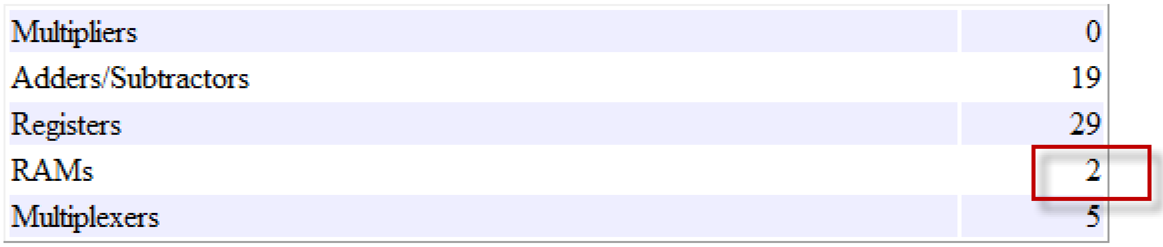


```
### Begin VHDL Code Generation
### Working on mlhdlc_sobel_FixPt/u_d_ram/DualPortRAM_128x9b as DualPortRAM_128x9b.vhd
### Working on mlhdlc_sobel_FixPt/u_d_ram as u_d_ram.vhd
### Working on mlhdlc_sobel_FixPt/u_d_ram/DualPortRAM_128x9b as DualPortRAM_128x9b_block.vhd
### Working on mlhdlc_sobel_FixPt/u_d_ram as u_d_ram_block.vhd
### Working on mlhdlc_sobel_FixPt as mlhdlc_sobel_FixPt.vhd
### Generating package file mlhdlc_sobel_FixPt_pkg.vhd
### The DUT requires an initial pipeline setup latency. Each output port experiences these
additional delays
### Output port 0: 4 cycles
### Output port 1: 4 cycles
### Generating Resource Utilization Report resource_report.html
### Elapsed Time: 33.1382 sec(s)
```

A warning message appears for each persistent matrix variable not mapped to RAM.

Examine the Resource Report

Take a look at the generated resource report, which shows the number of RAMs inferred, by following the 'Resource Utilization report...' link in the generated code window.



Multipliers	0
Adders/Subtractors	19
Registers	29
RAMs	2
Multiplexers	5

Additional Notes on RAM Mapping

- Persistent matrix variable accesses must be in unconditional regions, i.e., outside any if-else, switch case, or for-loop code.
- MATLAB functions can have any number of RAM matrices.
- All matrix variables in MATLAB that are declared persistent and meet the threshold criteria get mapped to RAMs.
- A warning is shown when a persistent matrix does not get mapped to RAM.
- Read-dependent write data cycles are not allowed: you cannot compute the write data as a function of the data read from the matrix.
- Persistent matrices cannot be copied as a whole or accessed as a sub matrix: matrix access (read/write) is allowed only on single elements of the matrix.
- Mapping persistent matrices with non-zero initial values to RAMs is not supported.

Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_sobel'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

Resource Sharing of Multipliers to Reduce Area

This example shows how to use the resource sharing optimization in HDL Coder™. This optimization identifies functionally equivalent multiplier operations in MATLAB® code and shares them in order to optimize design area. You have control over the number of multipliers to be shared in the design.

Introduction

Resource sharing is a design-wide optimization supported by HDL Coder™ for implementing area-efficient hardware.

This optimization enables users to share hardware resources by mapping 'N' functionally-equivalent MATLAB operators, in this case multipliers, to a single operator.

The user specifies 'N' using the 'Resource Sharing Factor' option in the optimization panel.

Consider the following example model of a symmetric FIR filter. It contains 4 product blocks that are functionally equivalent and which are mapped to 4 multipliers in hardware. The Resource Utilization Report shows the number of multipliers inferred from the design.

In this example you will run fixed-point conversion on the MATLAB design 'mlhdlc_sharing' followed by HDL Coder. This prerequisite step normalizes all the multipliers used in the fixed-point code. You will input a 'proposed-type settings' during this fixed-point conversion phase.

MATLAB Design

The MATLAB code used in the example is a simple symmetric FIR filter written in MATLAB and also has a testbench that exercises the filter.

```
design_name = 'mlhdlc_sharing';
testbench_name = 'mlhdlc_sharing_tb';
```

Let us take a look at the MATLAB design.

```
type(design_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB design: Symmetric FIR Filter
```

```
%
% Key Design pattern covered in this example:
% (1) Filter states represented using the persistent variables
% (2) Filter coefficients passed in as parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%   Copyright 2011-2015 The MathWorks, Inc.

%#codegen
function [y_out, x_out] = mlhdlc_sharing(x_in, h)
% Symmetric FIR Filter

persistent ud1 ud2 ud3 ud4 ud5 ud6 ud7 ud8;
if isempty(ud1)
    ud1 = 0; ud2 = 0; ud3 = 0; ud4 = 0; ud5 = 0; ud6 = 0; ud7 = 0; ud8 = 0;
end

x_out = ud8;


a1 = ud1 + ud8;
a2 = ud2 + ud7;
a3 = ud3 + ud6;
a4 = ud4 + ud5;

% filtered output
y_out = (h(1) * a1 + h(2) * a2) + (h(3) * a3 + h(4) * a4);


% update the delay line
ud8 = ud7;
ud7 = ud6;
ud6 = ud5;
ud5 = ud4;
ud4 = ud3;
ud3 = ud2;
ud2 = ud1;
ud1 = x_in;

end
```

```
type(testbench_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% MATLAB test bench for the FIR filter
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%   Copyright 2011-2015 The MathWorks, Inc.

clear mlhdlc_sharing;

% input signal with noise
x_in = cos(3.*pi.*(0:0.001:2).*(1+(0:0.001:2).*75)).';

len = length(x_in);
y_out = zeros(1,len);
x_out = zeros(1,len);

% Define a regular MATLAB constant array:
%
% filter coefficients
h = [-0.1339 -0.0838 0.2026 0.4064];

for ii=1:len
    data = x_in(ii);
    % call to the design 'mlhdlc_sfir' that is targeted for hardware
    [y_out(ii), x_out(ii)] = mlhdlc_sharing(data, h);
end

figure('Name', [mfilename, '_plot']);
plot(1:len,y_out);
```

Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_sfir_sharing'];

% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name,'.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name,'.m*']), mlhdlc_temp_dir);
```

Create a New HDL Coder Project

Run the following command to create a new project:

```
coder -hdlcoder -new mlhdlc_sfir_sharing
```

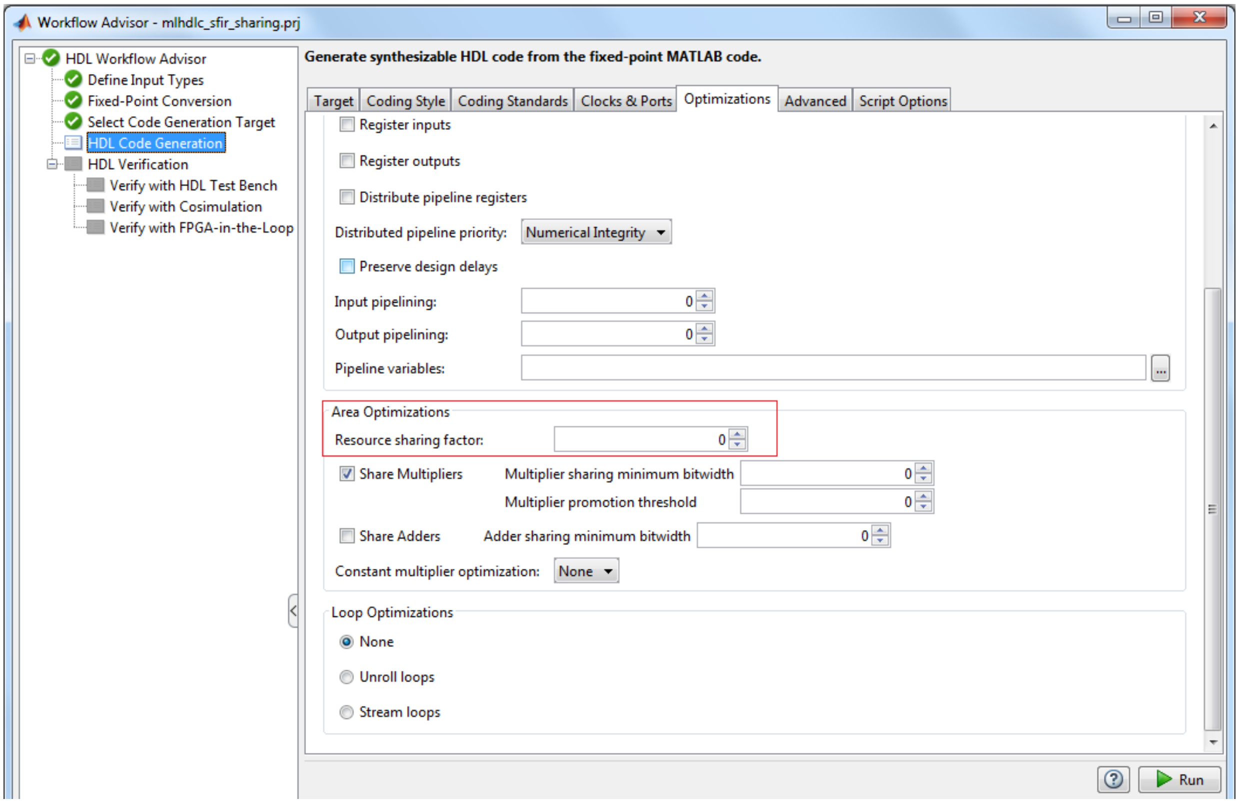
Next, add the file 'mlhdlc_sharing.m' to the project as the MATLAB Function and 'mlhdlc_sharing_tb.m' as the MATLAB Test Bench.

You can refer to Getting Started with MATLAB to HDL Workflow tutorial for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Realize an N-to-1 Mapping of Multipliers

Turn on the resource sharing optimization by setting the 'Resource Sharing Factor' to a positive integer value.

This parameter specifies 'N' in the N-to-1 hardware mapping. Choose a value of N > 1.


Workflow Advisor - mlhdlc_sfir_sharing.prj

- HDL Workflow Advisor
  - Define Input Types
  - Fixed-Point Conversion
  - Select Code Generation Target
  - HDL Code Generation
  - HDL Verification
    - Verify with HDL Test Bench
    - Verify with Cosimulation
    - Verify with FPGA-in-the-Loop

Generate synthesizable HDL code from the fixed-point MATLAB code.

Target | Coding Style | Coding Standards | Clocks & Ports | Optimizations | Advanced | Script Options

- [ ] Register inputs
- [ ] Register outputs
- [ ] Distribute pipeline registers

Distributed pipeline priority: Numerical Integrity

- [ ] Preserve design delays

Input pipelining: 0

Output pipelining: 0

Pipeline variables:

Area Optimizations

Resource sharing factor: 0

- [x] Share Multipliers — Multiplier sharing minimum bitwidth: 0; Multiplier promotion threshold: 0
- [ ] Share Adders — Adder sharing minimum bitwidth: 0

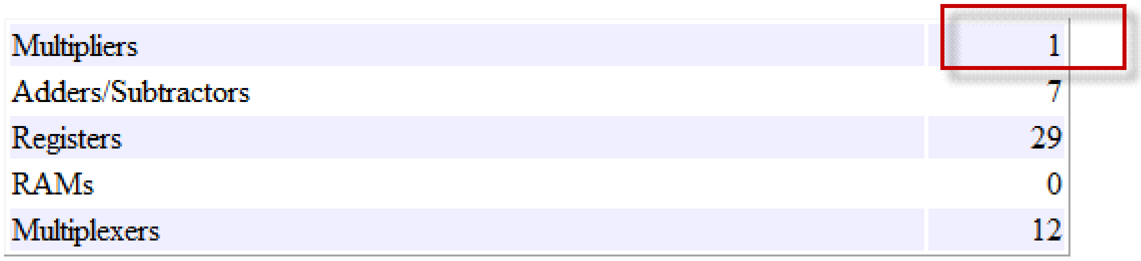
Constant multiplier optimization: None

Loop Optimizations

- ● None
- ○ Unroll loops
- ○ Stream loops

Examine the Resource Report

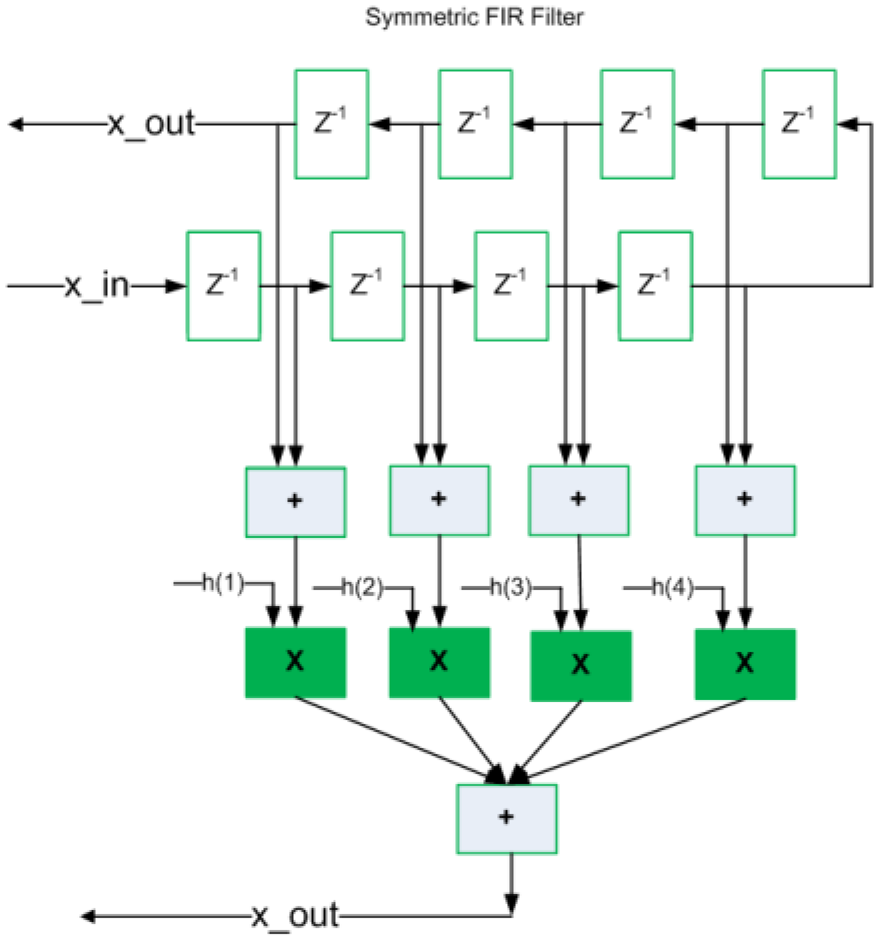
There are 4 multiplication operators in this example design. Generating HDL with a 'SharingFactor' of 4 will result in only one multiplier in the generated code.



Multipliers	1
Adders/Subtractors	7
Registers	29
RAMs	0
Multiplexers	12

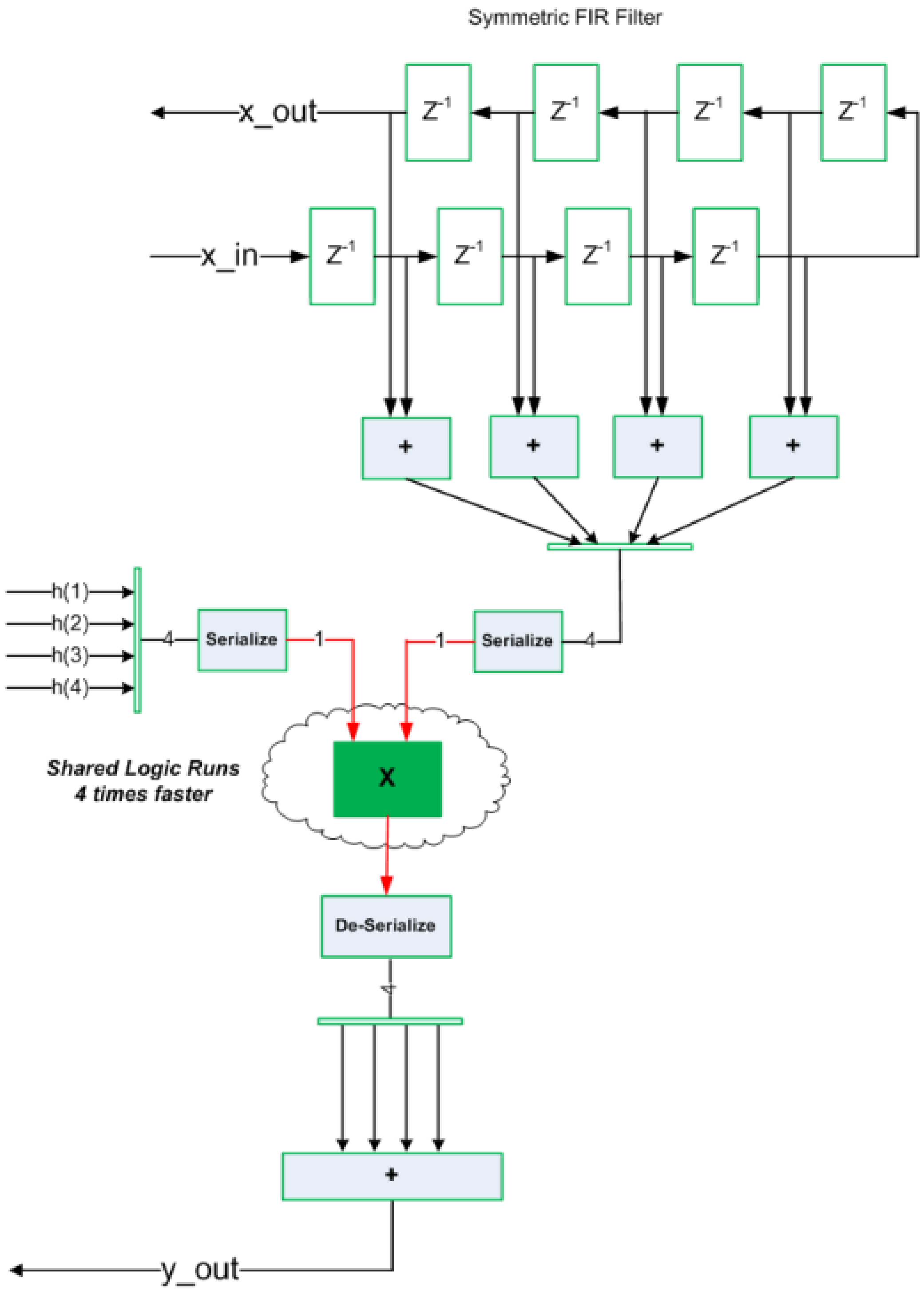
Sharing Architecture

The following figure shows how the algorithm is implemented in hardware when we synthesize the generated code without turning on the sharing optimization.



The following figure shows the sharing architecture automatically implemented by HDL Coder when the sharing optimization option is turned on.

The inputs to the shared multiplier are time-multiplexed at a faster rate (in this case 4x faster and denoted in red). The outputs are then routed to the respective consumers at a slower rate (in green).


Symmetric FIR Filter
x_out
x_in
z^{-1}
+
h(1)
h(2)
h(3)
h(4)
4
Serialize
1
Shared Logic Runs 4 times faster
X
De-Serialize
y_out

Run Fixed-Point Conversion and HDL Code Generation

Launch the Workflow Advisor and right-click the 'Code Generation' step. Choose the option 'Run to selected task' to run all the steps from the beginning through the HDL code generation.

The detailed example Fixed-point conversion derived ranges provides a tutorial for updating the type proposal settings during fixed-point conversion.

Note that to share multipliers of different word-length, in the Target and Optimizations -> Resource Sharing tab of HDL Configuration Parameters, specify the 'Multiplier promotion threshold'. For more information, see the Resource Sharing Documentation.

Run Synthesis and Examine Synthesis Results

Synthesize the generated code from the design with this optimization turned off, then with it turned on, and examine the area numbers in the resource report.

Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_sfir_sharing'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

Loop Streaming to Reduce Area

This example shows how to use the design-level loop streaming optimization in HDL Coder™ to optimize area.

Introduction

A MATLAB® for loop generates a FOR_GENERATE loop in VHDL. Such loops are always spatially unrolled for execution in hardware. In other words, the body of the software loop is replicated as many times in hardware as the number of loop iterations. This results in inefficient area usage.

The loop streaming optimization creates an alternative implementation of a software loop, where the body of the loop is shared in hardware. Instead of spatially replicating copies of the loop body, HDL Coder™ creates a single hardware instance of the loop body that is time-multiplexed across loop iterations.

MATLAB Design

The MATLAB code used in this example implements a simple FIR filter. This example also shows a MATLAB testbench that exercises the filter.

```
design_name = 'mlhdlc_fir';
testbench_name = 'mlhdlc_fir_tb';
```

1 Design: mlhdlc_fir
2 Test Bench: mlhdlc_fir_tb

Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

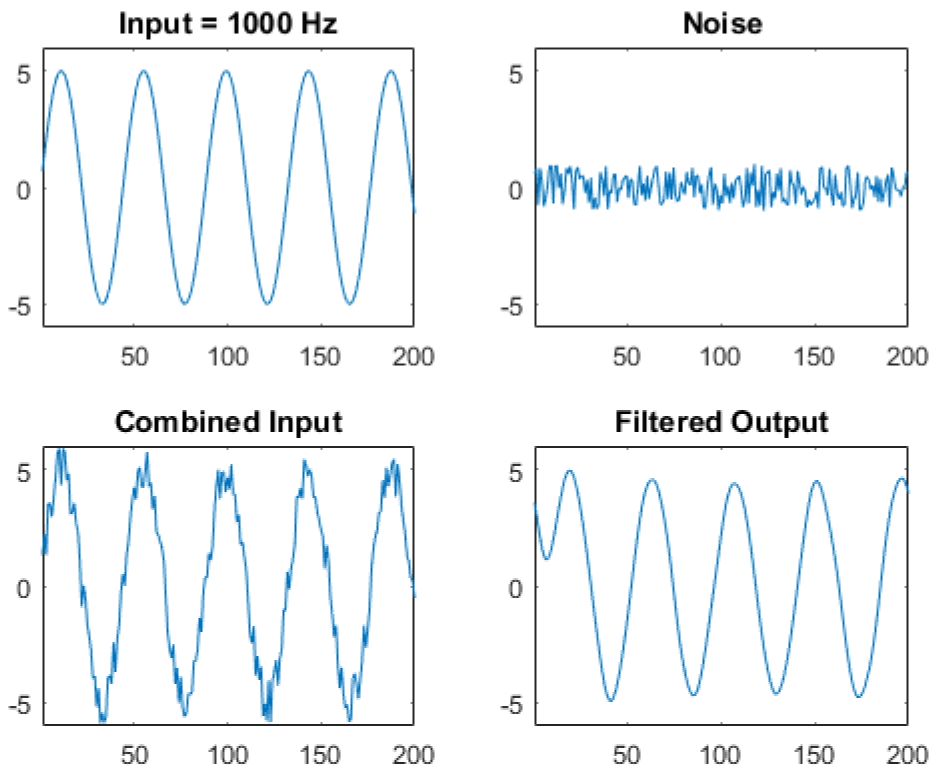
```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_fir'];

% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);
```

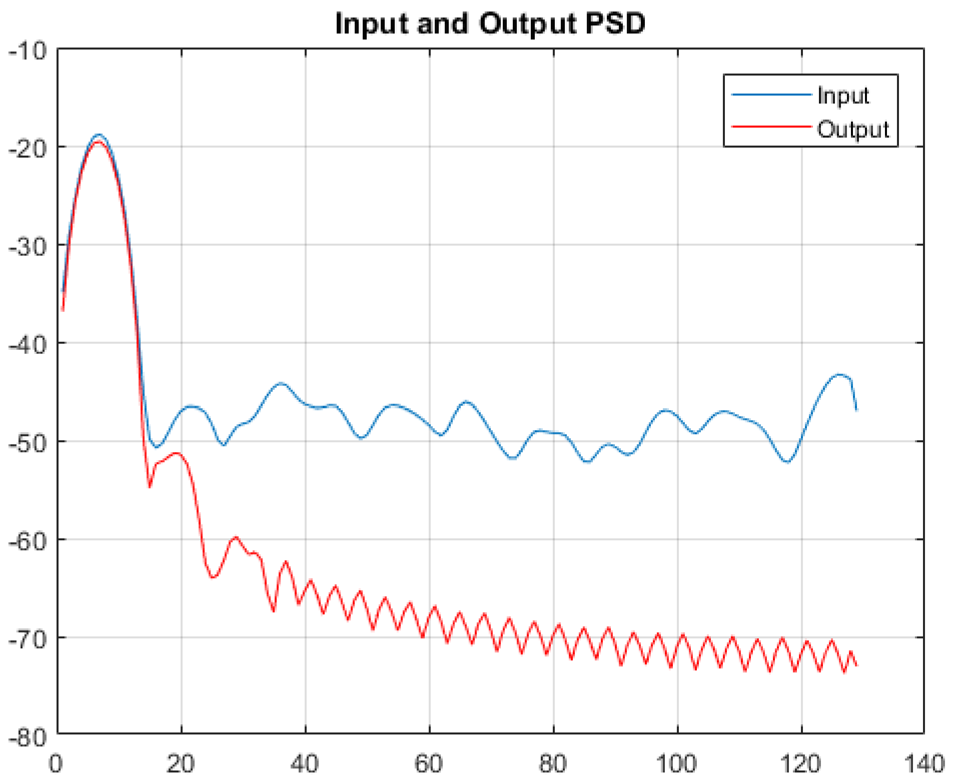
```
copyfile(fullfile(mlhdlc_demo_dir, [design_name,'.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name,'.m*']), mlhdlc_temp_dir);
```

Simulate the Design

Simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_fir_tb
```





Creating a New Project From the Command Line

To create a new project, enter the following command:

```
coder -hdlcoder -new fir_project
```

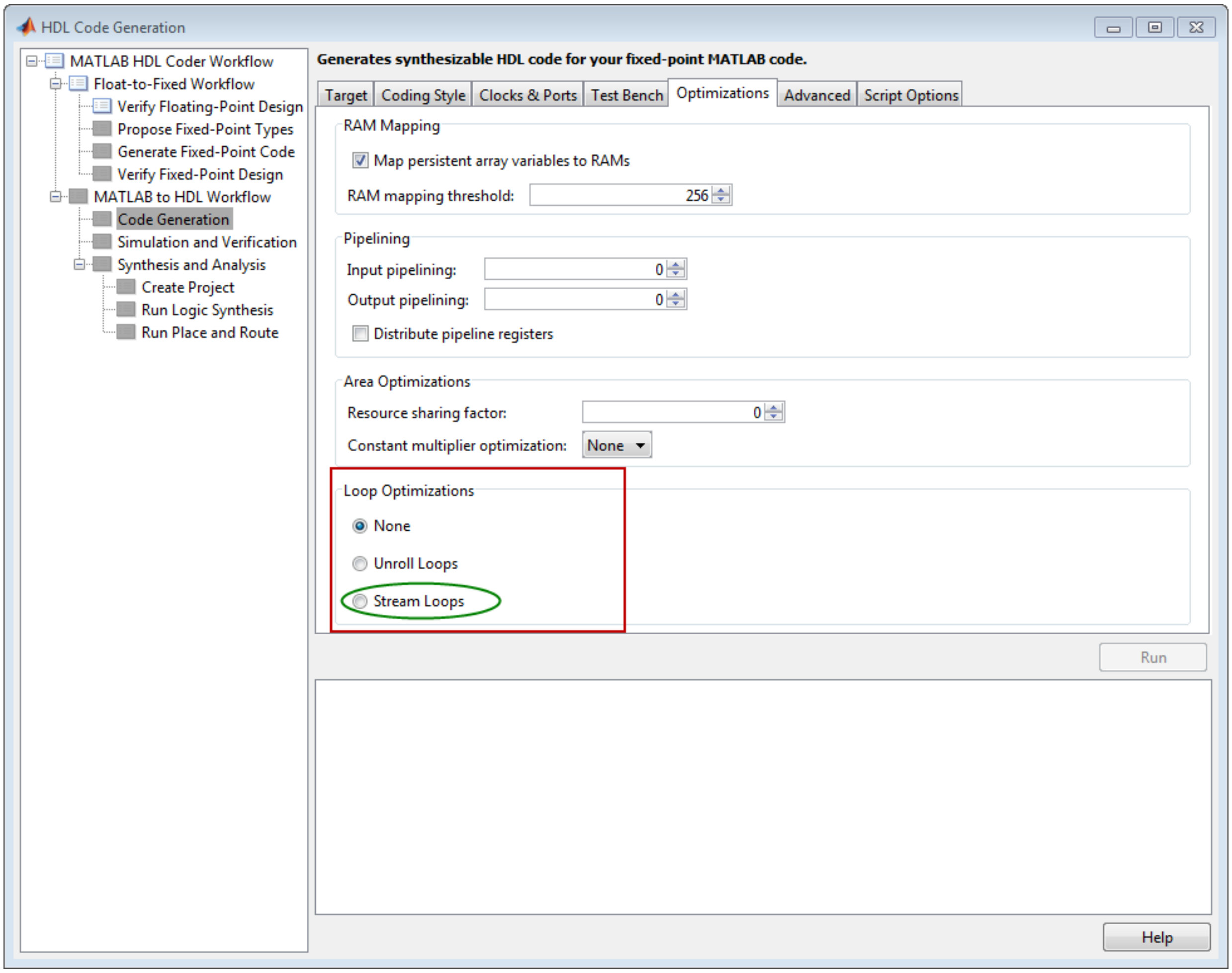
Next, add the file 'mlhdlc_fir.m' to the project as the MATLAB Function and 'mlhdlc_fir_tb.m' as the MATLAB Test Bench.

Launch the Workflow Advisor.

You can refer to Getting Started with MATLAB to HDL Workflow tutorial for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Turn On Loop Streaming

The loop streaming optimization in HDL Coder converts software loops (either written explicitly using a for-loop statement, or inferred loops from matrix/vector operators) to area-friendly hardware loops.


HDL Code Generation

- MATLAB HDL Coder Workflow
  - Float-to-Fixed Workflow
    - Verify Floating-Point Design
    - Propose Fixed-Point Types
    - Generate Fixed-Point Code
    - Verify Fixed-Point Design
  - MATLAB to HDL Workflow
    - Code Generation
    - Simulation and Verification
    - Synthesis and Analysis
      - Create Project
      - Run Logic Synthesis
      - Run Place and Route

Generates synthesizable HDL code for your fixed-point MATLAB code.

Target | Coding Style | Clocks & Ports | Test Bench | Optimizations | Advanced | Script Options

RAM Mapping

[✓] Map persistent array variables to RAMs

RAM mapping threshold: 256

Pipelining

Input pipelining: 0

Output pipelining: 0

[ ] Distribute pipeline registers

Area Optimizations

Resource sharing factor: 0

Constant multiplier optimization: None

Loop Optimizations

(●) None

( ) Unroll Loops

( ) Stream Loops

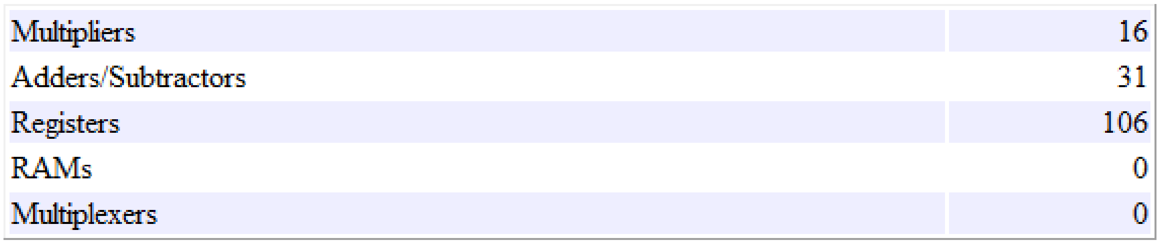
Run Fixed-Point Conversion and HDL Code Generation

Right-click the 'Code Generation' step. Choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

Examine the Generated Code

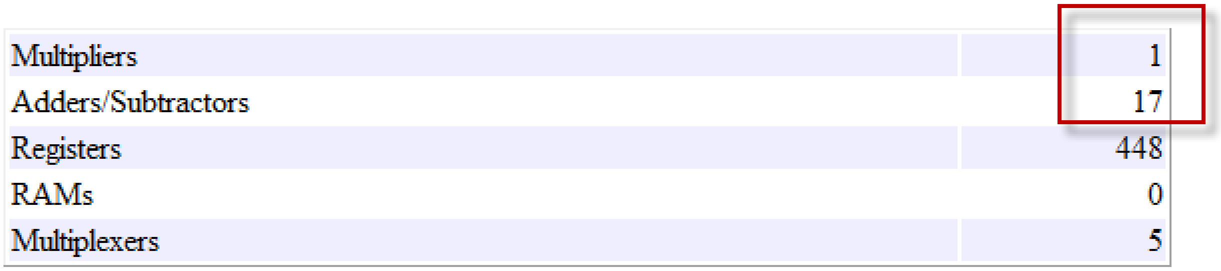
When you synthesize the design with the loop streaming optimization, you see a reduction in area resources in the resource report. Try generating HDL code with and without the optimization.

The resource report without the loop streaming optimization:



Multipliers	16
Adders/Subtractors	31
Registers	106
RAMs	0
Multiplexers	0

The resource report with the loop streaming optimization enabled:



Multipliers	1
Adders/Subtractors	17
Registers	448
RAMs	0
Multiplexers	5

Known Limitations

Loops will be streamed only if they are regular nested loops. A regular nested loop structure is defined as one where:

- None of the loops in any level of nesting appear in a conditional flow region, i.e. no loop can be embedded within if-else or switch-else regions.
- Loop index variables are monotonically increasing.
- Total number of iterations of the loop structure is non-zero.

- There are no back-to-back loops at the same level of the nesting hierarchy.

Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_fir'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

Constant Multiplier Optimization to Reduce Area

This example shows how to perform a design-level area optimization in HDL Coder by converting constant multipliers into shifts and adds using canonical signed digit (CSD) techniques.

Introduction

This tutorial shows how the use of canonical signed digit (CSD) representation of multiplier constants (for example, in gain coefficients or filter coefficients) can significantly reduce the area of the hardware implementation.

Canonical Signed Digit (CSD) Representation

A signed digit (SD) representation is an augmented binary representation with weights 0,1 and -1.

$$X_{10} = \sum_{r=0}^{B-1} x_r \cdot 2^r$$

where

$$x_r = 0, 1, -1(\bar{1})$$

For example, here are a couple of signed digit representations for 93:

$$X_{10} = 64 + 16 + 13 = 01011101$$

$$X_{10} = 128 - 32 - 2 - 1 = 10\bar{1}000\bar{1}\bar{1}$$

Note that the signed digit representation is non-unique. A canonical signed digit (CSD) representation is an SD representation with the minimum number of non-zero elements.

Here are some properties of CSD numbers:

1. No two consecutive bits in a CSD number are non-zero
2. CSD representation is guaranteed to have minimum number of non-zero bits

3 CSD representation of a number is unique

CSD Multiplier

Let us see how a CSD representation can yield an implementation requiring a minimum number of adders.

Let us look at CSD example:

```
y = 231 * x
  = (11100111) * x                   % 231 in binary form
  = (1001'01001') * x                % 231 in signed digit form
  = (256 - 32 + 8 - 1) * x           %
  = (x << 8) - (x << 5) + (x << 3) -x % cost of CSD: 3 Adders
```

FCSD Multiplier

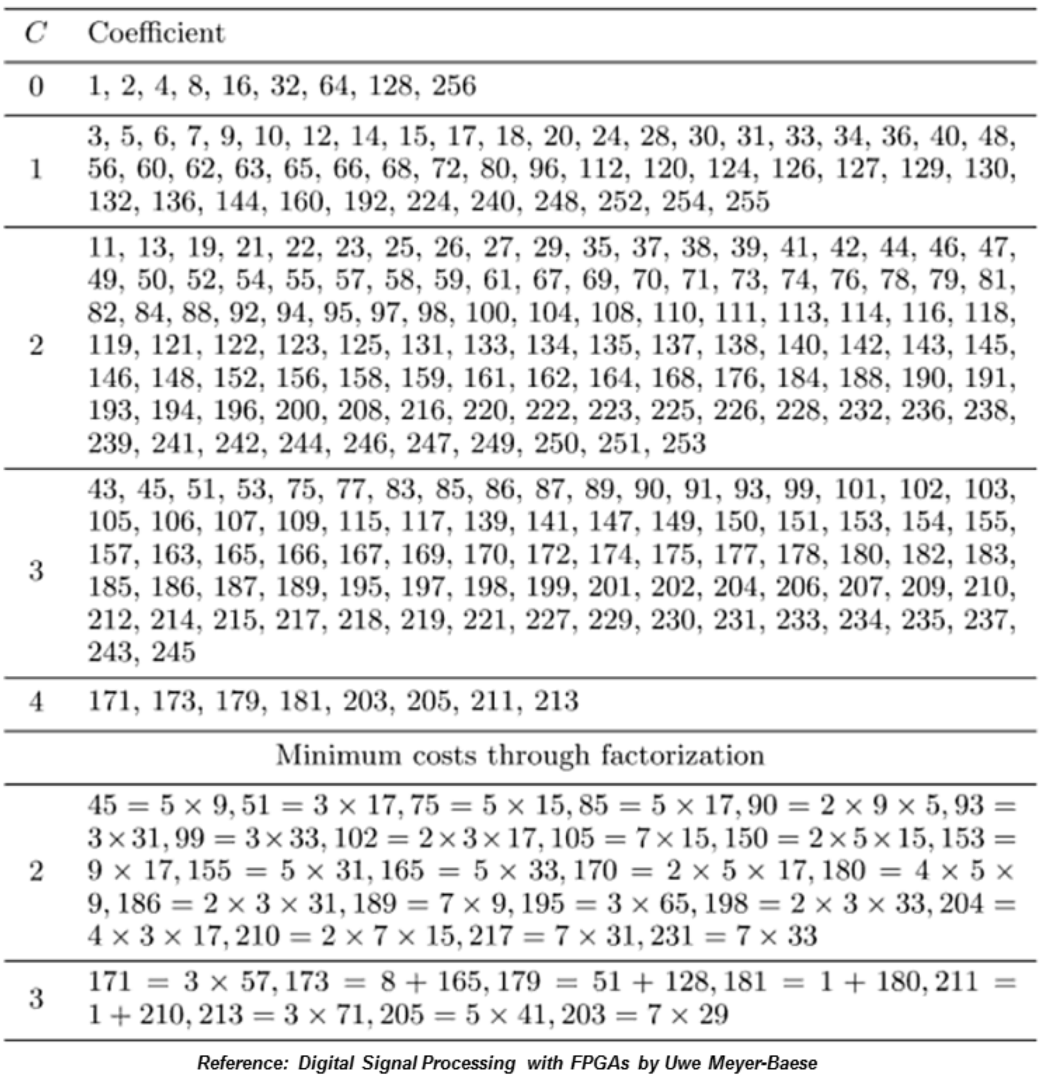
A combination of factorization and CSD representation of a constant multiplier can lead to further reduction in hardware cost (number of adders).

FCSD can further reduce the number of adders in the above constant multiplier:

```
y  = 231 * x
y  = (7 * 33) * x
y_tmp = (x << 5) + x
y  = (y_tmp << 3) - y_tmp          % cost of FCSD: 2 Adders
```

CSD/FCSD Costs

This table shows the costs (C) of all 8-bit multipliers.



C	Coefficient
0	1, 2, 4, 8, 16, 32, 64, 128, 256
1	3, 5, 6, 7, 9, 10, 12, 14, 15, 17, 18, 20, 24, 28, 30, 31, 33, 34, 36, 40, 48, 56, 60, 62, 63, 65, 66, 68, 72, 80, 96, 112, 120, 124, 126, 127, 129, 130, 132, 136, 144, 160, 192, 224, 240, 248, 252, 254, 255
2	11, 13, 19, 21, 22, 23, 25, 26, 27, 29, 35, 37, 38, 39, 41, 42, 44, 46, 47, 49, 50, 52, 54, 55, 57, 58, 59, 61, 67, 69, 70, 71, 73, 74, 76, 78, 79, 81, 82, 84, 88, 92, 94, 95, 97, 98, 100, 104, 108, 110, 111, 113, 114, 116, 118, 119, 121, 122, 123, 125, 131, 133, 134, 135, 137, 138, 140, 142, 143, 145, 146, 148, 152, 156, 158, 159, 161, 162, 164, 168, 176, 184, 188, 190, 191, 193, 194, 196, 200, 208, 216, 220, 222, 223, 225, 226, 228, 232, 236, 238, 239, 241, 242, 244, 246, 247, 249, 250, 251, 253
3	43, 45, 51, 53, 75, 77, 83, 85, 86, 87, 89, 90, 91, 93, 99, 101, 102, 103, 105, 106, 107, 109, 115, 117, 139, 141, 147, 149, 150, 151, 153, 154, 155, 157, 163, 165, 166, 167, 169, 170, 172, 174, 175, 177, 178, 180, 182, 183, 185, 186, 187, 189, 195, 197, 198, 199, 201, 202, 204, 206, 207, 209, 210, 212, 214, 215, 217, 218, 219, 221, 227, 229, 230, 231, 233, 234, 235, 237, 243, 245
4	171, 173, 179, 181, 203, 205, 211, 213
	Minimum costs through factorization
2	$45 = 5 \times 9, 51 = 3 \times 17, 75 = 5 \times 15, 85 = 5 \times 17, 90 = 2 \times 9 \times 5, 93 = 3 \times 31, 99 = 3 \times 33, 102 = 2 \times 3 \times 17, 105 = 7 \times 15, 150 = 2 \times 5 \times 15, 153 = 9 \times 17, 155 = 5 \times 31, 165 = 5 \times 33, 170 = 2 \times 5 \times 17, 180 = 4 \times 5 \times 9, 186 = 2 \times 3 \times 31, 189 = 7 \times 9, 195 = 3 \times 65, 198 = 2 \times 3 \times 33, 204 = 4 \times 3 \times 17, 210 = 2 \times 7 \times 15, 217 = 7 \times 31, 231 = 7 \times 33$
3	$171 = 3 \times 57, 173 = 8 + 165, 179 = 51 + 128, 181 = 1 + 180, 211 = 1 + 210, 213 = 3 \times 71, 205 = 5 \times 41, 203 = 7 \times 29$

Reference: Digital Signal Processing with FPGAs by Uwe Meyer-Baese

MATLAB® Design

The MATLAB code used in this example implements a simple FIR filter. The example also shows a MATLAB test bench that exercises the filter.

```
design_name = 'mlhdlc_csd';
testbench_name = 'mlhdlc_csd_tb';
```

1. Design: mlhdlc_csd
2. Test Bench: mlhdlc_csd_tb

Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_csd'];

% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name,'.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name,'.m*']), mlhdlc_temp_dir);
```

Simulate the Design

Simulate the design with the test bench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_csd_tb
```

Create a Fixed-Point Conversion Config Object

To perform fixed-point conversion, you need a 'fixpt' config object.

Create a 'fixpt' config object and specify your test bench name:

```
close all;
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'mlhdlc_csd_tb';
```

Create an HDL Code Generation Config Object

To generate code, you must create an 'hdl' config object and set your test bench name:

```
hdlcfg = coder.config('hdl');
hdlcfg.TestBenchName = 'mlhdlc_csd_tb';
```

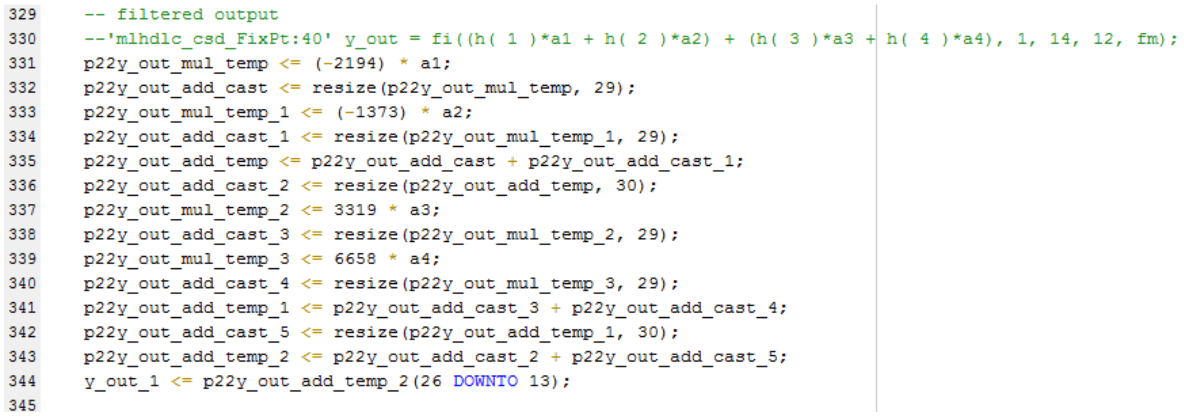
Generate Code without Constant Multiplier Optimization

```
hdlcfg.ConstantMultiplierOptimization = 'None';
```

Enable the 'Unroll Loops' option to inline multiplier constants.

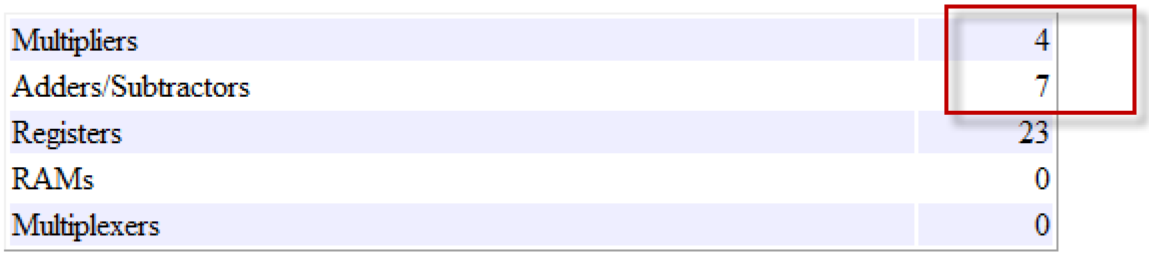
```
hdlcfg.LoopOptimization = 'UnrollLoops';
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_csd
```

Examine the generated code.



```
    -- filtered output
    --'mlhdlc_csd_FixPt:40' y_out = fi((h( 1 )*a1 + h( 2 )*a2) + (h( 3 )*a3 + h( 4 )*a4), 1, 14, 12, fm);
    p22y_out_mul_temp <= (-2194) * a1;
    p22y_out_add_cast <= resize(p22y_out_mul_temp, 29);
    p22y_out_mul_temp_1 <= (-1373) * a2;
    p22y_out_add_cast_1 <= resize(p22y_out_mul_temp_1, 29);
    p22y_out_add_temp <= p22y_out_add_cast + p22y_out_add_cast_1;
    p22y_out_add_cast_2 <= resize(p22y_out_add_temp, 30);
    p22y_out_mul_temp_2 <= 3319 * a3;
    p22y_out_add_cast_3 <= resize(p22y_out_mul_temp_2, 29);
    p22y_out_mul_temp_3 <= 6658 * a4;
    p22y_out_add_cast_4 <= resize(p22y_out_mul_temp_3, 29);
    p22y_out_add_temp_1 <= p22y_out_add_cast_3 + p22y_out_add_cast_4;
    p22y_out_add_cast_5 <= resize(p22y_out_add_temp_1, 30);
    p22y_out_add_temp_2 <= p22y_out_add_cast_2 + p22y_out_add_cast_5;
    y_out_1 <= p22y_out_add_temp_2(26 DOWNTO 13);

```

Take a look at the resource report for adder and multiplier usage without the CSD optimization.



Multipliers	4
Adders/Subtractors	7
Registers	23
RAMs	0
Multiplexers	0

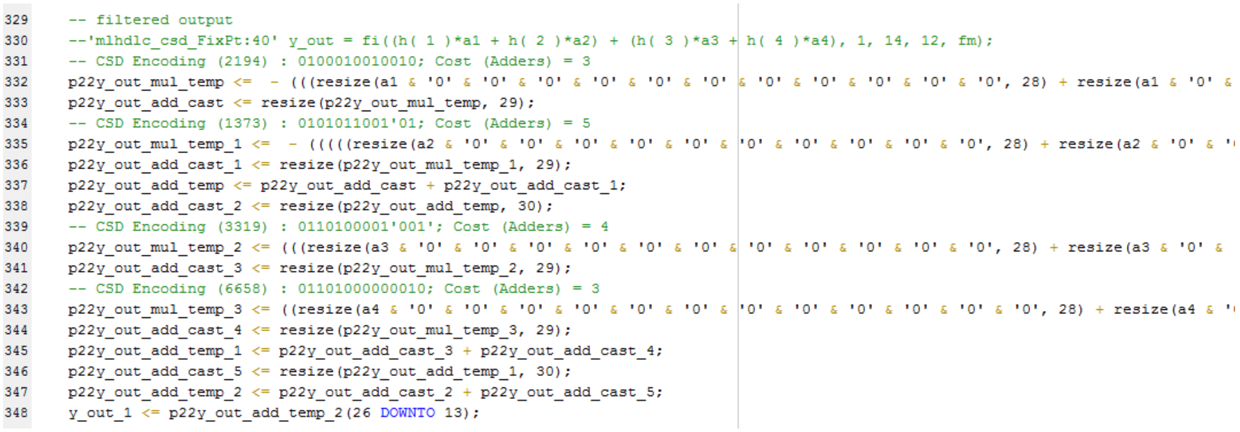
Generate Code with CSD Optimization

```
hdlcfg.ConstantMultiplierOptimization = 'CSD';
```

Enable the 'Unroll Loops' option to inline multiplier constants.

```
hdlcfg.LoopOptimization = 'UnrollLoops';
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_csd
```

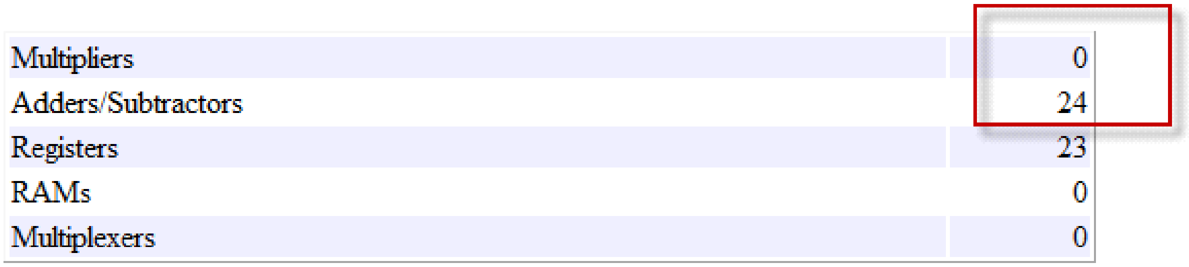
Examine the generated code.



```
-- filtered output
--'mlhdlc_csd_FixPt:40' y_out = fi((h( 1 )*a1 + h( 2 )*a2) + (h( 3 )*a3 + h( 4 )*a4), 1, 14, 12, fm);
-- CSD Encoding (2194) : 0100010010010; Cost (Adders) = 3
p22y_out_mul_temp <=  - (((resize(a1 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a1 & '0' &
p22y_out_add_cast <= resize(p22y_out_mul_temp, 29);
-- CSD Encoding (1373) : 0101011001'01; Cost (Adders) = 5
p22y_out_mul_temp_1 <=  - (((((resize(a2 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a2 & '0' &
p22y_out_add_cast_1 <= resize(p22y_out_mul_temp_1, 29);
p22y_out_add_temp <= p22y_out_add_cast + p22y_out_add_cast_1;
p22y_out_add_cast_2 <= resize(p22y_out_add_temp, 30);
-- CSD Encoding (3319) : 0110100001'001'; Cost (Adders) = 4
p22y_out_mul_temp_2 <= (((resize(a3 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a3 & '0' &
p22y_out_add_cast_3 <= resize(p22y_out_mul_temp_2, 29);
-- CSD Encoding (6658) : 01101000000010; Cost (Adders) = 3
p22y_out_mul_temp_3 <= ((resize(a4 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a4 &
p22y_out_add_cast_4 <= resize(p22y_out_mul_temp_3, 29);
p22y_out_add_temp_1 <= p22y_out_add_cast_3 + p22y_out_add_cast_4;
p22y_out_add_cast_5 <= resize(p22y_out_add_temp_1, 30);
p22y_out_add_temp_2 <= p22y_out_add_cast_2 + p22y_out_add_cast_5;
y_out_1 <= p22y_out_add_temp_2(26 DOWNTO 13);
```

Examine the code with comments that outline the CSD encoding for all the constant multipliers.

Look at the resource report and notice that with the CSD optimization, the number of multipliers is reduced to zero and multipliers are replaced by shifts and adders.



Multipliers	0
Adders/Subtractors	24
Registers	23
RAMs	0
Multiplexers	0

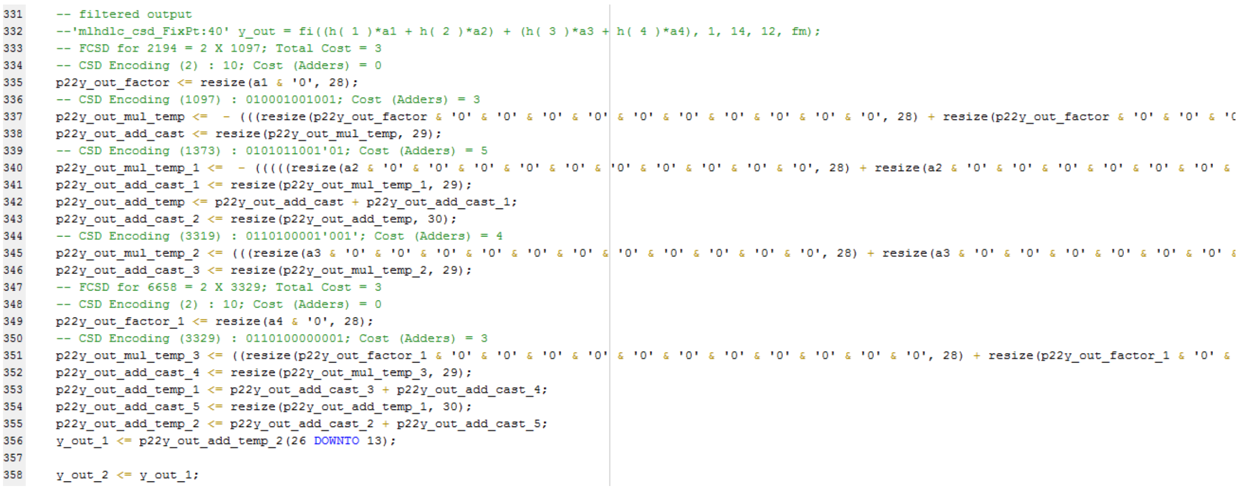
Generate Code with FCSD Optimization

```
hdlcfg.ConstantMultiplierOptimization = 'FCSD';
```

Enable the 'Unroll Loops' option to inline multiplier constants.

```
hdlcfg.LoopOptimization = 'UnrollLoops';
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_csd
```

Examine the generated code.



```
-- filtered output
--'mlhdlc_csd_FixPt:40' y_out = fi((h( 1 )*a1 + h( 2 )*a2) + (h( 3 )*a3 + h( 4 )*a4), 1, 14, 12, fm);
-- FCSD for 2194 = 2 X 1097; Total Cost = 3
-- CSD Encoding (2) : 10; Cost (Adders) = 0
p22y_out_factor <= resize(a1 & '0', 28);
-- CSD Encoding (1097) : 010001001001; Cost (Adders) = 3
p22y_out_mul_temp <=  - (((resize(p22y_out_factor & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(p22y_out_factor & '0' & '0' & 'C
p22y_out_add_cast <= resize(p22y_out_mul_temp, 29);
-- CSD Encoding (1373) : 0101011001'01; Cost (Adders) = 5
p22y_out_mul_temp_1 <=  - ((((resize(a2 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a2 & '0' & '0' & '0' & '0' & '0' & '0' &
p22y_out_add_cast_1 <= resize(p22y_out_mul_temp_1, 29);
p22y_out_add_temp <= p22y_out_add_cast + p22y_out_add_cast_1;
p22y_out_add_cast_2 <= resize(p22y_out_add_temp, 30);
-- CSD Encoding (3319) : 0110100001'001'; Cost (Adders) = 4
p22y_out_mul_temp_2 <= (((resize(a3 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a3 & '0' & '0' & '0' & '0' & '0' & '0' &
p22y_out_add_cast_3 <= resize(p22y_out_mul_temp_2, 29);
-- FCSD for 6658 = 2 X 3329; Total Cost = 3
-- CSD Encoding (2) : 10; Cost (Adders) = 0
p22y_out_factor_1 <= resize(a4 & '0', 28);
-- CSD Encoding (3329) : 0110100000001; Cost (Adders) = 3
p22y_out_mul_temp_3 <= ((resize(p22y_out_factor_1 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(p22y_out_factor_1 & '0' &
p22y_out_add_cast_4 <= resize(p22y_out_mul_temp_3, 29);
p22y_out_add_temp_1 <= p22y_out_add_cast_3 + p22y_out_add_cast_4;
p22y_out_add_cast_5 <= resize(p22y_out_add_temp_1, 30);
p22y_out_add_temp_2 <= p22y_out_add_cast_2 + p22y_out_add_cast_5;
y_out_1 <= p22y_out_add_temp_2(26 DOWNTO 13);

y_out_2 <= y_out_1;
```

Examine the code with comments that outline the FCSD encoding for all the constant multipliers. In this particular example, the generated code is identical in terms of area resources for the multiplier constants. However, take a look at the factorizations of the constants in the generated code.

If you choose the 'Auto' option, HDL Coder will automatically choose between the CSD and FCSD options for the best result.

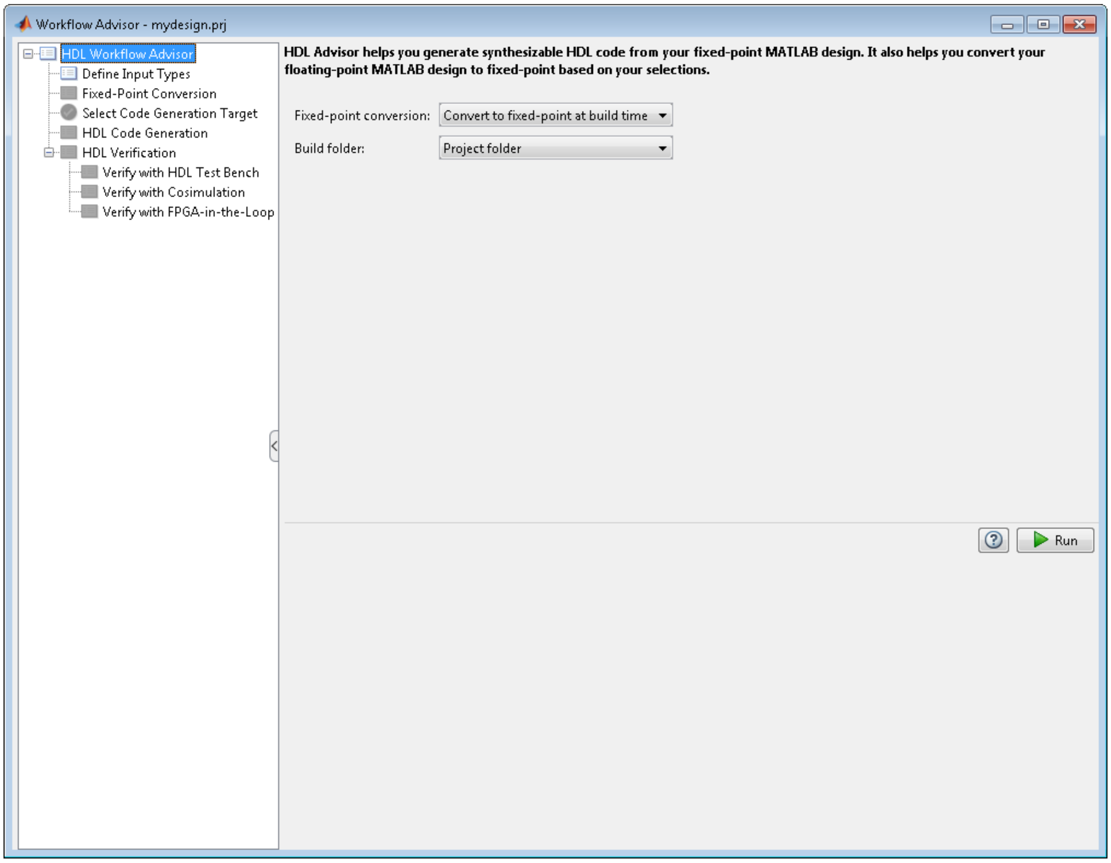
Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabh
mlhdlc_temp_dir = [tempdir 'mlhdlc_csd'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

HDL Workflow Advisor Reference

HDL Workflow Advisor



HDL Advisor helps you generate synthesizable HDL code from your fixed-point MATLAB design. It also helps you convert your floating-point MATLAB design to fixed-point based on your selections.

Fixed-point conversion: Convert to fixed-point at build time

Build folder: Project folder

Overview

The HDL Workflow Advisor is a tool that supports a suite of tasks covering the stages of the ASIC and FPGA design process, including converting floating-point MATLAB algorithms to fixed-point algorithms. Some tasks perform code validation or checking; others run the HDL code generator or third-party tools. Each folder at the top level of the HDL Workflow Advisor contains a group of related tasks that you can select and run.

Use the HDL Workflow Advisor to:

- Convert floating-point MATLAB algorithms to fixed-point algorithms.

  If you already have a fixed-point MATLAB algorithm, set **Design needs conversion to Fixed Point?** to `No` to skip this step.
- Generate HDL code from fixed-point MATLAB algorithms.
- Simulate the HDL code using a third-party simulation tool.
- Synthesize the HDL code and run a mapping process that maps the synthesized logic design to the target FPGA.
- Run a Place and Route process that takes the circuit description produced by the previous mapping process, and emits a circuit description suitable for programming an FPGA.

Procedures

Automatically Run Tasks

To automatically run the tasks within a folder:

1. Click the **Run** button. The tasks run in order until a task fails.

   Alternatively, right-click the folder to open the context menu. From the context menu, select `Run` to run the tasks within the folder.
2. If a task in the folder fails:

   a. Fix the failure using the information in the results pane.

   b. Continue the run by clicking the **Run** button.

Run Individual Tasks

To run an individual task:

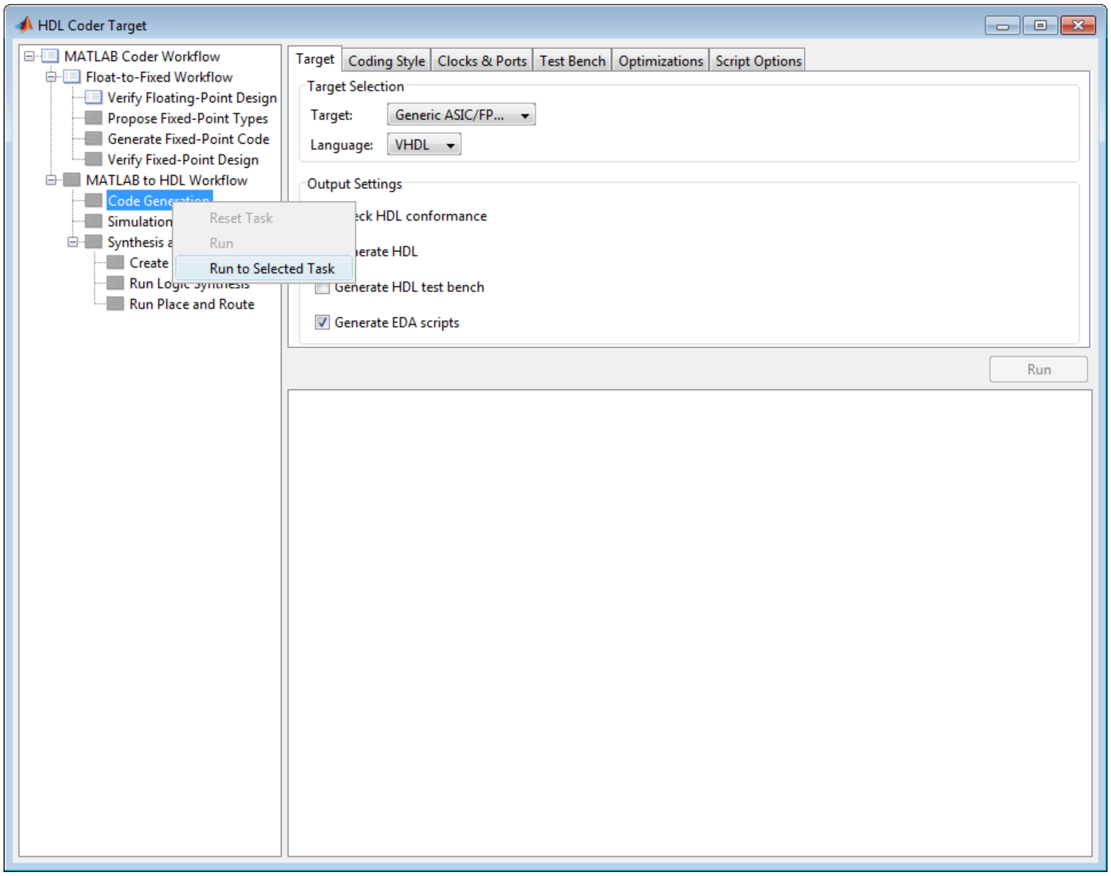
1. Click the **Run** button.

   Alternatively, right-click the task to open the context menu. From the context menu, select `Run` to run the selected task.
2. Review Results. The possible results are:
   Pass: Move on to the next task.
   Warning: Review results, decide whether to move on or fix.
   Fail: Review results, do not move on without fixing.
3. If required, fix the issue using the information in the results pane.
4. Once you have fixed a **Warning** or **Failed** task, rerun the task by clicking **Run**.

Run to Selected Task

To run the tasks up to and including the currently selected task:

1. Select the last task that you want to run.
2. Right-click this task to open the context menu.
3. From the context menu, select `Run to Selected Task`.



Note: If a task before the selected task fails, the Workflow Advisor stops at the failed task.

Reset a Task

To reset a task:

1. Select the task that you want to reset.
2. Right-click this task to open the context menu.
3. From the context menu, select `Reset Task` to reset this and subsequent tasks.

Reset All Tasks in a Folder

To reset a task:

1. Select the folder that you want to reset.
2. Right-click this folder to open the context menu.
3. From the context menu, select `Reset Task` to reset the tasks this folder and subsequent folders.

MATLAB to HDL Code and Synthesis

In this section...

MATLAB to HDL Code Conversion

The **MATLAB to HDL Workflow** task in the HDL Workflow Advisor generates HDL code from fixed-point MATLAB code, and simulates and verifies the HDL against the fixed-point algorithm. HDL Coder then runs synthesis, and optionally runs place and route to generate a circuit description suitable for programming an ASIC or FPGA.

Code Generation: Target Tab

Select target hardware and language and required outputs.

Input Parameters

Target

Target hardware. Select from the list:
- Generic ASIC/FPGA
- Xilinx
- Altera
- Simulation

Language

Select the language (VHDL or Verilog) in which code is generated. The selected language is referred to as the target language.

Default: VHDL

Check HDL Conformance

Enable HDL conformance checking.

Default: Off

Generate HDL

Enable generation of HDL code for the fixed-point MATLAB algorithm.

Default: On

Generate HDL Test Bench

Enable generation of HDL code for the fixed-point test bench.

Default: Off

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 synthesize generated HDL code.

Default: On

Code Generation: Coding Style Tab

Parameters that affect the style of the generated code.

Input Parameters

Preserve MATLAB code comments

Include MATLAB code comments in generated code.

Default: On

Include MATLAB source code as comments

Include MATLAB source code as comments in the generated code. The comments precede the associated generated code. Includes the function signature in the function banner.

Default: On

Generate Report

Enable a code generation report.

Default: Off

VHDL File Extension

Specify the file name extension for generated VHDL files.

Default: .vhd

Verilog File Extension

Specify the file name extension for generated Verilog files.

Default: .v

Comment in header

Specify comment lines in header of generated HDL and test bench files.

Default: None

Text entered in this field as a character vector generates a comment line in the header of the generated code. The code generator adds leading comment characters for the target language. When newlines or linefeeds are included in the text, the code generator emits single-line comments for each newline.

Package postfix

HDL Coder applies this option only if a package file is required for the design.

Default: _pkg

Entity conflict postfix

Specify the character vector to resolve duplicate VHDL entity or Verilog module names in generated code.

Default: _block

Reserved word postfix

Specify a character vector to append to value names, postfix values, or labels that are VHDL or Verilog reserved words.

Default: _rsvd

Clocked process postfix

Specify a character vector to append to HDL clock process names.

Default: _process

Complex real part postfix

Specify a character vector to append to real part of complex signal names.

Default: '_re'

Complex imaginary part postfix

Specify a character vector to append to imaginary part of complex signal names.

Default: '_im'

Pipeline postfix

Specify a character vector to append to names of input or output pipeline registers.

Default: '_pipe'

Enable prefix

Specify the base name as a character vector for internal clock enables and other flow control signals in generated code.

Default: 'enb'

Code Generation: Clocks and Ports Tab

Clock and port settings

Input Parameters

Reset type

Specify whether to use asynchronous or synchronous reset logic when generating HDL code for registers.

Default: Asynchronous

Reset Asserted level

Specify whether the asserted (active) level of reset input signal is active-high or active-low.

Default: Active-high

Reset input port

Enter the name for the reset input port in generated HDL code.

Default: reset

Clock input port

Specify the name for the clock input port in generated HDL code.

Default: clk

Clock enable input port

Specify the name for the clock enable input port in generated HDL code.

Default: clk

Oversampling factor

Specify frequency of global oversampling clock as a multiple of the design under test (DUT) base rate (1).

Default: 1

Input data type

Specify the HDL data type for input ports.

For VHDL, the options are:

- std_logic_vector

  Specifies VHDL type STD_LOGIC_VECTOR
- signed/unsigned

  Specifies VHDL type SIGNED or UNSIGNED

Default: std_logic_vector

For Verilog, the options are:

- In generated Verilog code, the data type for all ports is 'wire'. Therefore, **Input data type** is disabled when the target language is Verilog.

Default: wire

Output data type

Specify the HDL data type for output data types.

For VHDL, the options are:

- Same as input data type

  Specifies that output ports have the same type specified by Input data type.

- std_logic_vector

  Specifies VHDL type STD_LOGIC_VECTOR
- signed/unsigned

  Specifies VHDL type SIGNED or UNSIGNED

Default: Same as input data type

For Verilog, the options are:

- In generated Verilog code, the data type for all ports is ‘wire’. Therefore, Output data type is disabled when the target language is Verilog.

  Default: wire

Clock enable output port

Specify the name for the clock enable input port in generated HDL code.

Default: clk_enable

Code Generation: Test Bench Tab

Test bench settings.

Input Parameters

Test bench name postfix

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

Default: '_tb’

Force clock

Specify whether the test bench forces clock enable input signals.

Default: On

Clock High time (ns)

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

Default: 5

Clock low time (ns)

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

Default: 5

Hold time (ns)

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

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

Setup time (ns)

Display setup time for data input signals.

Default: 0

Force clock enable

Specify whether the test bench forces clock enable input signals.

Default: On

Clock enable delay (in clock cycles)

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

Default: 1

Force reset

Specify whether the test bench forces reset input signals.

Default: On

Reset length (in clock cycles)

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

Default: 2

Hold input data between samples

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

Default: On

Initialize testbench inputs

Specify initial value driven on test bench inputs before data is asserted to device under test (DUT).

Default: Off

Multi file testbench

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

Default: Off

Test bench data file name postfix

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

Default: '_data’

Test bench reference post fix

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

Default: '_ref'

Ignore data checking (number of samples)

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

Default: 0

Use fiaccel to accelerate test bench logging

To generate a test bench, HDL Coder simulates the original MATLAB code. Use the Fixed-Point Designer `fiaccel` function to accelerate this simulation and accelerate test bench logging.

Default: On

Code Generation: Optimizations Tab

Optimization settings

Input Parameters

Map persistent array variables to RAMs

Select to map persistent array variables to RAMs instead of mapping to shift registers.

Default: Off

Dependencies:

- **RAM Mapping Threshold**
- **Persistent variable names for RAM Mapping**

RAM Mapping Threshold

Specify the minimum RAM size required for mapping persistent array variables to RAMs.

Default: 256

Persistent variable names for RAM Mapping

Provide the names of the persistent variables to map to RAMs.

Default: None

Input Pipelining

Specify number of pipeline registers to insert at top level input ports. Can improve performance and help to meet timing constraints.

Default: 0

Output Pipelining

Specify number of pipeline registers to insert at top level output ports. Can improve performance and help to meet timing constraints.

Default: 0

Distribute Pipeline Registers

Reduces critical path by changing placement of registers in design. Operates on all registers, including those inserted using the **Input Pipelining** and **Output Pipelining** parameters, and internal design registers.

Default: Off

Sharing Factor

Number of additional sources that can share a single resource, such as a multiplier. To share resources, set **Sharing Factor** to 2 or higher; a value of 0 or 1 turns off sharing.

In a design that performs identical multiplication operations, HDL Coder can reduce the number of multipliers by the sharing factor. This can significantly reduce area.

Default: 0

Simulation and Verification

Simulates the generated HDL code using the selected simulation tool.

Input Parameters

Simulation tool

Lists the available simulation tools.

Default: None

Skip this step

Default: Off

Results and Recommended Actions

Conditions	Recommended Action
No simulation tool available on system path.	Add your simulation tool path to the MATLAB system path, then restart MATLAB. For more information, see “Synthesis Tool Path Setup”.

Synthesis and Analysis

This folder contains tasks to create a synthesis project for the HDL code. The task then runs the synthesis and, optionally, runs place and route to generate a circuit description suitable for programming an ASIC or FPGA.

Input Parameters

Skip this step

Default: Off

Skip this step if you are interested only in simulation or you do not have a synthesis tool.

Create Project

Create synthesis project for supported synthesis tool.

Description

This task creates a synthesis project for the selected synthesis tool and loads the project with the HDL code generated for your MATLAB algorithm.

You can select the family, device, package, and speed that you want.

When the project creation is complete, the HDL Workflow Advisor displays a link to the project in the right pane. Click this link to view the project in the synthesis tool's project window.

Input Parameters

Synthesis Tool

Select from the list:

- `Altera Quartus II`

  Generate a synthesis project for Altera Quartus II. When you select this option, HDL Coder sets:

  - **Chip Family** to `Stratix II`
  - **Device Name** to `EP2S60F1020C4`

  You can manually change these settings.
- `Xilinx ISE`

  Generate a synthesis project for Xilinx ISE. When you select this option, HDL Coder:

  - Sets **Chip Family** to `Virtex4`
  - Sets **Device Name** to `xc4vsx35`
  - Sets **Package Name** to `ff6...`
  - Sets **Speed Value** to `–...`

  You can manually change these settings.

Default: `No Synthesis Tool Specified`

When you select `No Synthesis Tool Specified`, HDL Coder does not generate a synthesis project. It clears and disables the fields in the **Synthesis Tool Selection** pane.

Chip Family

Target device family.

Default: None

Device Name

Specific target device, within selected family.

Default: None

Package Name

Available package choices. The family and device determine these choices.

Default: None

Speed Value

Available speed choices. The family, device, and package determine these choices.

Default: None

Results and Recommended Actions

Conditions	Recommended Action
Synthesis tool fails to create project.	Read the error message returned by synthesis tool, then check the synthesis tool version, and check that you have write permission for the project folder.
Synthesis tool does not appear in dropdown list.	Add your synthesis tool path to the MATLAB system path, then restart MATLAB. For more information, see "Synthesis Tool Path Setup".

Run Logic Synthesis

Launch selected synthesis tool and synthesize the generated HDL code.

Description

This task:

- Launches the synthesis tool in the background.
- Opens the previously generated synthesis project, compiles HDL code, synthesizes the design, and emits netlists and related files.

- Displays a synthesis log in the **Result** subpane.

Results and Recommended Actions

Conditions	Recommended Action
Synthesis tool fails when running place and route.	Read the error message returned by the synthesis tool, modify the MATLAB code, then rerun from the beginning of the HDL Coder workflow.

Run Place and Route

Launches the synthesis tool in the background and runs a Place and Route process.

Description

This task:

- Launches the synthesis tool in the background.
- Runs a Place and Route process that takes the circuit description produced by the previous mapping process, and emits a circuit description suitable for programming an FPGA.
- Displays a log in the Result subpane.

Input Parameters

Skip this step

If you select **Skip this step**, the HDL Workflow Advisor executes the workflow, but omits the Perform Place and Route, marking it Passed. You might want to select **Skip this step** if you prefer to do place and route work manually.

Default: Off

Results and Recommended Actions

Conditions	Recommended Action
Synthesis tool fails when running place and route.	Read the error message returned by the synthesis tool, modify the MATLAB code, then rerun from the beginning of the HDL Coder workflow.

HDL Code Generation from Simulink

Model Design for HDL Code Generation

Signal and Data Type Support

In this section...

Overview

HDL Coder supports code generation for Simulink signal types and data types with a few special cases.

Buses

You can generate HDL code for designs that use virtual and nonvirtual buses. For example, you can generate code for designs that contain:

- DUT subsystem ports connected to buses.
- Simulink and Stateflow® blocks that support buses and HDL code generation.

Bus Support Limitations

Buses are not supported in the IP Core Generation workflow. In addition, you cannot generate code for designs that use:

- A Black box model reference connected to a bus.
- A bus input to a Delay block with nonzero **Initial condition**.

Enumerations

You can generate code for Simulink, MATLAB, or Stateflow enumerations within your design.

Requirements

- The enumeration values must be monotonically increasing.

- If your target language is Verilog, all enumeration member names must be unique within the design.

Restrictions

Enumerations at the top-level DUT ports are not supported with the following workflows or verification methods:

- IP Core Generation workflow
- FPGA Turnkey workflow
- Simulink Real-Time FPGA I/O workflow
- Customization for the USRP Device workflow
- FPGA-in-the-loop
- HDL Cosimulation

Unsupported Signal and Data Types

Variable-size signals are not supported for code generation.

Related Examples

- “Generating HDL Code for Subsystems with Array of Buses” on page 10-24

More About

- “Signal Types” (Simulink)
- “About Data Types in Simulink” (Simulink)
- “Composite (Bus) Signals” (Simulink)
- “Use Enumerated Data in Simulink Models” (Simulink)
- “Enumerated Data” (Stateflow)

Simulink Templates for HDL Code Generation

HDL Coder model templates in Simulink provide you with design patterns and best practices for models intended for HDL code generation.

Models you create from one of the HDL Coder model templates have their configuration parameters and solver settings set up for HDL code generation. You can use `hdlsetup` to configure an existing model for HDL code generation.

To learn how to model hardware for efficient HDL code generation, create a model using an HDL Coder model template.

In this section...
"Create Model Using HDL Coder Model Template" on page 10-4
"HDL Coder Model Templates" on page 10-4

Create Model Using HDL Coder Model Template

1 To open the Simulink start page, in the MATLAB **Home** tab, click **Simulink** or at the command prompt, enter `simulink`.
2 In the Simulink start page, click the Blank Model template, and then click Create Model.

   A new model, with the template contents and settings, opens in the Simulink Editor. Click **File** > **Save as** to save the model.
3 To open the Simulink Library Browser, click the **Library Browser** button in the Simulink editor. Alternatively, at the command prompt, enter `slLibraryBrowser`.

HDL Coder Model Templates

- "Complex Multiplier" on page 10-5
- "MATLAB Arithmetic" on page 10-6
- "ROM" on page 10-9
- "Register" on page 10-10
- "SRL" on page 10-12

- "Simulink Hardware Patterns" on page 10-14
- "State Machine in MATLAB" on page 10-17

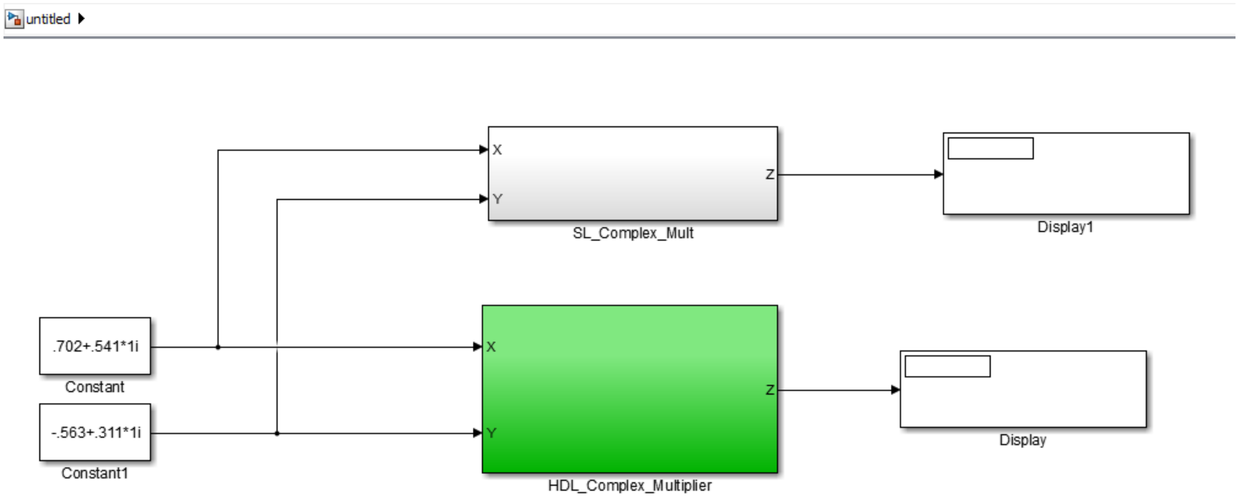
Complex Multiplier

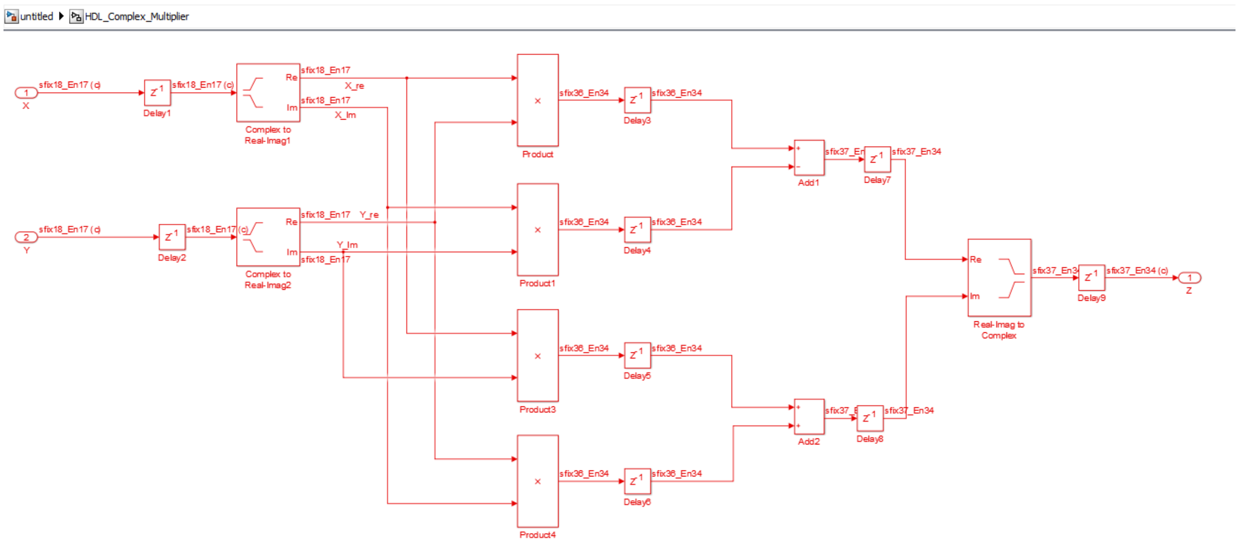
The Complex Multiplier template shows how to model a complex multiplier-accumulator and manually pipeline the intermediate stages. The hardware implementation of complex multiplication uses four multipliers and two adders.

The template applies the following best practices:

- In the Configuration Parameters dialog box, in **HDL Code Generation** > **Global Settings**, **Reset type** is set to `Synchronous`.
- To improve speed, Delay blocks, which map to registers in hardware, are at the inputs and outputs of the multipliers and adders.
- To support the output data of a full-precision complex multiplier, the output data word length is manually specified to be (*`operand_word_length`* * 2) + 1.

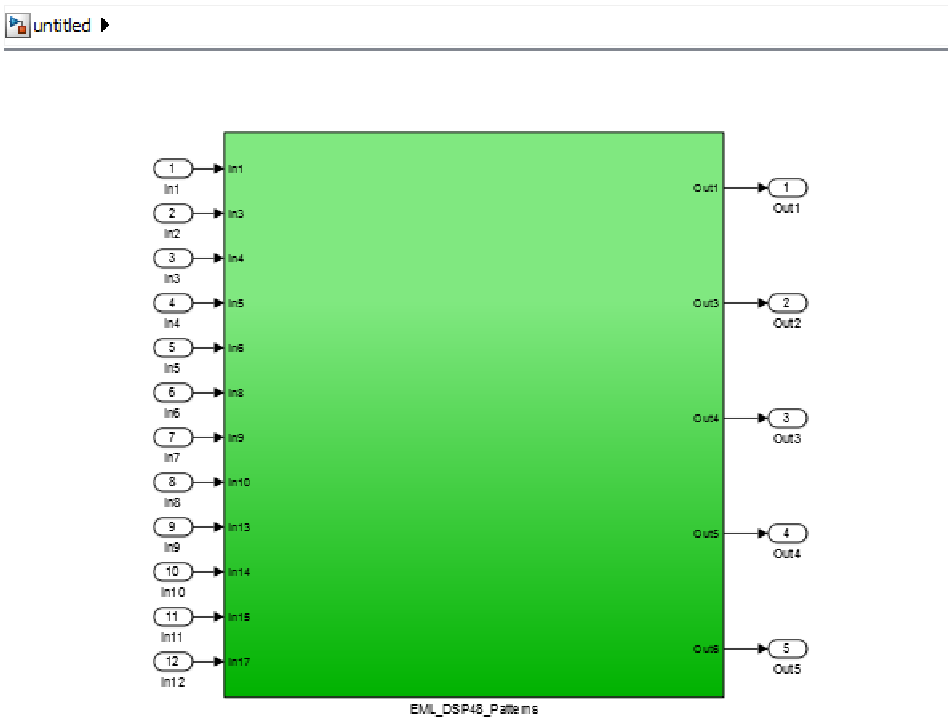
  For example, in the template, the operand word length is 18, and the output word length is 37.

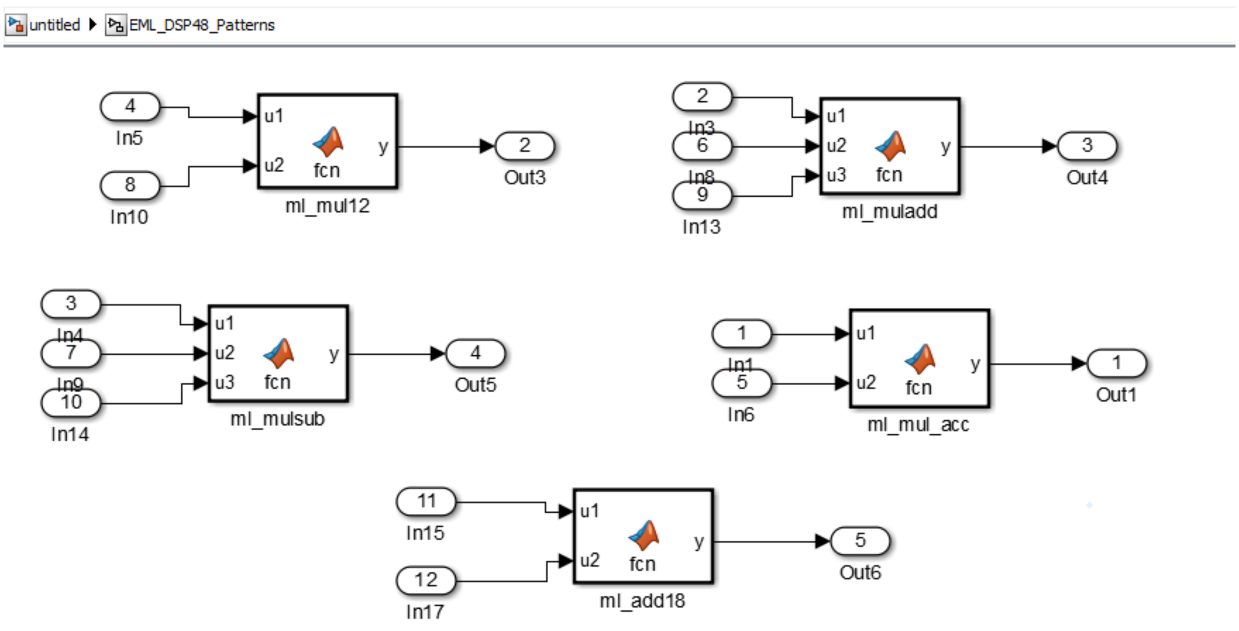




MATLAB Arithmetic

The MATLAB Arithmetic template contains MATLAB arithmetic operations that infer DSP48s in hardware.


untitled
1
In1
2
In2
3
In3
4
In4
5
In5
6
In6
7
In7
8
In8
9
In9
10
In10
11
In11
12
In12
In1
In3
In4
In5
In6
In8
In9
In10
In13
In14
In15
In17
Out1
Out3
Out4
Out5
Out6
1
Out1
2
Out2
3
Out3
4
Out4
5
Out5
EML_DSP48_Patterns



For example, the `ml_mul_acc` MATLAB Function block shows how to write a multiply-accumulate operation in MATLAB. hdlfimath on page 20-36 applies fixed-point math settings for HDL code generation.

```
function y = fcn(u1, u2)

% design of a 6x6 multipler
% same reset on inputs and outputs
% followed by an adder

nt = numerictype(0,6,0);
nt2 = numerictype(0,12,0);
fm = hdlfimath;

persistent u1_reg u2_reg mul_reg add_reg;
if isempty(u1_reg)
    u1_reg = fi(0, nt, fm);
    u2_reg = fi(0, nt, fm);
    mul_reg = fi(0, nt2, fm);
    add_reg = fi(0, nt2, fm);
```

```
end

mul = mul_reg;
mul_reg = u1_reg * u2_reg;
add = add_reg;
add_reg(:) = mul+add;
u1_reg = u1;
u2_reg = u2;

y = add;
```

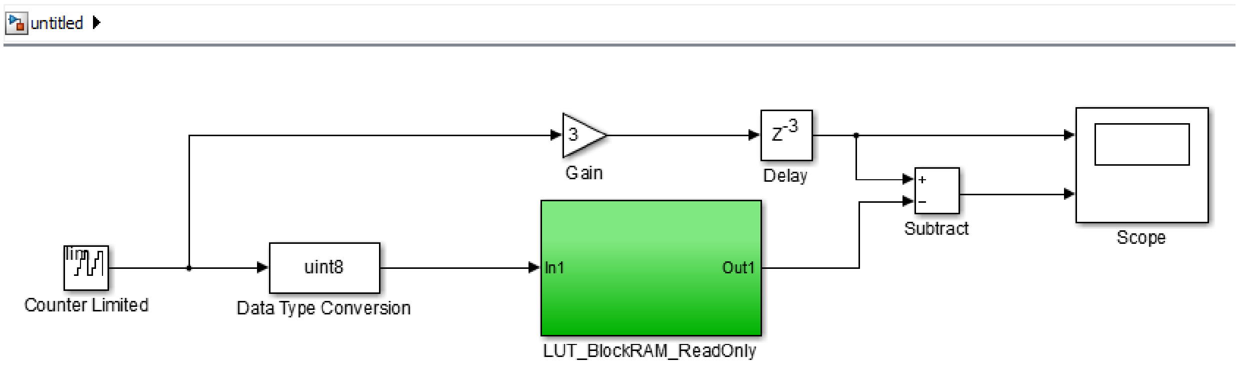
ROM

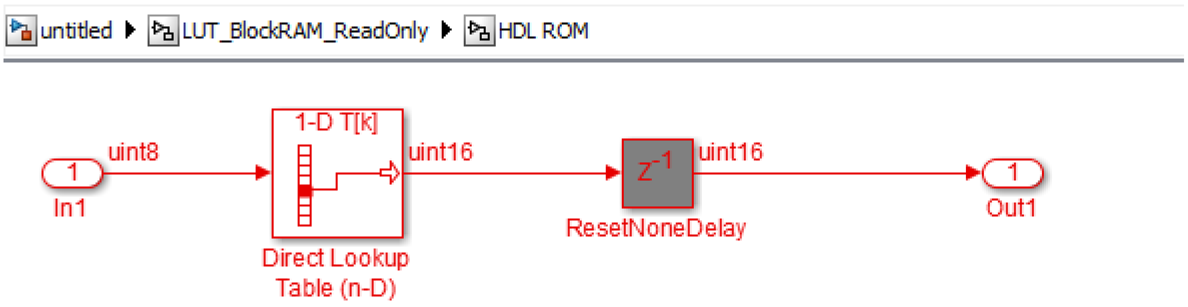
The ROM template is a design pattern that maps to a ROM in hardware.

The template applies the following best practices:

- At the output of the lookup table, there is a Delay block with `ResetType` = `none`.
- The lookup table is structured such that the spacing between breakpoints is a power of two.

  Using table dimensions that are a power of two enables HDL Coder to generate shift operations instead of division operations. If necessary, pad the table with zeros.
- The number of lookup table entries is a power of two. For some synthesis tools, a lookup table that has a power-of-two number of entries maps better to ROM. If necessary, pad the table with zeros.





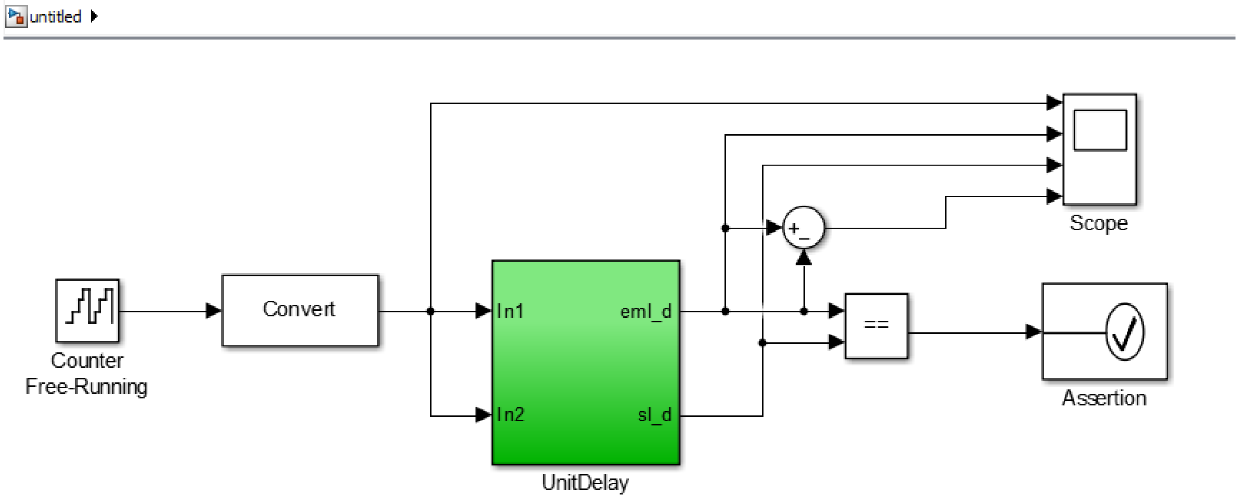
```
x=(0:99)';
Scale_by_3_LUT=3*x;
pad=2^nextpow2(length(Scale_by_3_LUT))-length(Scale_by_3_LUT);
Scale_by_3_LUT_pad=[Scale_by_3_LUT;zeros(pad,1)];
```

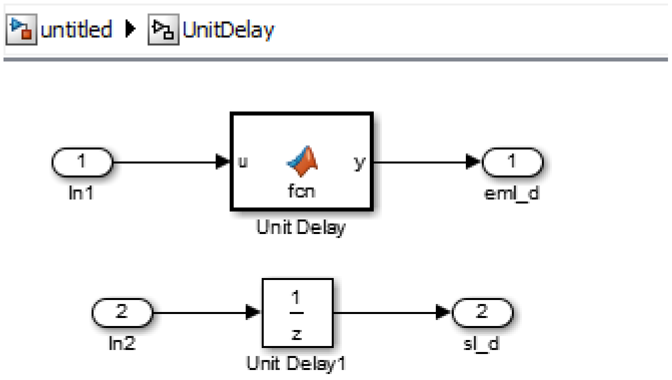
Register

The Register template shows how to model hardware registers:

- In Simulink, using the Delay block.
- In MATLAB, using persistent variables.

  This design pattern also shows how to use `cast` to propagate data types automatically.





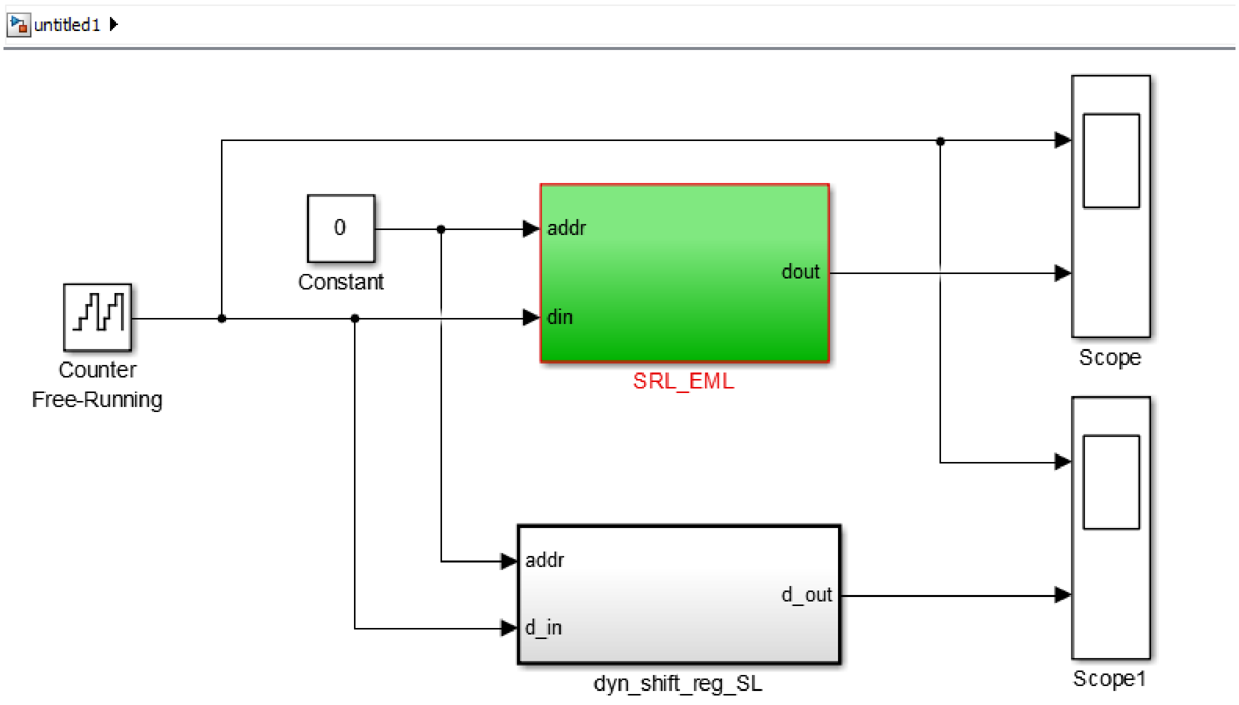
The MATLAB code in the MATLAB Function block uses a persistent variable to model the register.

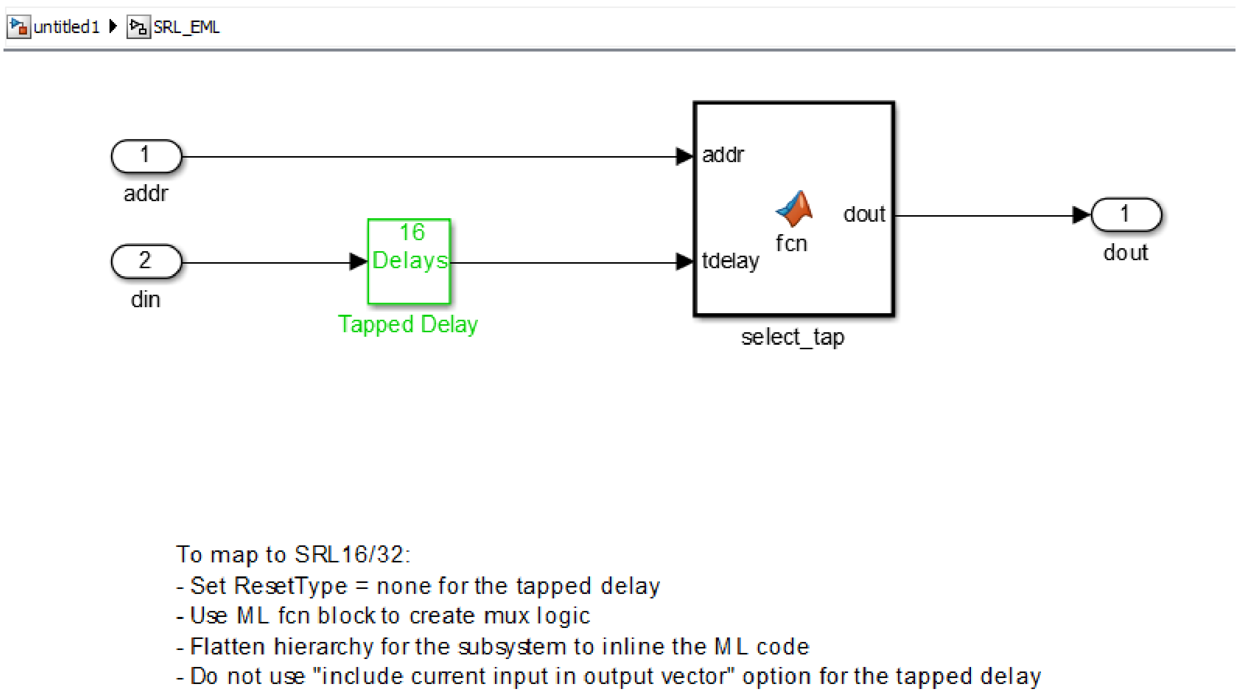
```
function y = fcn(u)
% Unit delay implementation that maps to a register in hardware

persistent u_d;
```

```
if isempty(u_d)
    % defines initial value driven by unit delay at time step 0
    u_d = cast(0, 'like', u);
end

% return delayed input from last sample time hit
y = u_d;

% store the current input
u_d = u;
```

SRL

The SRL template shows how to implement a shift register that maps to an SRL16 in hardware. You can use a similar pattern to map to an SRL32.





To map to SRL16/32:
- Set ResetType = none for the tapped delay
- Use ML fcn block to create mux logic
- Flatten hierarchy for the subsystem to inline the ML code
- Do not use "include current input in output vector" option for the tapped delay

In the shift register subsystem, the Tapped Delay implements the shift operation, and the MATLAB Function, `select_tap`, implements the output mux.

In `select_tap`, the zero-based address, `addr` increments by 1 because MATLAB indices are one-based.

```
function dout = fcn(addr, tdelay)
%#codegen

addr1 = fi(addr+1,0,5,0);
dout = tdelay(addr1);
```
In the generated code, HDL Coder automatically omits the increment because Verilog and VHDL are zero-based.

The template also applies the following best practices for mapping to an SRL16 in hardware:

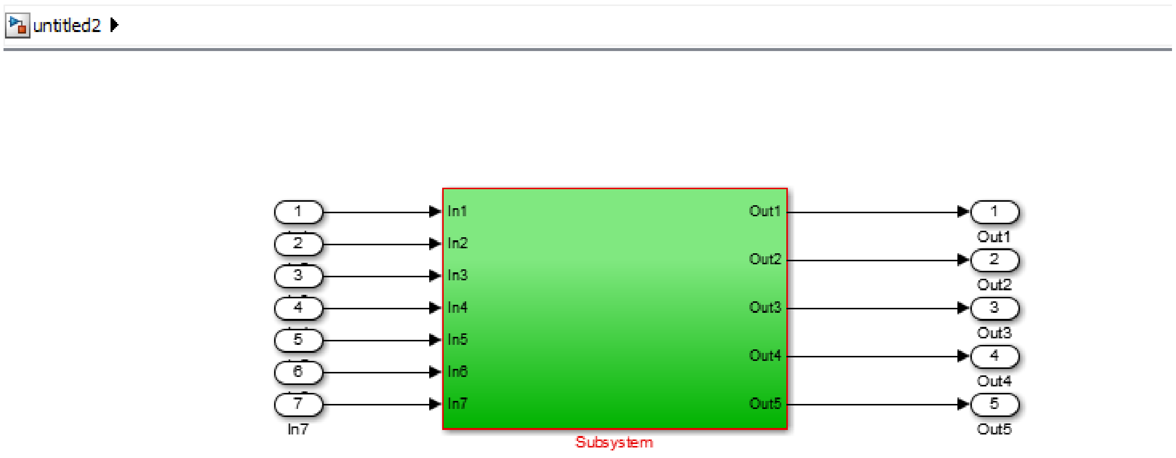
- For the Tapped Delay block:

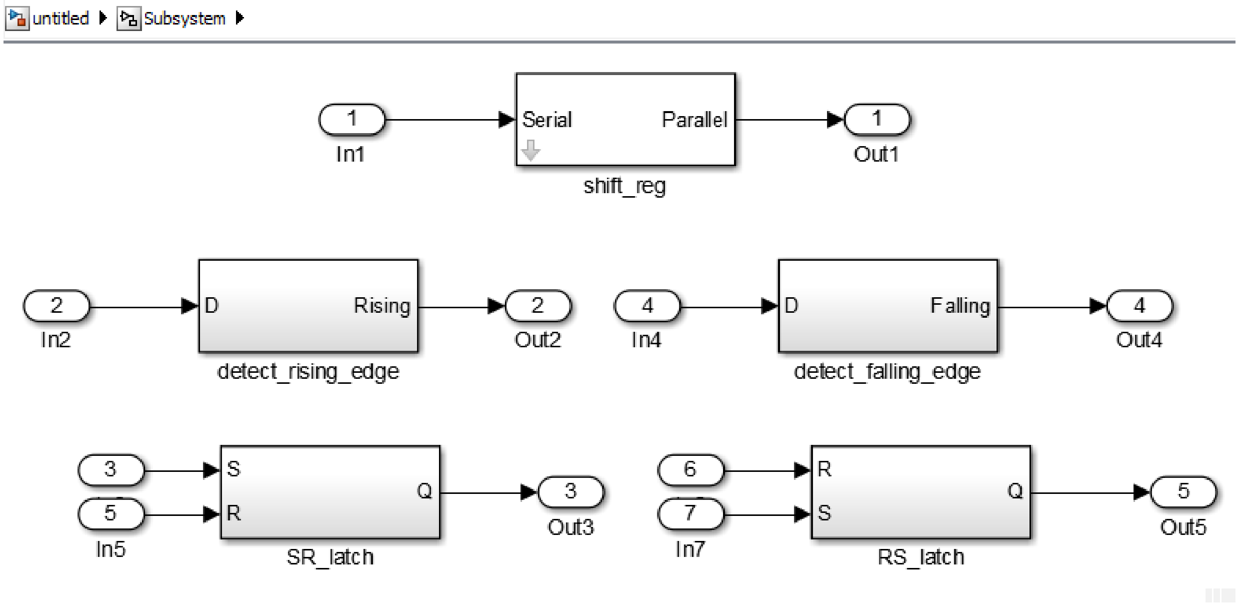
  - In the Block Parameters dialog box, **Include current input in output vector** is not enabled.
  - In the HDL Block Properties dialog box, **ResetType** is set to `none`.
- For the Subsystem block, in the HDL Block Properties dialog box, **FlattenHierarchy** is set to `on`.

Simulink Hardware Patterns

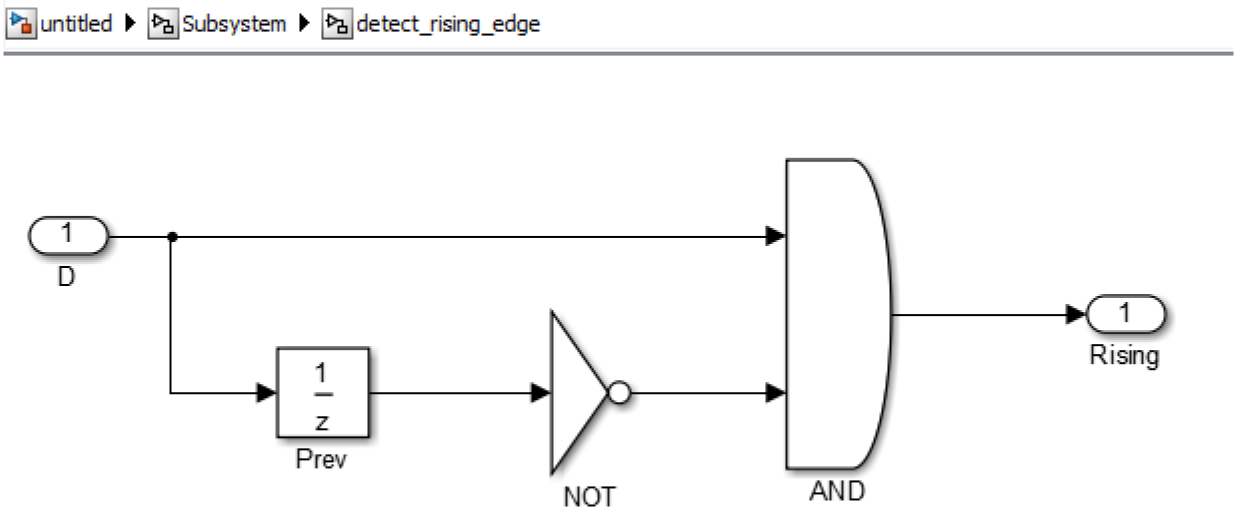
The Simulink Hardware Patterns template contains design patterns for common hardware operations:

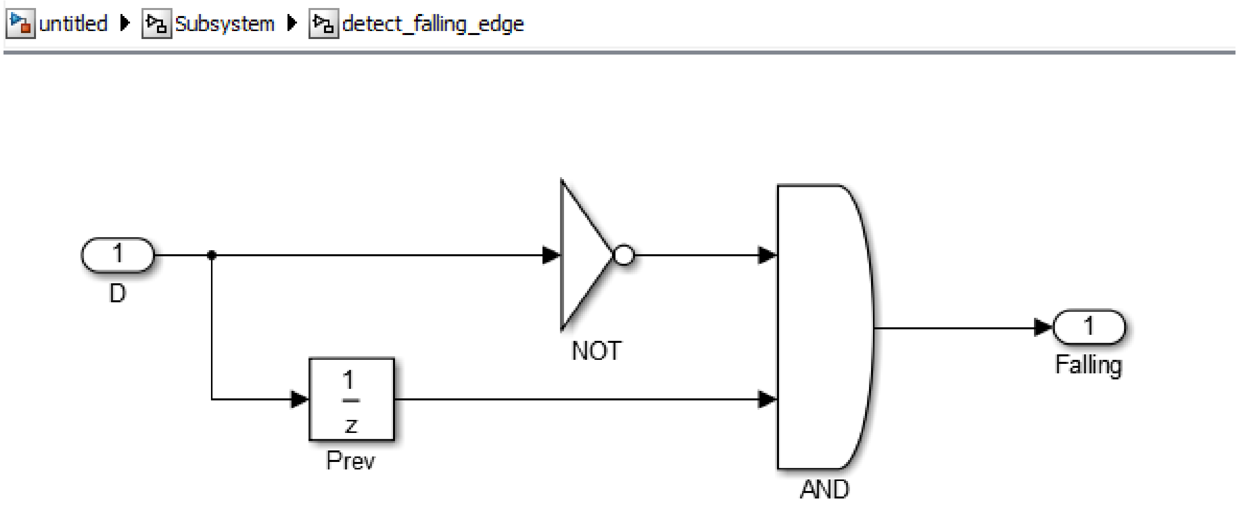
- Serial-to-parallel shift register
- Detect rising edge
- Detect falling edge
- SR latch
- RS latch





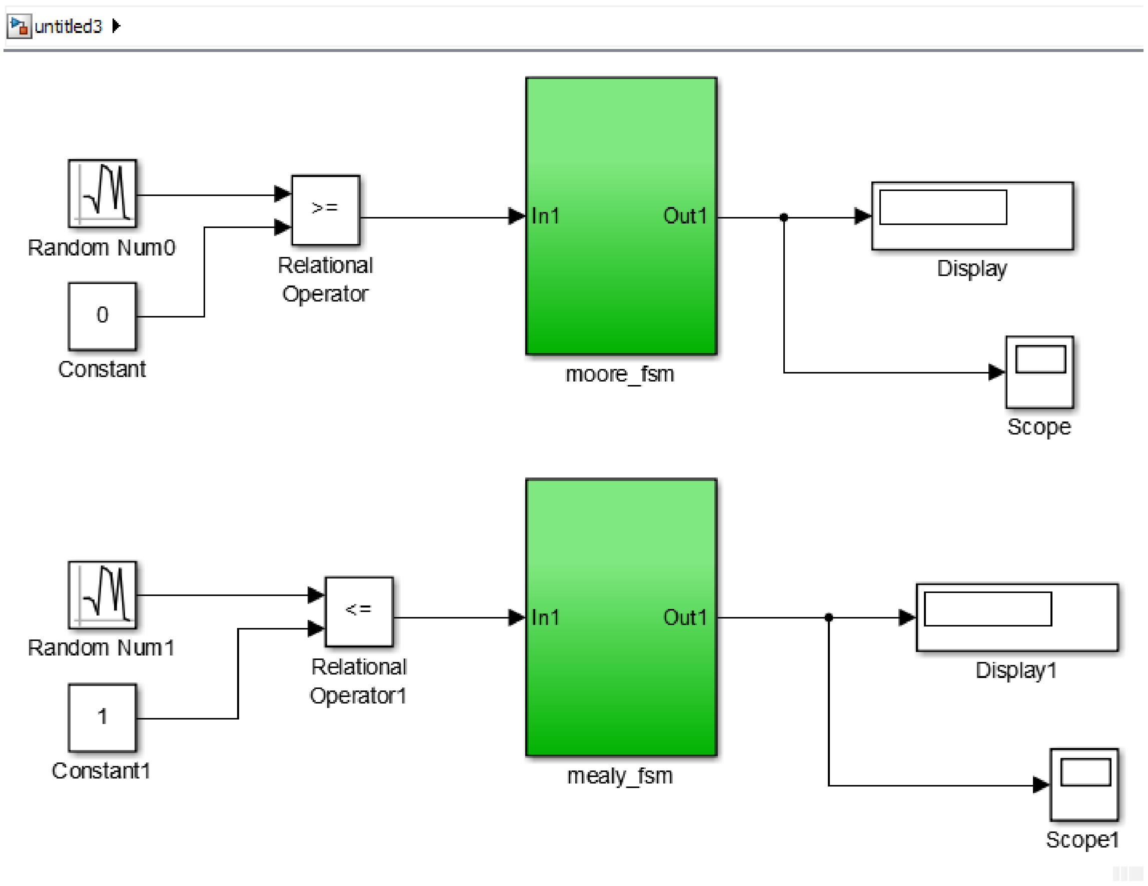
For example, the design patterns for rising edge detection and falling edge detection:


untitled ▶ Subsystem ▶ detect_rising_edge
1
D
1
z
Prev
NOT
AND
1
Rising


untitled ▶ Subsystem ▶ detect_falling_edge
1
D
NOT
1
z
Prev
AND
1
Falling

State Machine in MATLAB

The State Machine in MATLAB template shows how to implement Mealy and Moore state machines using the MATLAB Function block.



To learn more about best practices for modeling state machines, see “Model a State Machine for HDL Code Generation” on page 3-5.

See Also

`hdlsetup` | `makehdl`

More About

- “Prepare Simulink Model For HDL Code Generation”
- “Design Guidelines for the MATLAB Function Block” on page 20-35
- “Hardware Modeling with MATLAB Code”

Generate DUT Ports for Tunable Parameters

In this section...

Tunable parameters that you use to adjust your model behavior during simulation can map to top-level DUT ports in your generated HDL code. HDL Coder generates one DUT port per tunable parameter.

You can generate a DUT port for a tunable parameter by using it in one of these blocks:

- Gain
- Constant
- MATLAB Function
- MATLAB System
- Chart
- Truth Table
- State Transition Table

These blocks with the tunable parameter can be at any level of the DUT hierarchy, including within a model reference.

You cannot use HDL cosimulation with a DUT that uses tunable parameters in any of these blocks. If you use a tunable parameter in a block other than these blocks, code is generated inline and does not map to DUT ports. To use the value of a tunable parameter in a Chart or Truth Table block, see "Use Tunable Parameter in Other Blocks" on page 10-21.

You can define and store the tunable parameters in the base workspace or a Simulink data dictionary. However, a Simulink data dictionary provides more capabilities. For details, see "What Is a Data Dictionary?" (Simulink).

Prerequisites

- The Simulink compiled data type for all instances of a tunable parameter must be the same.
- Simulink blocks that use tunable parameters with the same name must operate at the same data rate.

To learn more about Simulink compiled data types, see “Control Block Parameter Data Types” (Simulink).

Create and Add Tunable Parameter That Maps to DUT Ports

To generate a DUT port for a tunable parameter:

1. Create a tunable parameter with `StorageClass` set to `ExportedGlobal`.

   For example, to create a tunable parameter, *`myParam`*, and initialize it to 5, at the command line, enter:

   ```
   myParam = Simulink.Parameter;
   myParam.Value = 5;
   myParam.CoderInfo.StorageClass = 'ExportedGlobal';
   ```

   Alternatively, using the Model Explorer, you can create a tunable parameter and set **Storage Class** to `ExportedGlobal`. See “Create Data Objects from Built-In Data Class Package Simulink” (Simulink).
2. In your Simulink design, use the tunable parameter as the:

   - **Constant value** in a Constant block.
   - **Gain** parameter in a Gain block.
   - MATLAB function argument in a MATLAB Function block.

Generated Code

The following VHDL code is an example of code that HDL Coder generates for a Gain block with its **Gain** field set to a tunable parameter, *`myParam`*:

```
ENTITY s IS
  PORT( In1           : IN   std_logic_vector(15 DOWNTO 0); -- sfix16_En5
        myParam       : IN   std_logic_vector(15 DOWNTO 0); -- sfix16_En5
```

```
        Out1          : OUT  std_logic_vector(31 DOWNTO 0) -- sfix32_En10
        );
END s;


ARCHITECTURE rtl OF s IS

  -- Signals
  SIGNAL myParam_signed  : signed(15 DOWNTO 0); -- sfix16_En5
  SIGNAL In1_signed      : signed(15 DOWNTO 0); -- sfix16_En5
  SIGNAL Gain_out1       : signed(31 DOWNTO 0); -- sfix32_En10

BEGIN
  myParam_signed <= signed(myParam);

  In1_signed <= signed(In1);

  Gain_out1 <= myParam_signed * In1_signed;

  Out1 <= std_logic_vector(Gain_out1);

END rtl;
```

Limitations

`TriggerAsClock` must be `off`.

Use Tunable Parameter in Other Blocks

To use the value of a tunable parameter in a Chart or Truth Table block:

1. Create the tunable parameter and use it in a Constant block.
2. Add an input port to the block where you want to use the tunable parameter.
3. Connect the output of the Constant block to the new input port.

Related Examples

- "Generate Parameterized Code for Referenced Models" on page 10-22

Generate Parameterized Code for Referenced Models

In this section...
"Parameterize Referenced Model for HDL Code Generation" on page 10-22
"Restrictions" on page 10-22

To generate parameterized code for referenced models, use model arguments. You can use model arguments in a masked or unmasked Model block.

HDL Coder generates a single VHDL `entity` or Verilog `module` for the referenced model, even if the DUT has multiple instances of the referenced model. In the generated code, each model argument is a VHDL `generic` or a Verilog `parameter`.

Parameterize Referenced Model for HDL Code Generation

1 In the referenced model, create one or more model arguments.

   To learn how to create a model argument, see "Specify Different Value for Each Instance of Reusable Model" (Simulink).
2 In the referenced model, use each model argument parameter in a Gain or Constant block.
3 In the DUT, for each model reference, in the **Model arguments** table, enter values for each model argument.

   Alternatively, create a model mask for the referenced model. In the DUT, for each model reference, enter values for each model argument.
4 Generate code for the DUT.

Restrictions

Model argument values:

- Must be scalar.
- Cannot be complex.
- Cannot be enumerated data.

Related Examples

- "Generate Reusable Code for Atomic Subsystems" on page 18-20

- “Generate DUT Ports for Tunable Parameters” on page 10-19

More About

- “Specify Different Value for Each Instance of Reusable Model” (Simulink)
- “Model Referencing for HDL Code Generation” on page 18-2

Generating HDL Code for Subsystems with Array of Buses

In this section...

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.

The array of buses represents structured data compactly. The array:

- Reduces the model complexity
- Reduces maintenance by organizing and routing signals in your Simulink model for vectorized algorithms

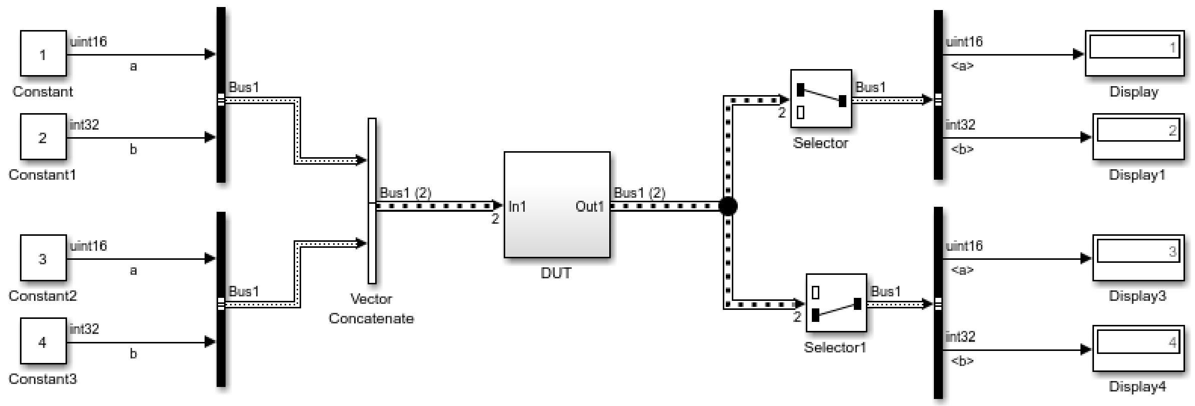
For more information, see "Combine Buses into an Array of Buses" (Simulink).

You can generate HDL code for virtual and nonvirtual blocks that Simulink supports with an array of buses. For more information, see "Use Arrays of Buses in Models" (Simulink).

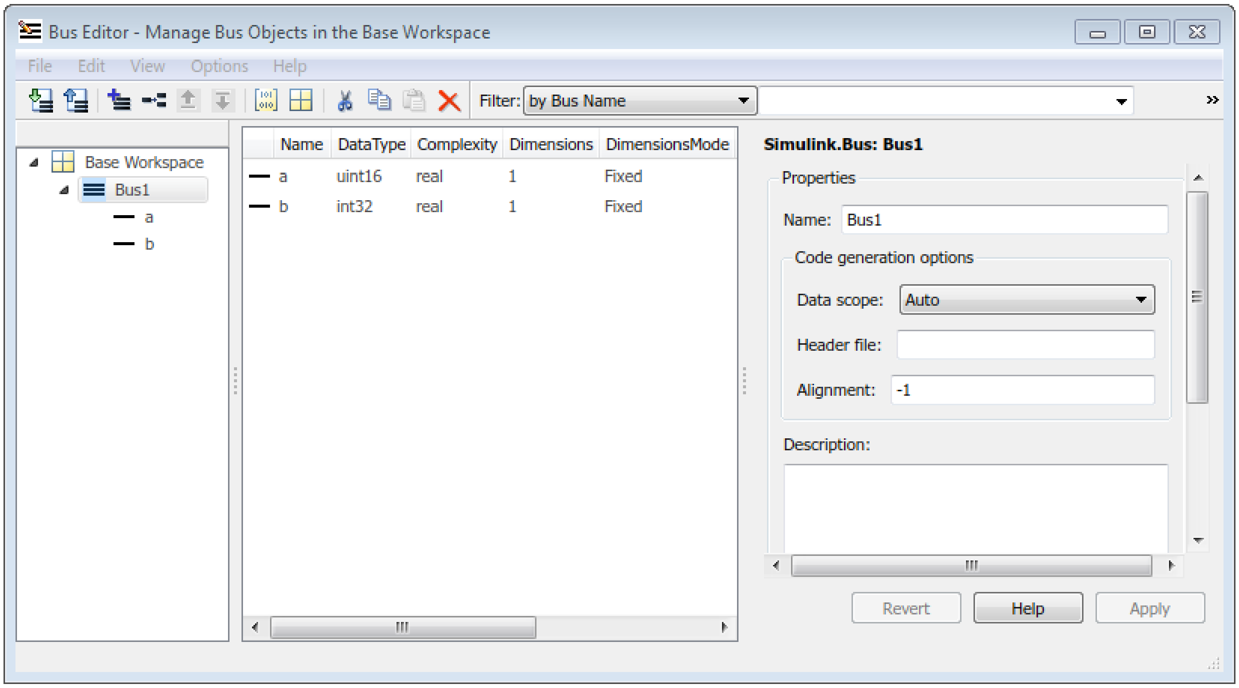
How HDL Coder Generates Code for Array of Buses

HDL Coder expands the array of buses in your Simulink model into the corresponding scalar signals in the generated code.

This Simulink model has an array of buses signal at the DUT interface.



The array of buses combines two nonvirtual bus elements, each having scalars a and b of types uint16 and int32 respectively.



Filter: by Bus Name

Name	DataType	Complexity	Dimensions	DimensionsMode
a	uint16	real	1	Fixed
b	int32	real	1	Fixed

Simulink.Bus: Bus1

Properties

Name: Bus1

Code generation options

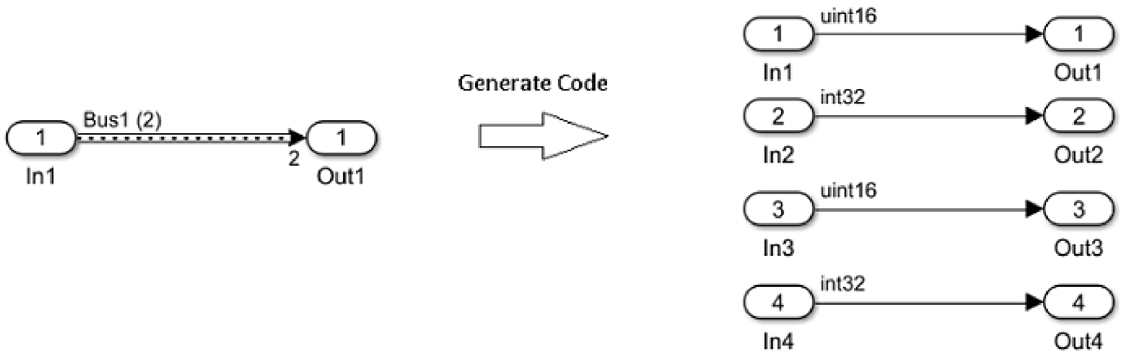
Data scope: Auto

Header file:

Alignment: -1

Description:

The resulting HDL code expands the array of buses into scalars, and contains four scalar input and output ports.



In the generated code, the array of bus expansion results in four scalar signals at the input and output ports. For the first bus object, the input ports are `In_1_a` and `In_1_b`. For the second bus object, they are `In_2_a` and `In_2_b`. At the output, for the first bus object, they are `Out_1_a` and `Out_1_b`. For the second bus object, they are `Out_2_a` and `Out_2_b`.

```
ENTITY DUT IS
  PORT( In1_1_a   :   IN    std_logic_vector(15 DOWNTO 0);  -- uint16
        In1_1_b   :   IN    std_logic_vector(31 DOWNTO 0);  -- int32
        In1_2_a   :   IN    std_logic_vector(15 DOWNTO 0);  -- uint16
        In1_2_b   :   IN    std_logic_vector(31 DOWNTO 0);  -- int32
        Out1_1_a  :   OUT   std_logic_vector(15 DOWNTO 0);  -- uint16
        Out1_1_b  :   OUT   std_logic_vector(31 DOWNTO 0);  -- int32
        Out1_2_a  :   OUT   std_logic_vector(15 DOWNTO 0);  -- uint16
        Out1_2_b  :   OUT   std_logic_vector(31 DOWNTO 0)   -- int32
        );
END DUT;
```

HDL Coder generates code in accordance with the order in which you specify the bus elements and the array elements in your Simulink model. If you specify the VHDL target language for your Simulink model that contains a bus object with arrays, HDL Coder preserves the arrays in the generated code, and does not expand into scalars.

Array of Buses Limitations

- Do not use the array of buses inside other data types. You cannot use a bus signal that contains an array of buses.
- MATLAB System and MATLAB Function blocks that contain System Objects are not supported with an array of buses.

More About

- “Signal and Data Type Support” on page 10-2
- “Signal Types” (Simulink)
- “About Data Types in Simulink” (Simulink)
- “Composite (Bus) Signals” (Simulink)
- “Use Enumerated Data in Simulink Models” (Simulink)
- “Enumerated Data” (Stateflow)

Generate HDL Code for Blocks Inside For Each Subsystem

This example shows how to use blocks inside a For Each Subsystem in your Simulink™ model, and then generate HDL code.

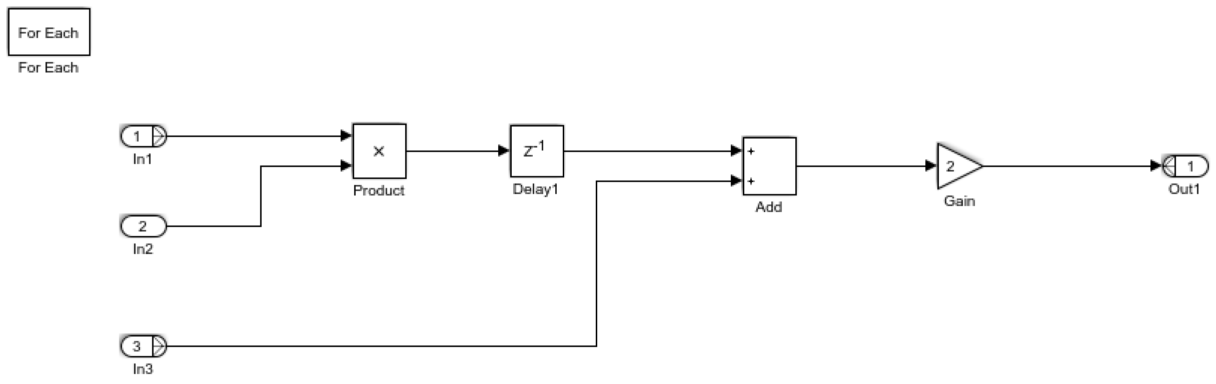
Why Use a For Each Subsystem?

To repeatedly perform the same algorithm on individual elements or subarrays of the input signals, use the For Each Subsystem block. The set of blocks within the Subsystem replicate the algorithm that is applied to individual elements or equally divided subarrays of the input signals. Using the For Each Subsystem block, you do not have to create and connect replicas of a Subsystem block to model the same algorithm. The For Each Subsystem:

- Supports vector processing, which reduces the simulation time of your model. You can process individual elements or subarrays of an input signal simultaneously.
- Improves code readability by using a for-generate loop in the generated HDL code. The for-generate loop reduces the number of lines of code, which can otherwise result in hundreds of lines of code for large vector signals.
- Supports HDL code generation for all data types, Simulink™ blocks, and predefined and user-defined system objects.
- Supports optimizations on and inside the block, such as resource sharing and pipelining. The parallel processing capability of the For Each Subsystem block combined with the optimizations that you specify produces high performance on the target FPGA device.

Modeling With the For Each Subsystem

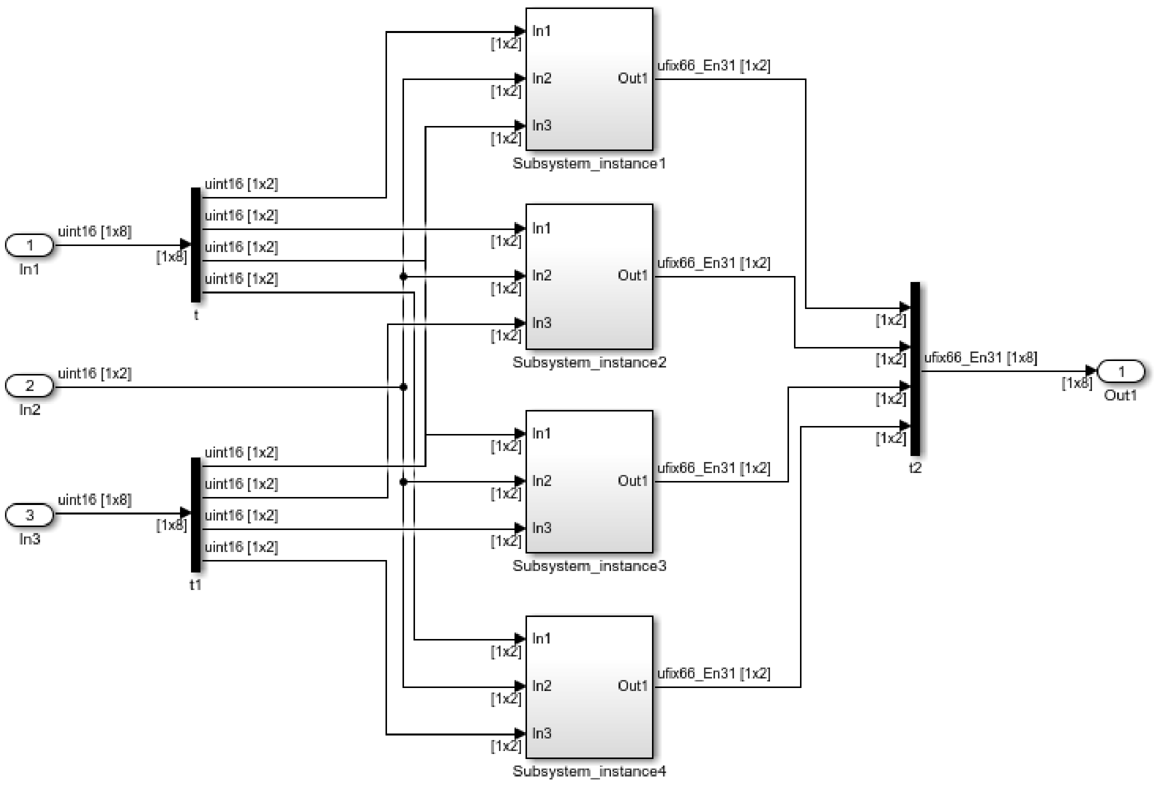
Open the `foreach_subsystem_example1` model. You see this simple algorithm modeled inside a For Each Subsystem block.



When you simulate the model, you see that the input signals `In1` and `In3` are paritioned into subarrays. To see this paritioning, double-click the For Each block. The block parameters **Partition Dimension** and **Partition Width** specify the dimension through which the input signal is partitioned and the width of each partition slice respectively. Based on the input signal sizes and the partitioning that you specify, the For Each Subsystem determines the number of iterations that it requires to compute the algorithm.

In this example, the input signals In1 and In3 of size 8 are partitioned into four subarrays, each of size 2. The input signal In2 of size 2 is not partitioned. To compute the algorithm, the For Each Subsystem requires four iterations, with each iteration repeating the algorithm on each of the four subarrays of In1 and In3.

The For Each Subsystem simplifies modeling of vectorized algorithms. This figure shows how you can model the same algorithm by creating multiple subsystem instances. This model can become graphically complex and difficult to maintain.



Generate HDL Code

To generate HDL code, in the `foreach_subsystem_example1` model, right-click the `Subsystem_Foreach` block and select **HDL Code** > **Generate HDL for Subsystem**.

To see the generated HDL code for the `Subsystem_Foreach` block, in the MATLAB™ Command Window, click the `Subsystem_Foreach.vhd` file. In the VHDL code snippet, you see this for-generate loop in the HDL code.This loop creates four subsystem instances, with each instance performing the algorithm on size 2 subarrays of inputs In1 and In3.

```
BEGIN
  -- <S2>/For Each Subsystem
  GEN_LABEL: FOR k IN 0 TO 3 GENERATE
    u_For_Each_Subsystem : For_Each_Subsystem
      PORT MAP( clk => clk,
                reset => reset,
                enb => clk_enable,
                In1 => In1(2*k TO 2*(k+1) - 1),  -- uint16 [2]
                In2 => In2,  -- uint16 [2]
                In3 => In3(2*k TO 2*(k+1) - 1),  -- uint16 [2]
                Out1 => For_Each_Subsystem_out1(2*k TO 2*(k+1) - 1)
                );
  END GENERATE;
```

Certain optimizations that you specify can change the contents of the subsystems that the For Each Subsystem instantiates. In such cases, the code generator does not use for-generate loops in the HDL code. The HDL code does not contain for-generate loops, if you have:

- Bus or complex input signals.
- Certain optimizations enabled on the subsystem, such as resource sharing and streaming.
- Vector inputs that get partitioned into nonscalar signals in the Verilog code. To obtain for-generate loops in the Verilog code, partition the vector signal to scalars.

See Also

For Each Subsystem

Allocate Sufficient Delays for Floating-Point Operations

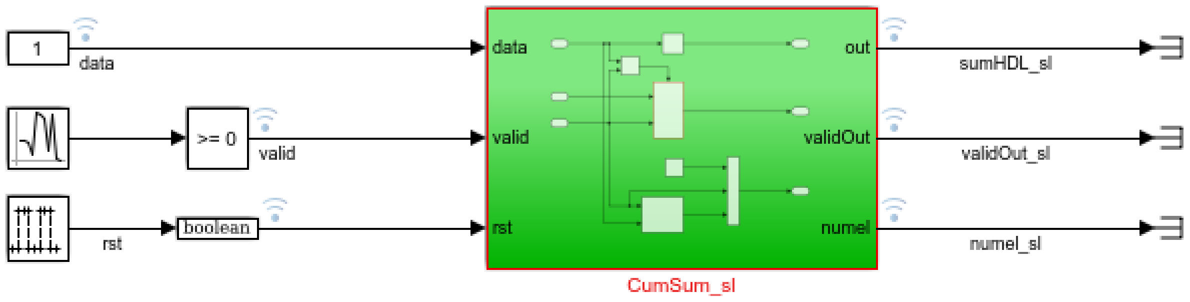
Problem

Sometimes, when generating code from your floating-point algorithm in Simulink, HDL Coder generates an error that it is unable to allocate sufficient number of delays.

Cause

This error message generally occurs when you have Simulink™ blocks performing floating-point operations inside a feedback loop. These blocks have a latency. HDL Coder™ is unable to allocate delays to compensate for the latency, because the code generator needs to add delays and balance them to maintain numerical accuracy.

If you open this example, you see a Simulink™ model that uses single data types.

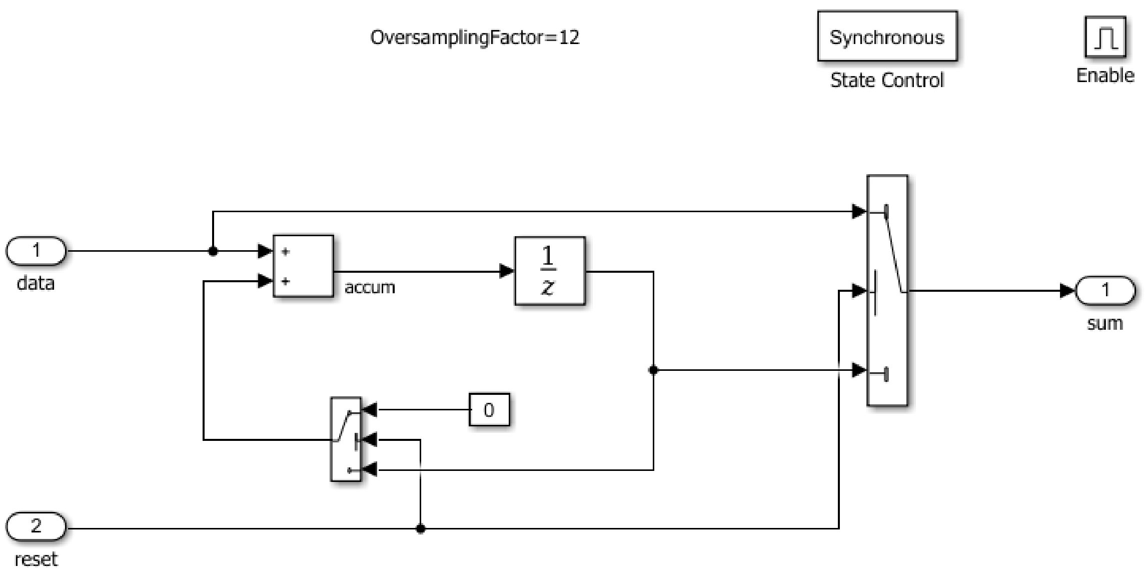


To generate HDL code for the `CumSum_Sl` Subsystem, right-click the subsystem and select **HDL Code** > **Generate HDL for Subsystem**. During code generation, HDL Coder™ generates an error:

```
Unable to allocate delays to compensate for the 12 delay(s)
introduced by RunningSum/CumSum_sl/Subsystem/Add in native floating-
point mode. Consider setting the 'Latency Strategy' to 'Zero', or
```

```
adding the necessary output pipelines via HDL block properties for
other blocks in the model to accommodate for the latency introduced
by this block.
```

By using the path to the block mentioned in the error message, navigate to the Add block in the model. This block is inside a feedback loop.



The Add block has a maximum latency of 12. When generating code, HDL Coder™ cannot allocate 12 delays for the block, because it cannot add matching delays to other paths.

This model serves as an example to illustrate the various strategies to solve this problem.

Solution

Strategy 1: Global Oversampling

This modeling paradigm uses the clock-rate pipelining optimization to oversample your design to a clock-rate much faster than the DUT sample rate. To enable this optimization, specify a global oversampling factor for your Simulink model. The floating-point delays then operate at the faster clock-rate and can be allocated successfully. For more information, see “Clock-Rate Pipelining” on page 15-66.

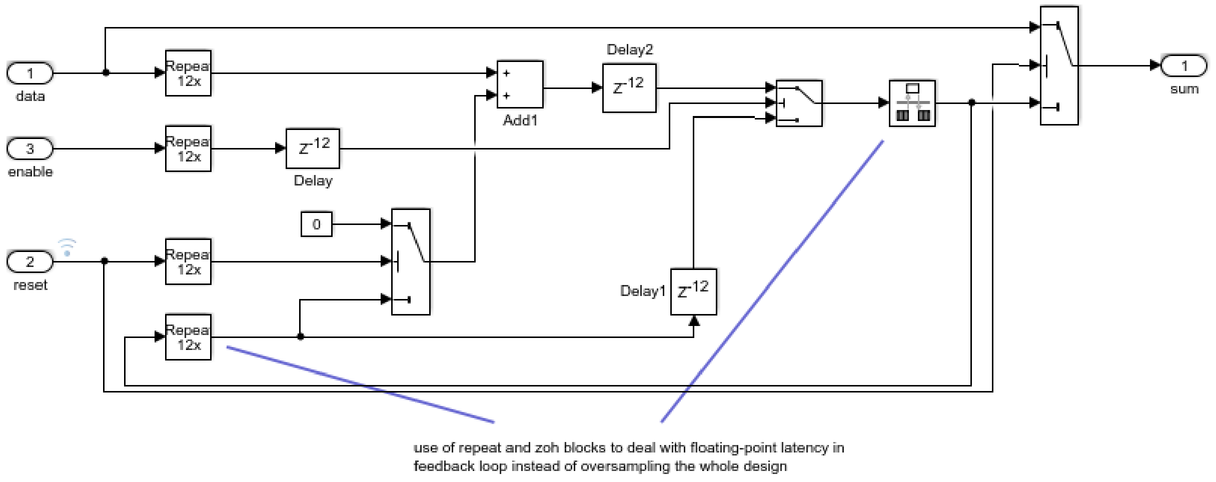
1. Specify an oversampling factor that is equal to or greater than the latency of the floating-point operators that are unable to allocate delays. For the `RunningSum` model, specify an oversampling factor at least equal to `12`. To learn about the latency values of the floating-point operators, see “Operators and Simulink Blocks Supported for Native Floating-Point” on page 10-64.

   To specify the oversampling factor, in the Configuration Parameters dialog box, on the **HDL Code Generation** > **Global Settings** tab, set **Oversampling factor** to `12`.
2. Enable hierarchy flattening on the DUT and make sure that subsystems inside the DUT inherit this setting. For the `RunningSum` model, right-click the `CumSum_sl` subsystem and select **HDL Code** > **HDL Block Properties**, and then set **FlattenHierarchy** to `on`.

Strategy 2: Local Oversampling

To model your design at the data rate and selectively increase the sample rate of blocks for which HDL Coder™ is unable to allocate delays, use local oversampling. These blocks then operate at the faster clock rate and can accommodate the required number of delays.

If you open the `RunningSum_OSmanual` model and navigate to the Add block, it shows how you can increase the sample rate of the Add block and allocate delays.


1
data
3
enable
2
reset
Repeat 12x
Repeat 12x
Repeat 12x
Repeat 12x
z-12
Delay
0
Add1
Delay2
z-12
Delay1
z-12
1
sum
use of repeat and zoh blocks to deal with floating-point latency in feedback loop instead of oversampling the whole design

- The blocks that are within the boundary of the Repeat and Zero Order Hold blocks operate at the clock rate that is 12 times faster than the sample rate of the model.
- The subsystem has a Delay block of length 12 at the output of the Add block. When generating code, the Add block absorbs this Delay block, which compensates for the latency of the operator. To balance delays, the subsystem contains Delay blocks of length 12 in other paths.

You can now generate HDL code for the `CumSum_sl` subsystem. To generate HDL code for the `CumSum_Sl` Subsystem, right-click the subsystem and select **HDL Code** > **Generate HDL for Subsystem**.

Strategy 3: Delay Blocks

Use this modeling paradigm to model your entire design at the Simulink data rate. For blocks that are unable to accommodate the required number of delays, add a Delay block with a sufficient **Delay length** at the output of the blocks. Specify a **Delay length** that is equal to the latency of the floating-point operator. Make sure that you add matching delays in other paths.

For the `RunningSum` model, you can add a Delay block of length 12 at the output of the Add block. When generating code, the Add block absorbs this delay, because the block has a latency of 12.

For more information, see “Delay Blocks in the Model” on page 10-49.

Strategy 4: Zero Latency

You can use the zero latency strategy setting for blocks in your design for which native floating point is unable to allocate delays. By default, blocks in your design inherit the native floating-point settings that you specify in the Configuration Parameters dialog box. To specify a custom latency strategy setting for a block:

1. Right-click the block, and select **HDL Code** > **HDL Block Properties**.
2. In the HDL Block Properties dialog box, switch to the **Native Floating Point** tab. Then, set **LatencyStrategy** to `Zero`.

For the `RunningSum` example, set the **LatencyStrategy** of the Add block to `Zero`. To choose the native floating point library and specify zero latency strategy, at the command line, enter:

```
fc = hdlcoder.createFloatingPointTargetConfig('NativeFloatingPoint');
% Set latency strategy of the Add block in RunningSum model to Zero.
hdlset_param('RunningSum/CumSum_sl/Subsystem/Add','LatencyStrategy', 'Zero');
```

Note: To obtain good performance on the target FPGA device, it is not recommended to set **Latency Strategy** to `Zero` from the Configuration Parameters dialog box.

See Also
`hdlcoder.createFloatingPointTargetConfig` | `NativeFloatingPoint`

Related Examples
•

More About
- “HDL Coder Native Floating-Point Support” on page 10-37
- “Latency Customization with Native Floating-Point” on page 10-47
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-52
- “Limitations of Native Floating-Point Support” on page 10-69

HDL Coder Native Floating-Point Support

In this section...

Native floating-point support in HDL Coder can generate code from your floating-point design. If your design has complex math and trigonometric operations, use native floating-point.

In your Simulink model:

- You can have single-precision floating-point data types and operations.
- You can have a combination of integer, fixed-point, and floating-point operations. By using Data Type Conversion blocks, you can perform conversions between single-precision and fixed-point data types.

The generated code:

- Complies with the IEEE-754 standard of floating-point arithmetic.
- Is target-independent. You can deploy the code on any generic FPGA or an ASIC.
- Does not require floating-point processing units or hard floating-point DSP blocks on the target ASIC or FPGA.

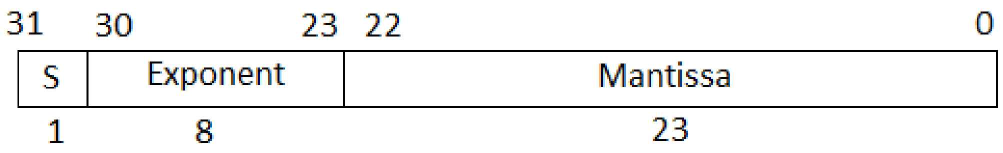
HDL Coder supports:

- Math and trigonometric functions, and a large subset of Simulink blocks.
- Denormal numbers.
- Customizing the latency of the floating-point operator.

Numerical Considerations and IEEE-754 Standard Compliance

HDL Coder generates code in compliance with the IEEE 754–2008 standard of floating-point arithmetic.

In the IEEE 754–2008 standard, the single-precision floating-point number is 32-bit in size. The 32-bit number encodes a 1-bit sign, an 8-bit exponent, and a 23-bit mantissa.



This graph is the normalized representation for floating-point numbers. You can compute the actual value of a normal number as:

$$value = (-1)^{sign} * (1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}) * 2^{(e-127)}$$

The exponent field represents the exponent plus a bias of 127. The size of the mantissa is 24 bits. The leading bit is a 1, so the representation encodes the lower 23 bits.

To generate code that complies with the IEEE-754 standard, HDL Coder supports:

- Round to nearest rounding mode.
- Denormal numbers.
- Exceptions such as NaN (Not a Number), Inf, and Zero.
- Customization of ULP (Units in the Last Place) and relative accuracy.

For more information, see “Numerical Considerations with Native Floating-Point” on page 10-42.

Best Practices

Before you generate HDL code for the subsystem with native floating point, consider these best practices:

Best Practice	Description	See Also
Use blocks from the **HDL Floating Point Operations** block library in HDL Coder.	This 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	“HDL Floating Point Operations”

Best Practice	Description	See Also
	with **Architecture** set to `Fully Parallel`.	
Use single data types in the model based on performance and accuracy requirements.	You can generate HDL code for models that contain single and fixed-point data types in native floating-point mode. Floating-point designs 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 single data types to model the algorithm data path and fixed-point types to model the control logic. To switch between single and fixed-point data types, use Data Type Conversion blocks.	"Data Type Considerations" on page 10-40
Enable optimizations on the model, such as resource sharing.	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 `ShareAdders` to on. • Floating-point multipliers, make sure `ShareMultipliers` is set to on. • Other floating-point resources, set `ShareFloatingPointIP` to on.	"Resource Sharing" on page 15-20

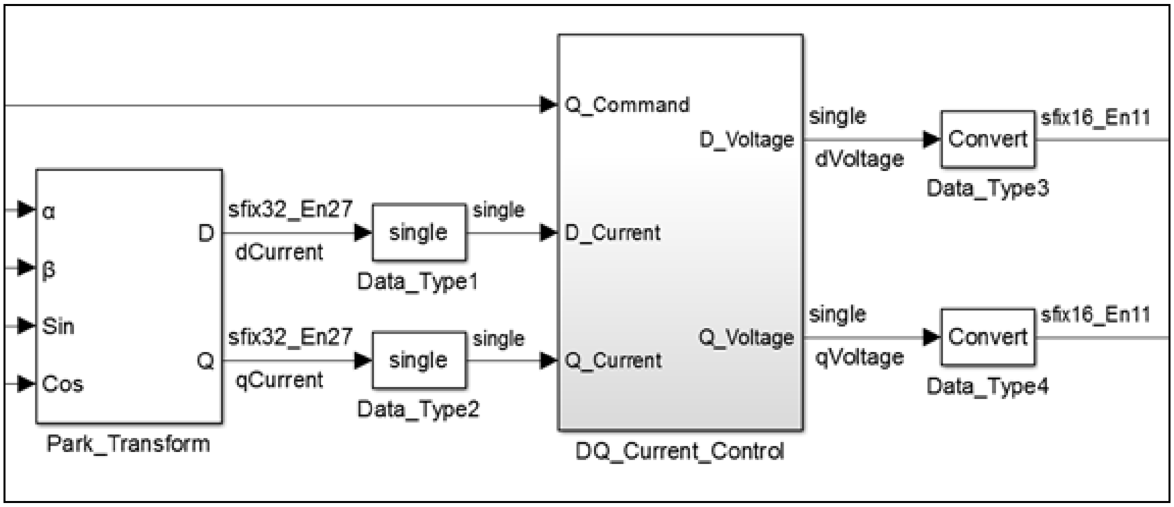
Best Practice	Description	See Also
Customize the latency of your Simulink model or for finer control, selectively customize the latency of certain blocks in your design.	You can customize the latency of an entire model, or selectively for certain blocks in your design. Using custom settings, you can design for trade-offs between latency and throughput.	“Latency Customization with Native Floating-Point” on page 10-47
If you are computing sine and cosine of an 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.	Trigonometric Function
Use `Tree` as the **HDL Architecture**.	To obtain a lower latency implementation, use `Tree` as the **HDL Architecture** for blocks that support this architecture, such as the Sum of Elements and Product of Elements.	Sum of Elements, Product of Elements

Data Type Considerations

With the native floating-point support, HDL Coder supports code generation from Simulink models that contain single-precision floating-point signals and fixed-point signals. Sometimes, you want to model your design with floating-point types to:

- Implement algorithms that have a large or unknown dynamic range that can fall outside the range of representable fixed-point types.
- Implement complex math and trigonometric operations that are difficult to design in fixed point.
- Obtain a higher precision and better accuracy.

Floating-point designs can potentially occupy more area on the target hardware. To design for these trade-offs, in your Simulink model, it is recommended to use single data types in the algorithm data path and fixed-point data types in the control logic of your algorithm. This figure shows a section of a Simulink model that uses `Single` and fixed-point types. By using Data Type Conversion blocks, you can perform conversions between the single and fixed-point types.



See Also

`hdlcoder.createFloatingPointTargetConfig` | `NativeFloatingPoint`

Related Examples

•

More About

- "Generate Target-Independent HDL Code with Native Floating-Point" on page 10-52
- "Latency Customization with Native Floating-Point" on page 10-47
- "Operators and Simulink Blocks Supported for Native Floating-Point" on page 10-64
- "Limitations of Native Floating-Point Support" on page 10-69

Numerical Considerations with Native Floating-Point

In this section...
"Round to Nearest Rounding Mode" on page 10-42
"Denormal Numbers" on page 10-43
"Exception Handling" on page 10-43
"Relative Accuracy and ULP Considerations" on page 10-44

Native floating-point technology can generate HDL code from your floating-point design. Floating-point designs have better precision, higher dynamic range, and a shorter development cycle than fixed-point designs. If your design has complex math and trigonometric operations, use native floating-point technology. HDL Coder generates code that complies with the IEEE-754 standard of floating-point arithmetic. HDL Coder native floating-point supports:

- Round to nearest rounding mode
- Denormal numbers
- Exceptions such as NaN (Not a Number), Inf, and Zero
- Customization of ULP (Units in the Last Place) and relative accuracy

Round to Nearest Rounding Mode

HDL Coder native floating-point uses the round to nearest even rounding mode. This mode resolves all ties by rounding to the nearest even digit.

This rounding method requires at least three trailing bits after the 23 bits of the mantissa. The MSB is called Guard bit, the middle bit is called the Round bit, and the LSB is called the Sticky bit. The table shows the rounding action that HDL Coder performs based on different values of the three trailing bits. `x` denotes a *don't care* value and can take either a 0 or a 1.

Rounding bits	Rounding Action
`0xx`	No action performed.
`100`	A tie. If the mantissa bit that precedes the Guard bit is a 1, round up, otherwise no action is performed.
`101`	Round up.

Rounding bits	Rounding Action
11x	Round up.

Denormal Numbers

Denormal numbers are numbers that have an exponent field equal to zero and a nonzero mantissa field. The leading bit of the mantissa is zero.

$$value = (-1)^{sign} * (0 + \sum_{i=1}^{23} b_{23-i} 2^{-i}) * 2^{-126}$$

Denormal numbers have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The presence of denormal numbers indicates loss of significant digits that can accumulate over subsequent operations and eventually result in unexpected values.

The logic to handle denormal numbers involves counting the number of leading zeros and performing a left shift operation to obtain the normalized representation. Addition of this logic increases the area footprint on the target device and can affect the timing of your design.

When using native floating-point support, you can specify whether you want HDL Coder to handle denormal numbers in your design. By default, HDL Coder does not check for denormal numbers, which saves area on the target platform.

Exception Handling

If you perform operations such as division by zero or compute the logarithm of a negative number, HDL Coder detects and reports exceptions. The table summarizes the mapping from the encoding of a floating-point number to the value of the number for various kinds of exceptions. x denotes a *don't care* value and can take either a 0 or a 1.

Sign	Exponent	Significand	Value	Description
x	0xFF	0x00000000	$value = (-1)^S \infty$	Infinity
x	0xFF	A nonzero value	$value = NaN$	Not a Number
x	0x00	0x00000000	$value = 0$	Zero

Sign	Exponent	Significand	Value	Description
x	0x00	A nonzero value	$value = (-1)^{sign} * (0 + \sum_{i=1}^{23} b_{23-i} 2^{-i}$	Denormal
x	0x00 < E < 0xFF	x	$value = (-1)^{sign} * (1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}$	Normal

Relative Accuracy and ULP Considerations

The representation of infinitely real numbers with a finite number of bits requires an approximation. This approximation can result in rounding errors in floating-point computation. To measure the rounding errors, the floating-point standard uses relative error and ULP (Units in the Last Place) error.

By using the floating-point tolerance check setting, you can specify whether to check for floating-point tolerance based on relative error or ULP error. You can then enter custom tolerance values. For more information, see `FPToleranceStrategy` and `FPToleranceValue`.

ULP

If the exponent range is not upper-bounded, Units in Last Place (ULP) of a floating-point number x is the distance between two closest straddling floating-point numbers a and b nearest to x. The IEEE-754 standard requires that you correctly round the result of an elementary arithmetic operation such as addition, multiplication, and division. A correctly rounded result means that the rounded result is within 0.5ULP of the exact result.

An ULP of one means adding a 1 to the decimal value of the number. The table shows the approximation of pi to nine decimal digits and how the ULP of one changes the approximate value.

Floating-point number	Value in decimal	IEEE-754 representation	ULP
3.141592741	1078530011	0|10000000| 10010010000111111011011	0
3.141592979	1078530012	0|10000000| 10010010000111111011100	1

The gap between two consecutively representable floating-point numbers varies according to magnitude.

Floating-point number	Value in decimal	IEEE-754 representation	ULP
1234567	1234613304	0|10010011| 00101101011010000111000	0
1234567.125	1234613305	0|10010011| 00101101011010000111001	1

For the ULP values of the floating-point operators that HDL Coder supports, see "Operators and Simulink Blocks Supported for Native Floating-Point" on page 10-64 .

Relative Error

Relative error measures the difference between a floating-point number and the approximation of the real number. Relative error returns the distance from 1.0 to the next larger single-precision number. This table shows how the real value of a number changes with the relative accuracy.

Floating-point number	Value in decimal	IEEE-754 representation	ULP	Relative error
8388608	1258291200	0|10010110| 00000000000000000000000	0	1
8388607	1258291198	0|10010101| 11111111111111111111110	1	0.5
1	1065353216	0|01111111| 00000000000000000000000	0	1.1920929e-07
2	1073741824	0|10000000| 00000000000000000000000	1	2.3841858e-07

The magnitude of the relative error depends on the real value of the floating-point number.

In MATLAB, the `eps` function measures the relative accuracy of the floating-point number. For more information, see `eps`.

See Also

`hdlcoder.createFloatingPointTargetConfig` | `NativeFloatingPoint`

Related Examples

-

More About

- “HDL Coder Native Floating-Point Support” on page 10-37
- “Operators and Simulink Blocks Supported for Native Floating-Point” on page 10-64
- “Limitations of Native Floating-Point Support” on page 10-69
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-52

Latency Customization with Native Floating-Point

In this section...

HDL Coder native floating-point technology can generate HDL code from your floating-point design. Floating-point designs have better precision, higher dynamic range, and a shorter development cycle than fixed-point designs. If your design has complex math and trigonometric operations, use native floating-point technology.

When generating code with native floating-point, you can customize the latency of the blocks in your design. Using custom settings, you can optimize your design implementation on the target FPGA device for area and speed. You can customize the latency using:

- Latency Strategy setting: Specify whether to map your Simulink model to maximum, minimum, or zero latency of the floating-point operator.
- Oversampling factor: The design operates at a faster clock-rate and absorbs the clock-rate pipelines with the latency of the floating-point operator.
- Delay blocks in the model: If your Simulink model has a latency, HDL Coder can absorb some or all of the latency with the native floating-point implementation.

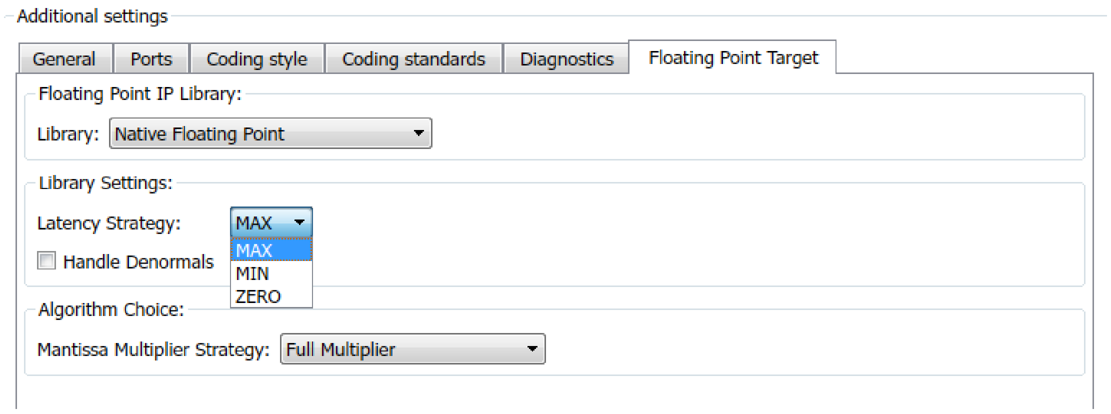
Specify the Latency Strategy Setting

When generating code with HDL Coder native floating-point, you can specify whether to use zero, minimum, or maximum latency for the floating-point operators. If you specify maximum or minimum latency, you can get the latency information of the floating-point operator from the generated model. For certain blocks in your design, you can optionally specify a custom latency strategy setting that is different from the latency strategy setting for the model.

Specify Latency Strategy for a Model

To specify this setting from the Configuration Parameters dialog box:

1. In the **HDL Code Generation** > **Global Settings** > **Floating Point Target** tab, for **Library**, select `Native Floating Point`.



2 For **Latency Strategy**, specify `MAX`, `MIN`, or `ZERO`.

To specify this setting from the command line:

1 Create a `hdlcoder.FloatingPointTargetConfig` object: Use the `hdlcoder.createFloatingPointTargetConfig` function to create a native floating-point configuration object.

```
nfpconfig = hdlcoder.createFloatingPointTargetConfig('NATIVEFLOATINGPOINT');
hdlset_param('sfir_single', 'FloatingPointTargetConfiguration', nfpconfig);
```

2 Specify the latency strategy: If you have denormal numbers in your design, you can configure the native floating-point library to handle denormals.

```
nfpconfig.LibrarySettings.LatencyStrategy = 'MAX'
```

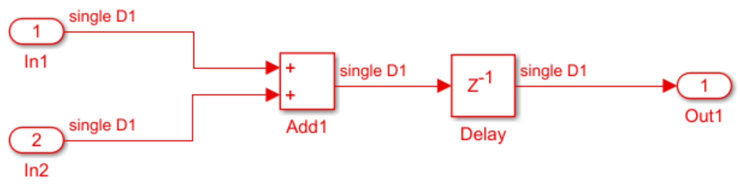
Specify Latency Strategy for a Block

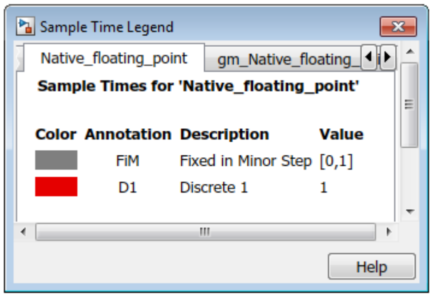
For certain blocks in your Simulink model, you can selectively customize the latency strategy. To learn how to specify the latency strategy for a block, see "LatencyStrategy" on page 12-16.

Oversampling Factor

When you design the blocks in your Simulink model at the data rate, specify an **Oversampling factor** greater than one. The blocks now operate at a faster clock rate instead of the data rate. The oversampling factor inserts pipeline registers at the clock rate, which improves clock frequency and reduces area usage. To learn more about clock-rate pipelining, see “Clock-Rate Pipelining” on page 15-66.

Consider this Add block with inputs of `Single` data type, and a latency of 1 in your Simulink model. The model is operating at a sample time of 1. The **Oversampling factor** of the model is set to `40`.

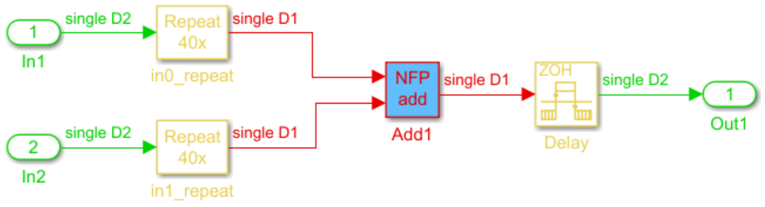


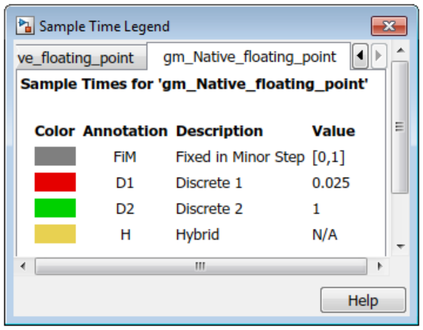


Sample Times for 'Native_floating_point'

Color	Annotation	Description	Value
▨	FiM	Fixed in Minor Step	[0,1]
🟥	D1	Discrete 1	1

After HDL code generation, the generated model shows an **NFP add** block. This block corresponds to HDL Coder implementation of the floating-point add operation. The **NFP add** block is operating at a clock rate that is 40 times faster than the Add block in your model. The **NFP add** block absorbed the Delay block in your Simulink model, which now operates at the clock rate. This implementation saves area by absorbing the additional latency, and improves timing by operating at the faster clock rate.





Sample Times for 'gm_Native_floating_point'

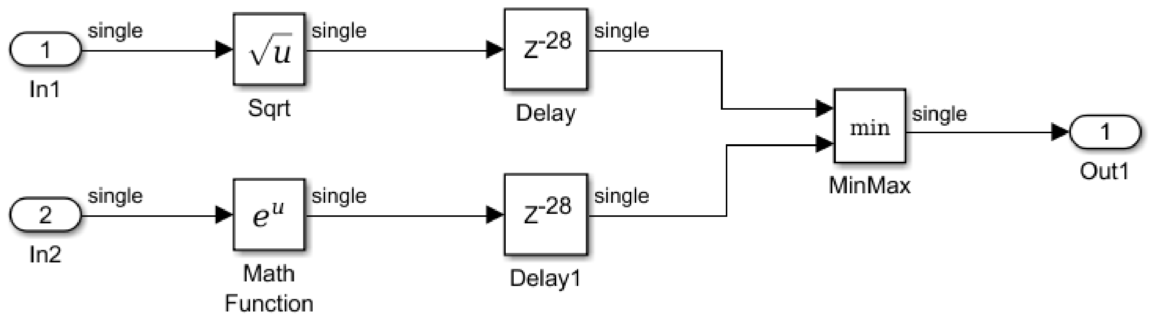
Color	Annotation	Description	Value
▨	FiM	Fixed in Minor Step	[0,1]
🟥	D1	Discrete 1	0.025
🟩	D2	Discrete 2	1
🟨	H	Hybrid	N/A

Delay Blocks in the Model

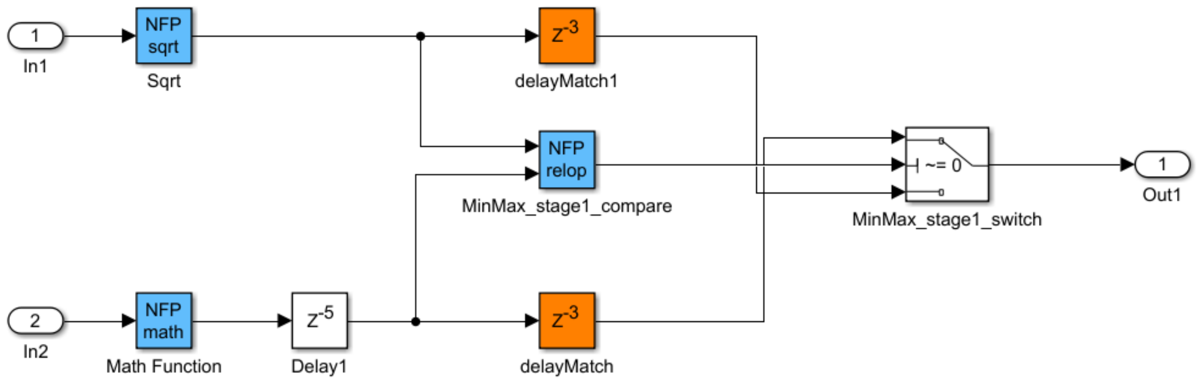
If your Simulink model has a latency, HDL Coder absorbs some or all of the latency with the native floating-point operator implementation.

If you have a Delay block in your design with **Delay length** less than or equal to the latency of the floating-point operator, HDL Coder absorbs the latency inside the operator. This latency absorption avoids some of the additional latency in your model.

Consider this Simulink model that compares the results of a square root and an exponential operation. The latency strategy setting uses the maximum latency.



When generating HDL code, the generated model shows how HDL Coder implemented the floating-point operators.



The **NFP Sqrt** block has a maximum latency of 28. This latency is equal to the **Delay length** of the Delay block at the output of your Simulink model. HDL Coder absorbs the Delay inside the **NFP Sqrt** block.

The **NFP math** block has a maximum latency of 23. This latency is less than the **Delay length** of the Delay block at the output of your Simulink model. HDL Coder absorbs a **Delay length** equal to the latency of the floating-point operator. A Delay block with the excess **Delay length** appears outside the operator.

To learn more about the supported operators and their maximum and minimum latency values, see "Operators and Simulink Blocks Supported for Native Floating-Point" on page 10-64.

See Also

`hdlcoder.createFloatingPointTargetConfig` | `NativeFloatingPoint`

Related Examples

•

More About

- "HDL Coder Native Floating-Point Support" on page 10-37
- "Generate Target-Independent HDL Code with Native Floating-Point" on page 10-52
- "Operators and Simulink Blocks Supported for Native Floating-Point" on page 10-64
- "Limitations of Native Floating-Point Support" on page 10-69

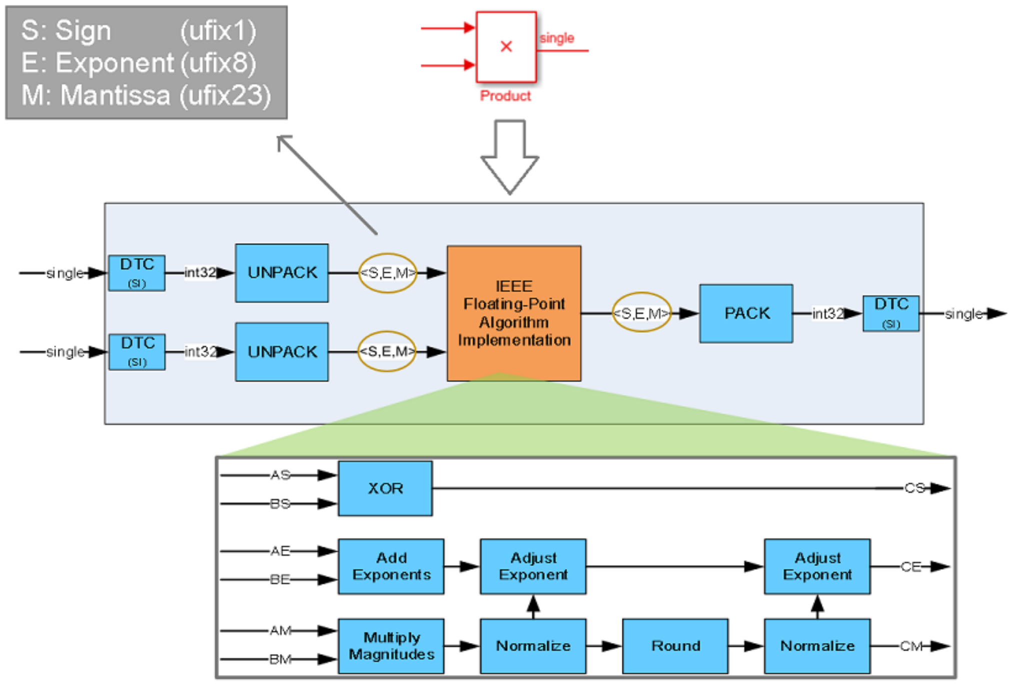
Generate Target-Independent HDL Code with Native Floating-Point

In this section...

HDL Coder native floating-point technology can generate target-independent HDL code from your floating-point design. You can synthesize your floating-point design on any generic FPGA or ASIC. Floating-point designs have better precision, higher dynamic range, and a shorter development cycle than fixed-point designs. If your design has complex math and trigonometric operations, use native floating-point technology.

How HDL Coder Generates Target-Independent HDL Code

This figure shows how HDL Coder generates code with the native floating-point technology.



The `Unpack` and `Pack` blocks convert the single-precision types to the sign, exponent, and mantissa. In the figure, *S*, *E*, and *M* represent the sign, exponent, and mantissa respectively. This interpretation is based on the IEEE-754 standard of floating-point arithmetic.

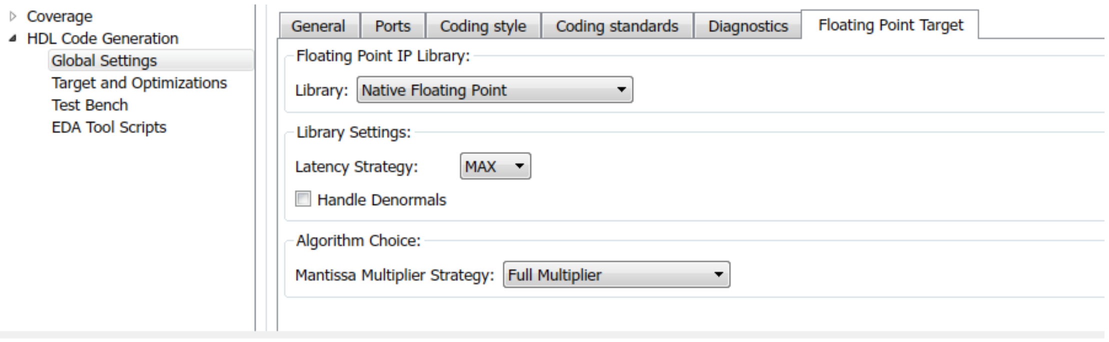
The **Floating-Point Algorithm Implementation** block performs computations on the *S*, *E*, and *M*. With this conversion, the generated HDL code is target-independent. You can deploy the design on any generic FPGA or an ASIC.

Generate Code

You can generate code in the Configuration Parameters dialog box or at the command line.

To specify the native floating-point settings and generate HDL code in the Configuration Parameters dialog box:

1 In the **HDL Code Generation** > **Global Settings** > **Floating Point Target** tab, for **Library**, specify `Native Floating Point`.



Floating Point IP Library:

Library: Native Floating Point

Library Settings:

Latency Strategy: MAX

☐ Handle Denormals

Algorithm Choice:

Mantissa Multiplier Strategy: Full Multiplier

2 Specify the **Latency Strategy** to map your design to maximum or minimum latency or no latency.

3 If you have denormal numbers in your design, select **Handle Denormals**. Denormal numbers are numbers that have an exponent field equal to zero and a nonzero mantissa field. See “Handle Denormals” on page 11-98.

4 If your design has Product blocks, to specify how you want HDL Coder to implement the multiplication operation, use the **Mantissa Multiplier Strategy**. See “Mantissa Multiplier Strategy” on page 11-99.

5 To share floating-point resources, on the **HDL Code Generation** > **Target and Optimizations** > **Resource Sharing** tab, make sure that you select **Floating-point IPs**. The number of blocks that get shared depends on the **SharingFactor** that you specify for the subsystem.

6 Click **Apply**. You can now generate HDL code from your Simulink model. (see “HDL Code Generation from a Simulink Model”)

To generate HDL code at the command line, use the:

- `NativeFloatingPoint` property: To generate code with the default settings for the native floating-point library, use the `makehdl` function. For example, to enable `NativeFloatingPoint` for an `sfir_single/symmetric_fir` subsystem and generate HDL code, enter:

  ```
  makehdl ('sfir_single/symmetric_sfir','NativeFloatingPoint','on')
  ```

- `hdlcoder.createFloatingPointTargetConfig` function: You can use this function to create an `hdlcoder.FloatingPointTargetConfig` object for the native floating-point library.

  ```
  nfpconfig = hdlcoder.createFloatingPointTargetConfig('NATIVEFLOATINGPOINT');
  hdlset_param('sfir_single', 'FloatingPointTargetConfiguration', nfpconfig);
  ```

  Optionally, you can specify the latency strategy and whether you want HDL Coder to handle denormal numbers in your design:

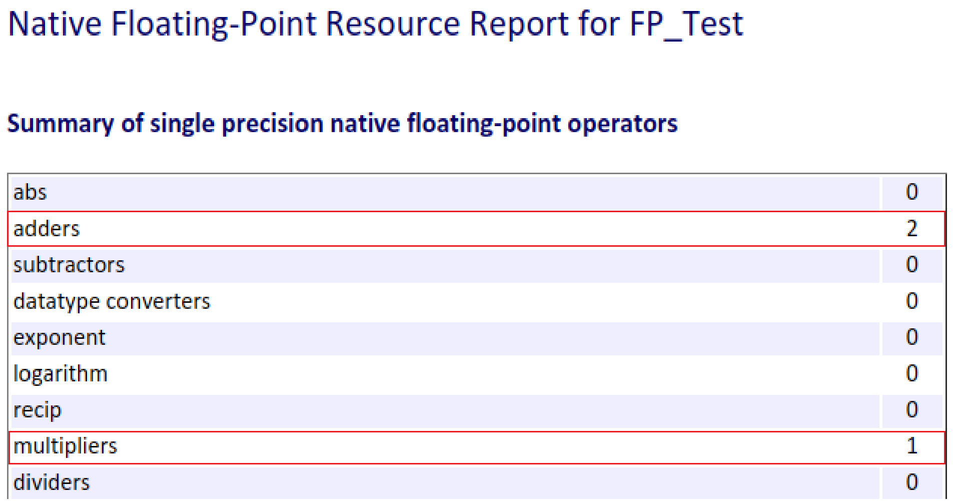
  ```
  nfpconfig.LibrarySettings.HandleDenormals = 'on';
  nfpconfig.LibrarySettings.LatencyStrategy = 'MAX';
  ```

To learn how you can verify the generated code, see “Verify the Generated Code from Native Floating-Point” on page 10-59.

View Code Generation Report

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 enable the reports, in the Configuration Parameters dialog box, on the **HDL Code Generation** pane, enable **Generate resource utilization report** and **Generate optimization report**. See also “Create and Use Code Generation Reports” on page 16-2.

To see the list of native floating-point operators that HDL Coder supports and the floating-point operators to which your Simulink blocks mapped to, in the Code Generation Report, select **Native Floating-Point Resource Report**.

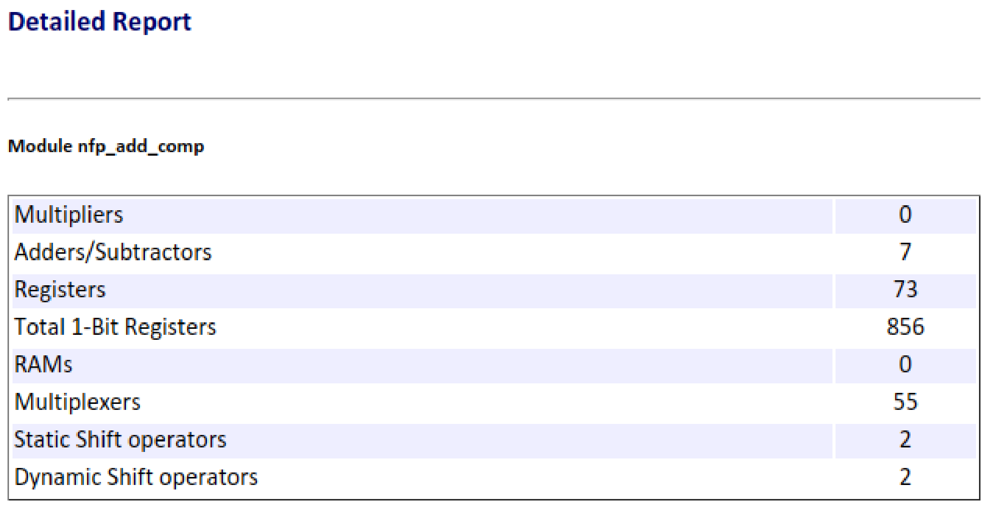


Native Floating-Point Resource Report for FP_Test

Summary of single precision native floating-point operators

abs	0
adders	2
subtractors	0
datatype converters	0
exponent	0
logarithm	0
recip	0
multipliers	1
dividers	0

A detailed report shows the various resources that the floating-point blocks use on the target device that you specify. To learn more about the Resource Utilization report, see “Resource Utilization Report” on page 16-4.



Detailed Report

Module nfp_add_comp

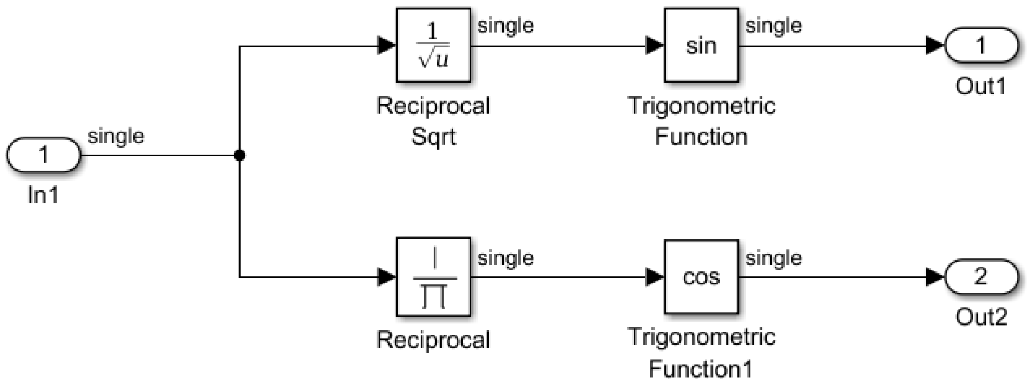
Multipliers	0
Adders/Subtractors	7
Registers	73
Total 1-Bit Registers	856
RAMs	0
Multiplexers	55
Static Shift operators	2
Dynamic Shift operators	2

To see the native floating-point settings that you applied to the model and whether HDL Coder successfully generated HDL code, in the Code Generation Report, select **Target Code Generation**. To learn more about the Optimization report, see “Optimization Report” on page 16-9.

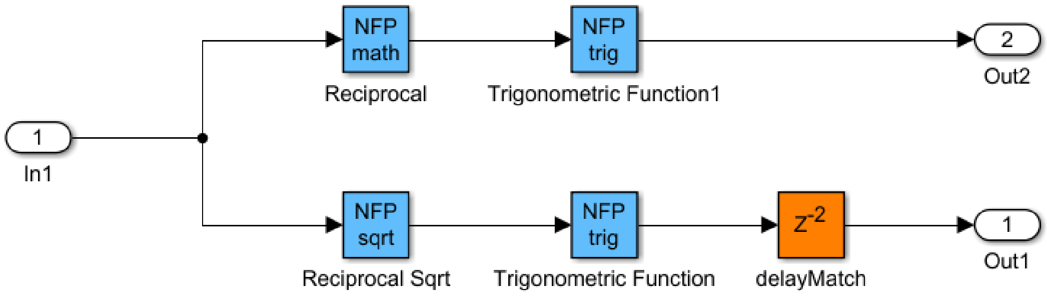
Analyze Results

If you select maximum or minimum latency for the latency strategy, you can get the latency information of the floating-point operator from the generated model.

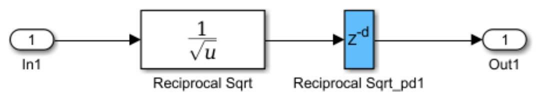
This Simulink model has a `Single` input and has **Latency Strategy** set to `MIN`.

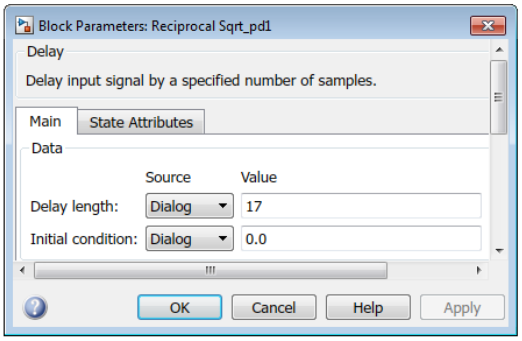


After HDL code generation, the generated model shows the native floating-point operators corresponding to the blocks in your Simulink model.



To get the latency information of the floating-point operator, double-click that **NFP** block. For example, if you double-click the **NFP sqrt** block, you can get the latency of the floating-point reciprocal square-root implementation. Here, the latency is 17.




Block Parameters: Reciprocal Sqrt_pd1

Delay

Delay input signal by a specified number of samples.

Main | State Attributes

Data

	Source	Value
Delay length:	Dialog	17
Initial condition:	Dialog	0.0

To learn more about the generated model, see “Generated Model and Validation Model” on page 14-2.

See Also

`hdlcoder.createFloatingPointTargetConfig` | `NativeFloatingPoint`

Related Examples

•

More About

- “HDL Coder Native Floating-Point Support” on page 10-37
- “Latency Customization with Native Floating-Point” on page 10-47
- “Verify the Generated Code from Native Floating-Point” on page 10-59
- “Operators and Simulink Blocks Supported for Native Floating-Point” on page 10-64

Verify the Generated Code from Native Floating-Point

In this section...

HDL Coder native floating-point technology can generate target-independent HDL code from your floating-point design. You can synthesize your floating-point design on any generic FPGA or ASIC. Floating-point designs have better precision, higher dynamic range, and a shorter development cycle than fixed-point designs. If your design has complex math and trigonometric operations, use native floating-point technology.

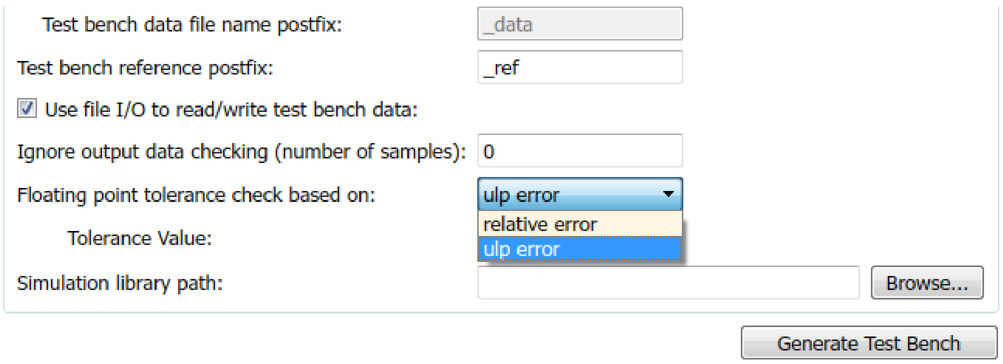
When representing infinitely real numbers with a finite number of bits, there can be rounding errors with the correct rounding range of values that the IEEE-754 standard specifies. To measure the rounding errors, you can specify the floating-point tolerance check based on `relative error` or `ulp error`. For more information about these rounding errors, see "Relative Accuracy and ULP Considerations" on page 10-44.

Specify the Tolerance Strategy

Before generating the testbench, specify the floating-point tolerance check for verifying the generated code.

To specify the tolerance check in the Configuration Parameters dialog box:

1 In the **HDL Code Generation** > **Testbench** pane, for **Floating point tolerance check based on**, specify `relative error` or `ulp error`.


Test bench data file name postfix: _data
Test bench reference postfix: _ref
☑ Use file I/O to read/write test bench data:
Ignore output data checking (number of samples): 0
Floating point tolerance check based on: ulp error
Tolerance Value:
Simulation library path:

2 Enter the **Tolerance Value** and click **Apply**. If you choose `relative error`, the default is a tolerance value of `1e-07`. If you choose `ulp error`, the default tolerance value is zero. To learn more, see "Numerical Considerations with Native Floating-Point" on page 10-42.

To specify the tolerance strategy at the command-line, use:

1 Specify the floating point tolerance check setting by using `FPToleranceStrategy`.

```
hdlset_param('sfir_single', 'FPToleranceStrategy', 'Relative');  % check for floati
hdlset_param('sfir_single', 'FPToleranceStrategy', 'ULP');       % check for floati
```

2 Based on the `FPToleranceStrategy` setting, enter the tolerance value by using `FPToleranceValue`.

```
hdlset_param('FP_test_16a', 'FPToleranceValue', 1e-06);  % if using relative error,
hdlset_param('FP_test_16a', 'FPToleranceValue', 1);      % if using ULP error, ente
```

Verify the Generated Code with HDL Test Bench

To generate an HDL test bench for verifying the generated code:

1 In the Configuration Parameters dialog box, on the **HDL Code Generation** > **Test Bench** pane, in the Test Bench Generation Output section, select **HDL test bench**.

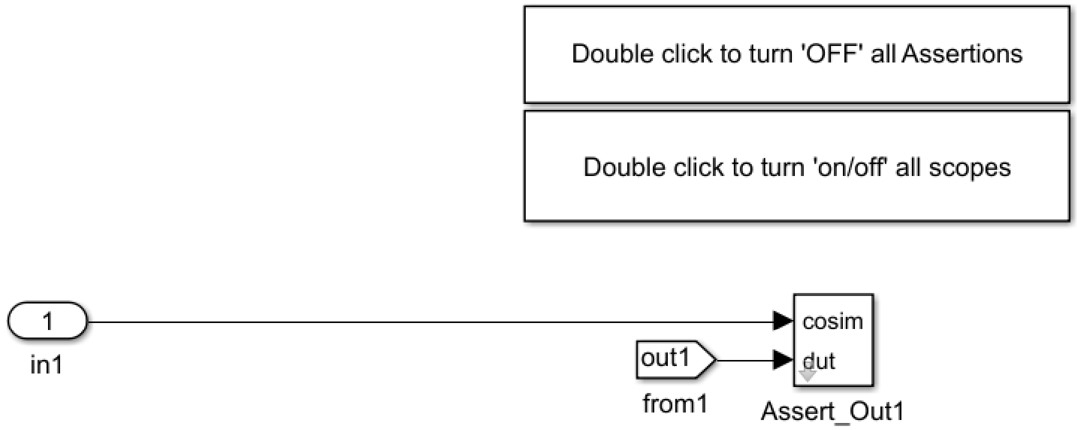
2 In the Configuration section, make sure that **Use file I/O to read/write test bench data** is enabled. To generate a test bench that uses constants instead of file I/O, clear **Use file I/O to read/write test bench data**.
3 Click **Apply**, and then click **Generate Test Bench**.

To learn more about how HDL test bench generation works, see “Test Bench Generation” on page 6-6.

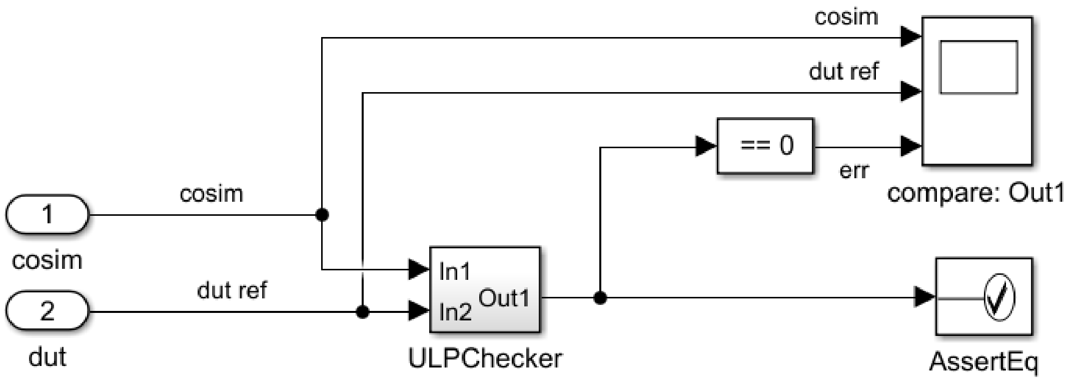
Verify the Generated Code with Cosimulation

To generate a cosimulation model for verifying the generated code:

1 In the Configuration Parameters dialog box, on the **HDL Code Generation** > **Test Bench** pane, for **Cosimulation model for use with**, select the cosimulation tool.
2 Click **Apply**, and then click **Generate Test Bench**.
3 After test bench generation, save the cosimulation model. In the model, double-click the `Compare` subsystem.

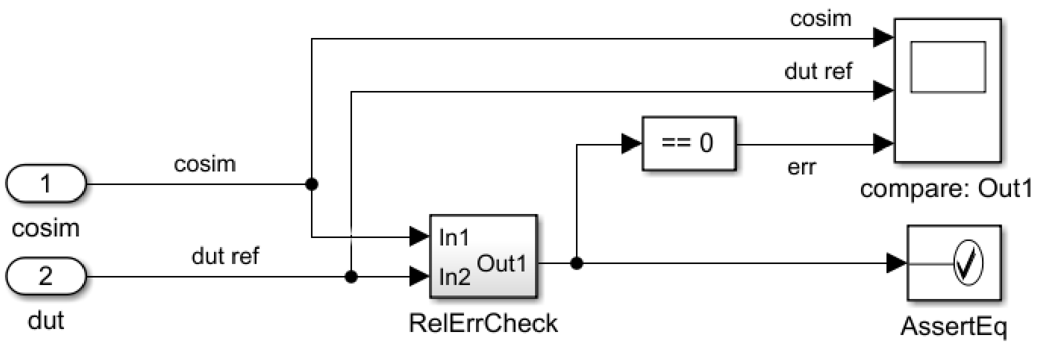

Double click to turn 'OFF' all Assertions
Double click to turn 'on/off' all scopes
1
in1
out1
from1
cosim
dut
Assert_Out1

4 If you double-click the `Assert_Out1` block, the block parameters show the **ToleranceValue** that you specify.
5 To look inside the `Assert_Out1` block, click the mask. If you specify the floating-point tolerance check based on `ulp error`, the model shows a `ULPChecker` block.



The `ULPChecker` has a MATLAB Function block that shows how HDL Coder accounts for the ULP error when checking for numerical accuracy.

If you specify the floating-point tolerance check based on `relative error`, the model shows a `RelErrCheck` block.



`RelerrCheck` has a MATLAB Function block that shows how HDL Coder accounts for the relative error when checking for numerical accuracy.

6 In the Simulink Editor for the model, start simulation. At the end of cosimulation, check the `compare: Out1` scope.

The scope compares the difference between the result signal from the cosimulation block and the reference signal from the DUT.

See also “Generate a Cosimulation Model” on page 18-40.

Limitation

When verifying the generated code, constructs that use IEEE standards prior to VHDL-2008 are not supported with native floating-point.

See Also

`hdlcoder.createFloatingPointTargetConfig` | `NativeFloatingPoint`

Related Examples

•

More About

- “HDL Coder Native Floating-Point Support” on page 10-37
- “Numerical Considerations with Native Floating-Point” on page 10-42
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-52
- “Operators and Simulink Blocks Supported for Native Floating-Point” on page 10-64

Operators and Simulink Blocks Supported for Native Floating-Point

In this section...
“Supported Native Floating-Point Operators” on page 10-64
“Supported Simulink Blocks” on page 10-65

HDL Coder native floating-point technology can generate target-independent HDL code from your floating-point design. You can synthesize your floating-point design on any generic FPGA or ASIC. Floating-point designs have better precision, higher dynamic range, and a shorter development cycle than fixed-point designs. If your design has complex math and trigonometric operations, use native floating-point technology.

HDL Coder supports math and trigonometric functions, and several Simulink blocks with native floating-point technology. You can customize the latency of these operators.

Supported Native Floating-Point Operators

The table shows a list of floating-point operators that HDL Coder supports, the maximum and minimum latency, and the units in the last place (ulp) error.

Simulink Blocks	Minimum Output Latency	Maximum Output Latency	Units in the Last Place (ULP) error
Absolute value	0	0	0
Signum	0	0	0
Unary Minus	0	0	0
Minimum/maximum	3	3	0
Relational Operator	3	3	0
Rounding	5	5	0
Data Type Conversion	6	6	0
Add	7	12	0
Subtract	7	12	0
Multiply	6	8	0

Simulink Blocks	Minimum Output Latency	Maximum Output Latency	Units in the Last Place (ULP) error
Divide	17	32	0
Reciprocal	16	31	0
Reciprocal square root	17	17	2
Square root	16	28	0
Exponential	23	23	3
Power of two	1	2	0
Natural logarithm	20	20	3
Modulo	15	25	0
Remainder	15	24	0
Sine	27	27	2
Cosine	27	27	2
Arctangent	36	36	2
Arctangent 2 (ArcTangent (input1 / input2))	42	42	5

Supported Simulink Blocks

HDL Coder supports several Simulink blocks including math and trigonometric blocks. The **HDL Floating Point Operations** library contains blocks that are configured for HDL code generation in the native floating-point mode.

In the **Math Operations** library, these blocks are supported:

- Sum
- Add
- Subtract
- Bias
- Gain
- Product
- Divide
- Abs

- Unary Minus
- MinMax
- Sum of Elements
- Product of Elements
- Dot Product
- Sqrt
- Reciprocal Sqrt
- Assignment
- Vector Concatenate
- Reshape
- Complex to Real-Imag
- Real-Imag to Complex
- Math Function where **Function** can be:
  - `reciprocal`
  - `log`
  - `exp`
  - `rem`
  - `mod`
- Trigonometric Function where **Function** can be:
  - `sin`
  - `cos`
  - `sincos`
  - `cos + jsin`
  - `atan`
  - `atan2`

The table shows the list of supported blocks in other block libraries.

Block Library	Supported blocks
Discrete	Zero Order Hold and the set of delay blocks including Integer Delay and Tapped Delay.

Block Library	Supported blocks
HDL Operations	Multiply-Add, HDL Counter, Serializer1D, Deserializer1D, and all RAM blocks are supported. The RAM blocks include Single Port RAM, Simple Dual Port RAM, Dual Port RAM, and Dual Rate Dual Port RAM.
HDL Subsystems	All blocks are supported. This library includes the State Control block and subsystems that use enable and reset ports with the State Control block.
Logic and Bit Operations	Compare To Constant and Compare To Zero
Lookup Tables	Direct Lookup Table (n-D) block is supported.
Model Verification	All blocks are supported. This library includes blocks such as Assertion and Check Dynamic Range.
Model-Wide Utilities	All blocks are supported. This library includes the DocBlock and Model Info blocks.
Ports & Subsystems	Enable, reset, input, and output ports, and model references, and subsystem blocks are supported.
Signal Attributes	All blocks are supported. This library includes blocks such as Bus to Vector and Data Type Propagation.
Signal Routing	All blocks are supported. This library includes blocks such as Mux and Bus Selector.
Sources	Inport, Constant, and Ground blocks.
Sinks	All blocks are supported. This library includes blocks such as Display, To File, and Spectrum Analyzer.

See Also

`hdlcoder.createFloatingPointTargetConfig` | `NativeFloatingPoint`

Related Examples

-

More About

- “HDL Coder Native Floating-Point Support” on page 10-37
- “Limitations of Native Floating-Point Support” on page 10-69
- “Numerical Considerations with Native Floating-Point” on page 10-42
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-52

Limitations of Native Floating-Point Support

In this section...

HDL Coder native floating-point technology can generate target-independent HDL code from your floating-point design. You can synthesize your design on any generic FPGA or ASIC. The code generator supports complex math and trigonometric operations, and several Simulink blocks in native floating-point mode. These blocks compute results of mathematical operations according to the IEEE-754 standard of floating-point arithmetic.

To generate HDL code in native floating-point mode, use:

- Single data types. Double data types are not supported.
- Discrete sample times. Blocks operating at a continuous sample time are not supported.

Native floating-point support has these restrictions:

- Numerical accuracy: Certain blocks that are supported in native floating-point mode can introduce a ULP (units in the last place) mismatch or produce a different numeric result in some corner cases.
- Simulink block support: The code generator does not support certain blocks such as the MATLAB Function block and the MATLAB System block in native floating-point mode.
- HDL optimizations: Certain optimizations, such as distributed pipelining, are not supported in native floating-point mode.

Numerical Accuracy Restrictions

For certain floating-point input values, some blocks can introduce a ULP mismatch, or produce simulation results that vary from the MATLAB simulation results. To see the difference in results, before you generate code, enable generation of the validation model. In the Configuration Parameters dialog box, on the **HDL Code Generation** pane, select the **Generate validation model** check box.

- If you perform computations that involve complex numbers and an exception such as `Inf` or `NaN`, the HDL simulation result with native floating point can potentially vary from the Simulink simulation result. For example, if you multiply a complex input with `Inf`, the Simulink simulation result is `Infi` whereas the HDL simulation result is `NaN+Infi`.
- If you compute the square root or logarithm of a negative number, the HDL simulation result with native floating point is `0`. This result matches the simulation result when you verify the design with a SystemVerilog DPI test bench. In Simulink, the result obtained is `NaN`. According to the IEEE-754 standard, if you compute the square root or logarithm of a negative number, the result is that number itself.
- If the input to the Direct Lookup Table (n-D) is of `single` data type, but the elements of the table use a smaller data type such as `uint8`, the generated HDL code can be potentially incorrect. To obtain accurate HDL simulation results, use the same data type for the input signal and the elements of the lookup table.
- If you use the Cosine block with the inputs `-7.729179E28` or `7.729179E28`, the generated HDL code has a ULP of 4. For all other inputs, the ULP is 2.
- When you use a Math Function block to compute `mod(a,b)` or `rem(a,b)`, where `a` is the dividend and `b` is the divisor, the simulation result in native floating-point point mode varies from the MATLAB simulation result in these cases:
  - If b is integer and $\frac{a}{b} > 2^{32}$, the simulation result in native floating-point mode is zero. For such significant difference in magnitude between the numbers `a` and `b`, this implementation saves area on the target FPGA device.
  - If $\frac{a}{b}$ is close to 2^{23}, the simulation result in native floating-point mode can potentially vary from the MATLAB simulation results.

Simulink Block Restrictions

In native floating-point mode, the code generator does not support these blocks or block architectures:

- Biquad Filter.
- Switch block with input to the control port as a floating-point type.
- Sum of Elements with complex input types.
- MATLAB Function and MATLAB System blocks.

- Dot Product in complex mode with **Architecture** as `Tree` or `Linear`.
- Lookup table blocks other than Direct Lookup table (n-D).
- Discrete FIR Filter with **Architecture** other than `Fully Parallel`.
- Dead Zone and Dead Zone Dynamic.
- Polar to Cartesian.
- For the Data Type Conversion block:
  - `Stored Integer (SI)` mode for **Input and output to have equal** setting is not supported.
  - The **Saturate on integer overflow** check box must be left cleared.

HDL Optimization Restrictions

The code generator does not support the distributed pipelining optimization in native floating-point mode.

See Also
`hdlcoder.createFloatingPointTargetConfig` | `NativeFloatingPoint`

Related Examples

•

More About

- "HDL Coder Native Floating-Point Support" on page 10-37
- "Numerical Considerations with Native Floating-Point" on page 10-42
- "Operators and Simulink Blocks Supported for Native Floating-Point" on page 10-64
- "Verify the Generated Code from Native Floating-Point" on page 10-59

Supported Data Types and Scope

Supported Data Types

HDL Coder supports the following subset of MATLAB data types.

Types	Supported Data Types	Restrictions
Integer	• `uint8`, `uint16`, `uint32`, `uint64` • `int8`, `int16`, `int32`, `int64`	In Simulink, MATLAB Function block ports must use numeric types `sfix64` or `ufix64` for 64-bit data.
Real	• `double` • `single`	HDL code generated with `double` or `single` data types can be used for simulation, but is not synthesizable. To generate synthesizable HDL code, use: • HDL Coder native floating-point when you have Single data types. You can deploy the generated code on any generic ASIC or FPGA. To learn more, see “HDL Coder Native Floating-Point Support” on page 10-37. • FPGA floating-point target libraries when you have Single or Double data types. You can map the Simulink model to an Altera or Xilinx FPGA. To learn more, see “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 22-26.
Character	`char`	
Logical	`logical`	
Fixed point	• Scaled (binary point only) fixed-point numbers	Fixed-point numbers with slope (not equal to 1.0) and bias (not equal to 0.0) are not supported.

Types	Supported Data Types	Restrictions
	• Custom integers (zero binary point)	Maximum word size for fixed-point numbers is 128 bits.
Vectors	• unordered `{N}` • row `{1, N}` • column `{N, 1}`	The maximum number of vector elements allowed is 2^32. Before a variable is subscripted, it must be fully defined.
Matrices	`{N, M}`	Matrices are supported in the body of the design algorithm, but are not supported as inputs to the top-level design function. Do not use matrices in the testbench.
Structures	`struct`	Arrays of structures are not supported. For the FPGA Turnkey and IP Core Generation workflows, structures are supported in the body of the design algorithm, but are not supported as inputs to the top-level design function.
Enumerations	`enumeration`	Enumeration values must be monotonically increasing. If your target language is Verilog, all enumeration member names must be unique within the design. Enumerations at the top-level DUT ports are not supported with the following workflows or verification methods: • IP Core Generation workflow • FPGA Turnkey workflow • FPGA-in-the-Loop • HDL Cosimulation

Unsupported Data Types

In the current release, the following data types are not supported:

- Cell array
- `Inf`

Scope for Variables

Global variables are not supported for HDL code generation.

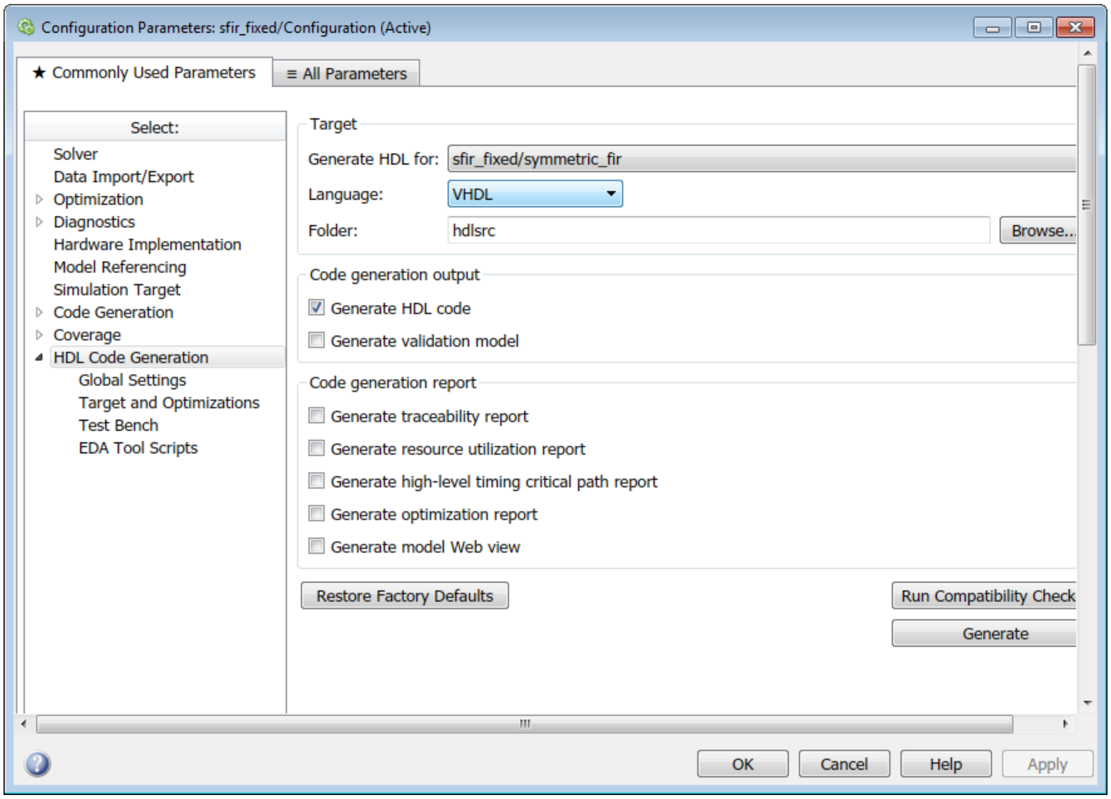
Code Generation Options in the HDL Coder Dialog Boxes

Set HDL Code Generation Options

In this section...

HDL Code Generation Options in the Configuration Parameters Dialog Box

The following figure shows the top-level **HDL Code Generation** pane in the Configuration Parameters dialog box. To open this dialog box, select **Simulation** > **Model Configuration Parameters** in the Simulink window. Then select **HDL Code Generation** from the list on the left.


Configuration Parameters: sfir_fixed/Configuration (Active)

Target

Generate HDL for: sfir_fixed/symmetric_fir

Language: VHDL

Folder: hdlsrc

Code generation output

- [✓] Generate HDL code
- [ ] Generate validation model

Code generation report

- [ ] Generate traceability report
- [ ] Generate resource utilization report
- [ ] Generate high-level timing critical path report
- [ ] Generate optimization report
- [ ] Generate model Web view

Note: When the **HDL Code Generation** pane of the Configuration Parameters dialog box appears, clicking the **Help** button displays general help for the Configuration Parameters dialog box.

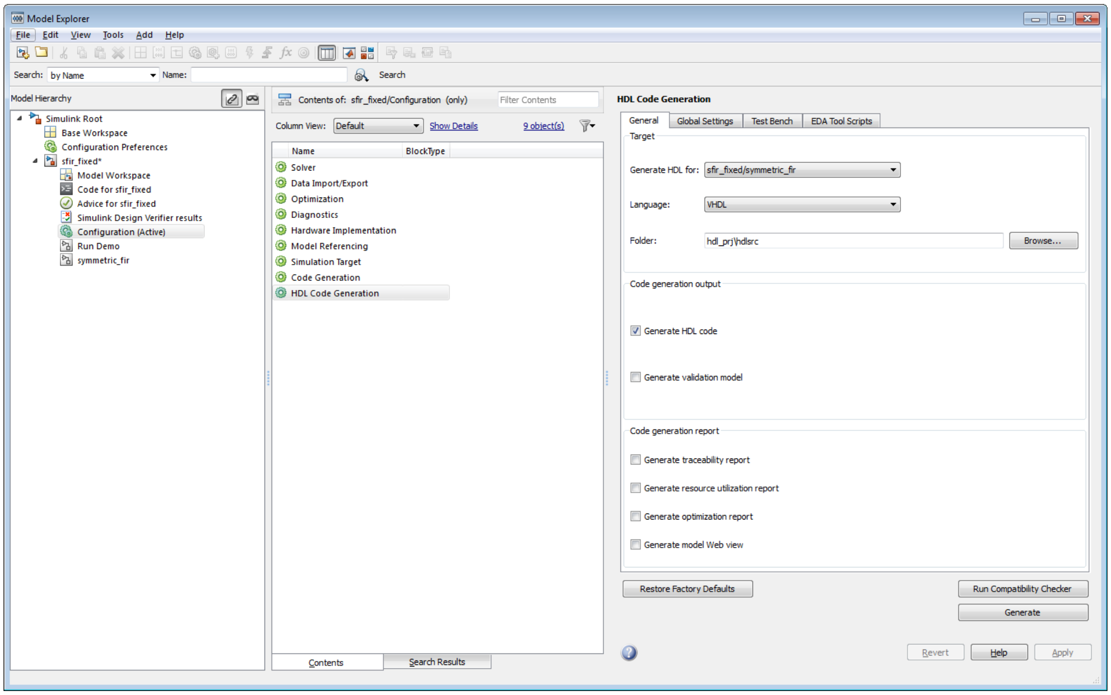
HDL Code Generation Options in the Model Explorer

The following figure shows the top-level **HDL Code Generation** pane as displayed in the Contents pane of the Model Explorer.

To view this dialog box:

1. Open the Model Explorer.
2. Select your model's active configuration set in the **Model Hierarchy** tree on the left.

3 Select **HDL Code Generation** from the list in the Contents pane.



Generate HDL for: sfir_fixed/symmetric_fir

Language: VHDL

Folder: hdl_prj\hdlsrc

- [✓] Generate HDL code
- [ ] Generate validation model
- [ ] Generate traceability report
- [ ] Generate resource utilization report
- [ ] Generate optimization report
- [ ] Generate model Web view

Code Menu

The **Code** > **HDL Code** submenu provides shortcuts to the HDL code generation options. You can also use this submenu to initiate code generation.

Options include:

- **HDL Workflow Advisor**: Open the HDL Workflow Advisor.
- **Options**: Open the **HDL Code Generation** pane in the Configuration Parameters dialog box.
- **Generate HDL**: Initiate HDL code generation; equivalent to the **Generate** button in the Configuration Parameters dialog box or Model Explorer.
- **Generate Test Bench**: Initiate test bench code generation; equivalent to the **Generate Test Bench** button in the Configuration Parameters dialog box or Model

Explorer. If you do not select a subsystem in the **Generate HDL for** menu, the **Generate Test Bench** menu option is not available.

- **Add HDL Coder Configuration to Model** or **Remove HDL Coder Configuration from Model**: The *HDL configuration component* is internal data 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 use the **HDL Code Generation** pane to set HDL code generation options. If you need to add or remove the HDL Code Generation configuration component to or from a model, use this option to do so. For more information, see "Add or Remove the HDL Configuration Component" on page 16-51.

HDL Code Options in the Block Context Menu

When you right-click a block that HDL Coder supports, the context menu for the block includes an **HDL Code** submenu. The coder enables items in the submenu according to:

- The block type: for subsystems, the menu enables some options that are specific to subsystems.
- Whether or not code and traceability information has been generated for the block or subsystem.

The following summary describes the **HDL Code** submenu options.

Option	Description	Availability
Check Subsystem Compatibility	Runs the HDL compatibility checker (`checkhdl`) on the subsystem.	Available only for subsystems.
Generate HDL for Subsystem	Runs the HDL code generator (`makehdl`) and generates code for the subsystem.	Available only for subsystems.
HDL Coder Properties	Opens the Configuration Parameters dialog box, with the top-level HDL Code Generation pane selected.	Available for blocks or subsystems.
HDL Block Properties	Opens a block properties dialog box for the block or subsystem. See "Set and View HDL Block	Available for blocks or subsystems.

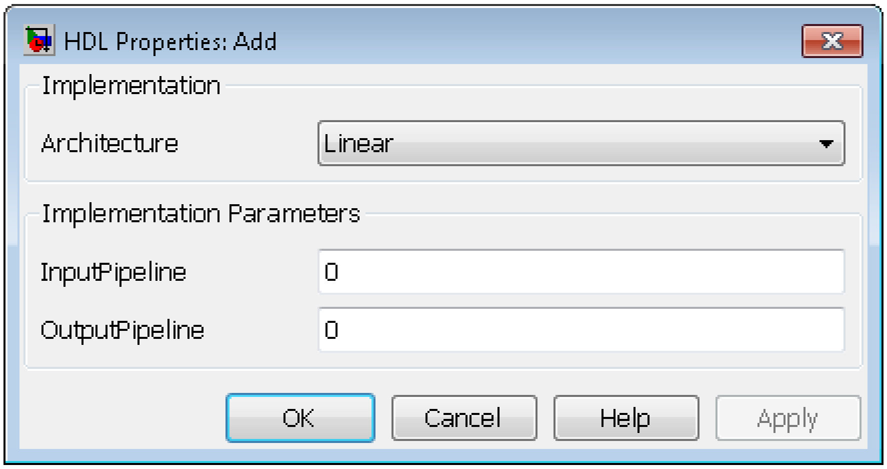
Option	Description	Availability
	Parameters" on page 12-55 for more information.	
HDL Workflow Advisor	Opens the HDL Workflow Advisor for the subsystem.	Available only for subsystems.
Navigate to Code	Activates the HTML code generation report window, displaying the beginning of the code generated for the selected block or subsystem. See "Tracing from Model to Code" on page 16-29 for more information.	Enabled when both code and a traceability report have been generated for the block or subsystem.

The HDL Block Properties Dialog Box

HDL Coder provides selectable alternate *block implementations* for many block types. Each implementation is optimized for different characteristics, such as speed or chip area. The HDL Properties dialog box lets you choose the implementation for a selected block.

Most block implementations support a number of *implementation parameters* that let you control further details of code generation for the block. The HDL Properties dialog box lets you set implementation parameters for a block.

The following figure shows the HDL Properties dialog box for a block.


HDL Properties: Add

Implementation

Architecture: Linear

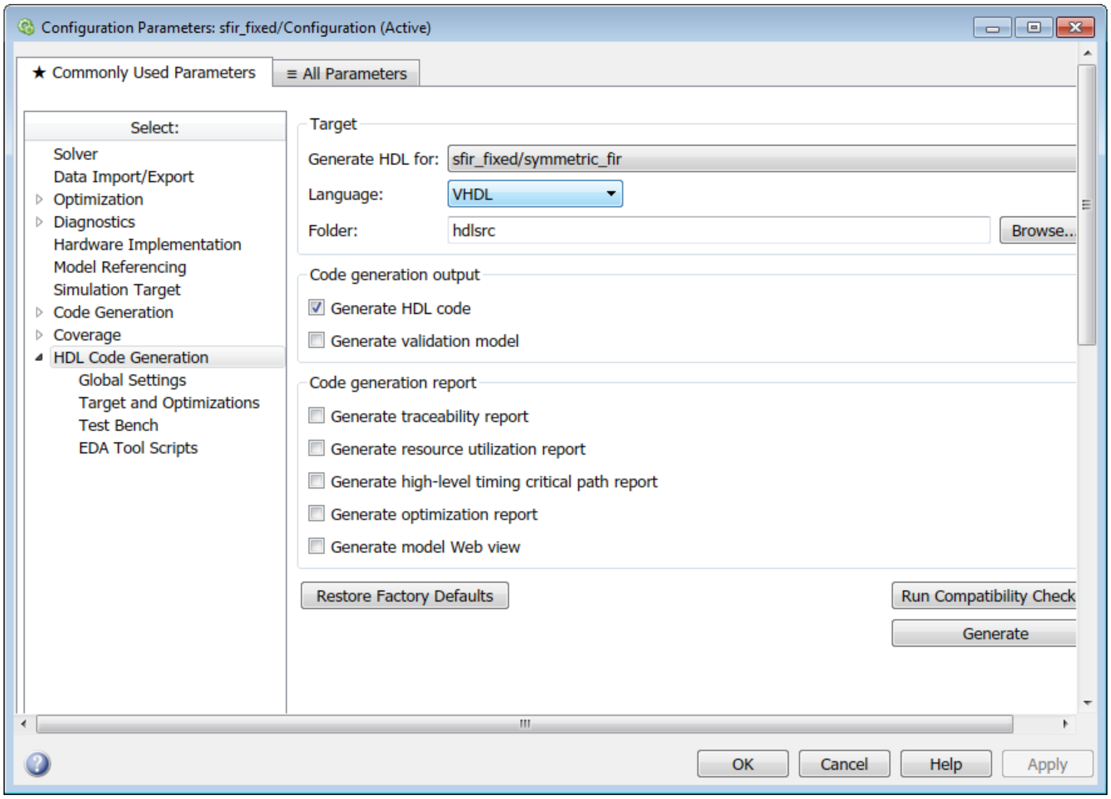
Implementation Parameters

InputPipeline: 0

OutputPipeline: 0

There are a number of ways to specify implementations and implementation parameters for individual blocks or groups of blocks. See “Set and View HDL Block Parameters” on page 12-55 for detailed information.

HDL Code Generation Pane: General


Configuration Parameters: sfir_fixed/Configuration (Active)

Target

Generate HDL for: sfir_fixed/symmetric_fir

Language: VHDL

Folder: hdlsrc

Code generation output

- [✓] Generate HDL code
- [ ] Generate validation model

Code generation report

- [ ] Generate traceability report
- [ ] Generate resource utilization report
- [ ] Generate high-level timing critical path report
- [ ] Generate optimization report
- [ ] Generate model Web view

In this section...

In this section...
“Generate high-level timing critical path report” on page 11-18
“Generate optimization report” on page 11-18
“Generate model Web view” on page 11-20

HDL Code Generation Top-Level Pane Overview

The top-level **HDL Code Generation** pane contains buttons that initiate code generation and compatibility checking, and sets code generation parameters.

Buttons in the HDL Code Generation Top-Level Pane

The buttons in the **HDL Code Generation** pane perform functions related to code generation. These buttons are:
Generate: Initiates code generation for the system selected in the **Generate HDL for** menu. See also makehdl.
Run Compatibility Checker: Invokes the compatibility checker to examine the system selected in the **Generate HDL for** menu for compatibility problems. See also checkhdl.
Browse: Lets you navigate to and select the target folder to which generated code and script files are written. The path to the target folder is entered into the **Folder** field.
Restore Factory Defaults: Sets model parameters to their default values.

Generate HDL for

Select the subsystem or model from which code is generated. The list includes the path to the root model and to subsystems in the model.

Settings

Default: The root model is selected.

Command-Line Information

Pass in the path to the model or subsystem for which code is to be generated as the first argument to `makehdl`.

See Also

makehdl

Language

Select the language (VHDL or Verilog) in which code is generated. The selected language is referred to as the target language.

Settings

Default: `VHDL`

`VHDL`

Generate VHDL code.

`Verilog`

Generate Verilog code.

Command-Line Information

Property: `TargetLanguage`
Type: character vector
Value: `'VHDL' | 'Verilog'`
Default: `'VHDL'`

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

See Also

- TargetLanguage
- makehdl

Folder

Enter a path to the folder into which code is generated. Alternatively, click **Browse** to navigate to and select a folder. The selected folder is referred to as the target folder.

Settings

Default: The default target folder is a subfolder of your working folder, named `hdlsrc`.

Command-Line Information

Property: `TargetDirectory`
Type: character vector
Value: A valid path to your target folder
Default: `'hdlsrc'`

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

See Also

- TargetDirectory
- makehdl

Generate HDL code

Enable or disable HDL code generation for the model.

Settings

Default: On

✓ On

Generate HDL code.

☐ Off

Do not generate HDL code.

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`.

See Also

`GenerateHDLCode`

Generate 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 both the original and the generated DUT models.

If you enable generation of a validation model, also enable delay balancing to keep 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

Generate the validation model.

☐ Off

Do not generate the validation 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`.

See Also

GenerateValidationModel, BalanceDelays

Generate traceability report

Enable or disable generation of an HTML code generation report with hyperlinks from code to model and model to code.

Settings

Default: Off

☑ On

Create and display an HTML code generation report. See “Create and Use Code Generation Reports” on page 16-2.

☐ Off

Do not create an HTML code generation report.

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`.

See Also

Traceability

Generate resource utilization report

Enable or disable generation of an HTML resource utilization report

Settings

Default: Off

☑ On

Create and display an HTML resource utilization report. The report contains information about the number of hardware resources (multipliers, adders, registers) used in the generated HDL code. The report includes hyperlinks to the referenced blocks in the model. See “Create and Use Code Generation Reports” on page 16-2.

☐ 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`.

See Also

ResourceReport

Generate high-level timing critical path report

Specify whether to generate a highlighting script that shows the estimated critical path.

Settings

Default: Off

☑ On

Generate a highlighting script that shows the estimated critical path.

The highlighting script is displayed as a link you can click to highlight the estimated critical path in the generated model. If your design contains blocks without timing information, another highlighting script is generated to highlight those blocks.

☐ 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`.

See Also

- `CriticalPathEstimation`
- `CriticalPathEstimationFile`
- `BlocksWithNoCharacterizationFile`
- "Find Estimated Critical Paths Without Synthesis Tools" on page 15-78

Generate optimization report

Enable or disable generation of an HTML optimization report

Settings

Default: Off

☑ On

Create and display an HTML optimization report. The report contains information about the results of streaming, sharing, and distributed pipelining optimizations that were implemented in the generated code. The report includes hyperlinks back to referenced blocks, subsystems, or validation models.See “Create and Use Code Generation Reports” on page 16-2.

☐ Off

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`.

See Also

OptimizationReport

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. 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.

☐ Off

Omit model Web view in the Code Generation report.

Dependencies

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

Command-Line Information

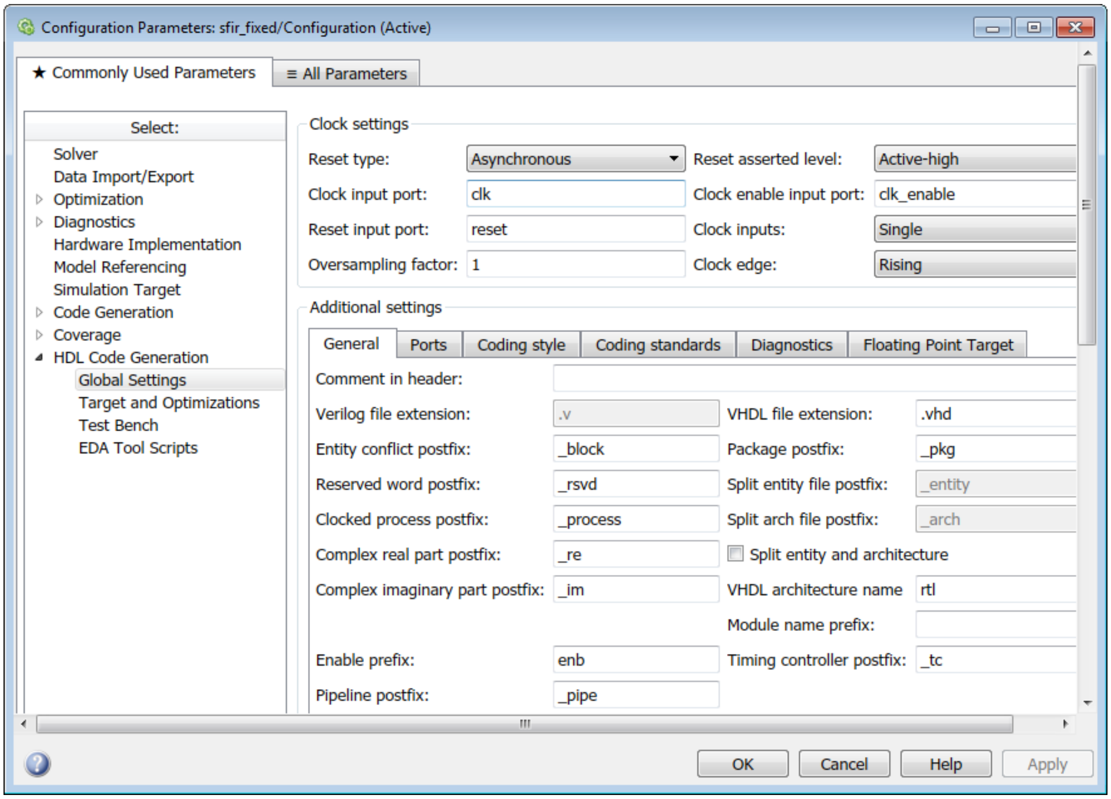
Parameter: `HDLGenerateWebview`
Type: character vector
Value: `'on' | 'off'`
Default: `'off'`

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

See Also

"Web View of Model in Code Generation Report" on page 16-35

HDL Code Generation Pane: Global Settings


Configuration Parameters: sfir_fixed/Configuration (Active)

Clock settings

Reset type: Asynchronous
Reset asserted level: Active-high
Clock input port: clk
Clock enable input port: clk_enable
Reset input port: reset
Clock inputs: Single
Oversampling factor: 1
Clock edge: Rising

Additional settings

Comment in header:
Verilog file extension: .v
VHDL file extension: .vhd
Entity conflict postfix: _block
Package postfix: _pkg
Reserved word postfix: _rsvd
Split entity file postfix: _entity
Clocked process postfix: _process
Split arch file postfix: _arch
Complex real part postfix: _re
Split entity and architecture
Complex imaginary part postfix: _im
VHDL architecture name rtl
Module name prefix:
Enable prefix: enb
Timing controller postfix: _tc
Pipeline postfix: _pipe

In this section...
"Global Settings Overview" on page 11-25
"Reset type" on page 11-26
"Reset asserted level" on page 11-27
"Clock input port" on page 11-28
"Clock enable input port" on page 11-29
"Reset input port" on page 11-30
"Clock inputs" on page 11-31
"Oversampling factor" on page 11-32

In this section...

In this section...
“Use Verilog `` `timescale `` directives” on page 11-63
“Inline VHDL configuration” on page 11-64
“Concatenate type safe zeros” on page 11-65
“Emit time/date stamp in header” on page 11-66
“Scalarize vector ports” on page 11-67
“Minimize intermediate signals” on page 11-68
“Include requirements in block comments” on page 11-69
“Inline MATLAB Function block code” on page 11-70
“Generate parameterized HDL code from masked subsystem” on page 11-71
“Initialize all RAM blocks” on page 11-72
“RAM Architecture” on page 11-72
“No-reset registers initialization” on page 11-73
“HDL coding standard” on page 11-74
“Do not show passing rules in coding standard report” on page 11-74
“Check for duplicate names” on page 11-75
“Check for HDL keywords in design names” on page 11-76
“Check for initial statements that set RAM initial values” on page 11-77
“Check module, instance, and entity name length” on page 11-78
“Check signal, port, and parameter name length” on page 11-79
“Minimize use of clock enable signals” on page 11-80
“Detect usage of reset signals” on page 11-81
“Detect usage of asynchronous reset signals” on page 11-82
“Check for conditional statements in processes” on page 11-83
“Minimize use of variables” on page 11-84
“Check if-else statement chain length” on page 11-85
“Check if-else statement nesting depth” on page 11-86
“Check multiplier width” on page 11-87
“Check for non-integer constants” on page 11-88
“Check line length” on page 11-89

In this section...

Global Settings Overview

The **Global Settings** pane enables you to specify detailed characteristics of the generated code, such as HDL element naming, and whether to map to a floating-point IP library.

Reset type

Specify whether to use asynchronous or synchronous reset logic when generating HDL code for registers.

Settings

Default: `Asynchronous`

`Asynchronous`

Use asynchronous reset logic.

`Synchronous`

Use synchronous reset logic.

Command-Line Information

Property: `ResetType`
Type: character vector
Value: `'async' | 'sync'`
Default: `'async'`

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

See Also

ResetType

Reset asserted level

Specify whether the asserted (active) level of reset input signal is active-high or active-low.

Settings

Default: `Active-high`

`Active-high`

Asserted (active) level of reset input signal is active-high (`1`).

`Active-low`

Asserted (active) level of reset input signal is active-low (`0`).

Command-Line Information

Property: `ResetAssertedLevel`
Type: character vector
Value: `'active-high'` | `'active-low'`
Default: `'active-high'`

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

See Also

ResetAssertedLevel

Clock input port

Specify the name for the clock input port in generated HDL code.

Settings

Default: `clk`

Enter the clock signal name in generated HDL code as a character vector. If you specify a VHDL or Verilog reserved word, the code generator appends a reserved word postfix string to form a valid VHDL or Verilog identifier. For example, if you specify the reserved word `signal`, the resulting name string would be `signal_rsvd`.

Command-Line Information

Property: `ClockInputPort`
Type: character vector
Value: A valid identifier in the target language
Default: `'clk'`

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

See Also

ClockInputPort

Clock enable input port

Specify the name for the clock enable input port in generated HDL code.

Settings

Default: `clk_enable`

Enter the clock enable input port name in generated HDL code as a character vector. If you specify a VHDL or Verilog reserved word, the code generator appends a reserved word postfix string to form a valid VHDL or Verilog identifier. For example, if you specify the reserved word `signal`, the resulting name string would be `signal_rsvd`.

Tip

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

Command-Line Information

Property: `ClockEnableInputPort`
Type: character vector
Value: A valid identifier in the target language
Default: `'clk_enable'`

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

See Also

ClockEnableInputPort

Reset input port

Enter the name for the reset input port in generated HDL code.

Settings

Default: `reset`

Enter a character vector for the reset input port name in generated HDL code. If you specify a VHDL or Verilog reserved word, the code generator appends a reserved word postfix string to form a valid VHDL or Verilog identifier. For example, if you specify the reserved word `signal`, the resulting name string would be `signal_rsvd`.

Tip

If the reset asserted level is set to active-high, the reset input signal is asserted active-high (1) and the input value must be high (1) for the entity's registers to be reset. If the reset asserted level is set to active-low, the reset input signal is asserted active-low (0) and the input value must be low (0) for the entity's registers to be reset.

Command-Line Information

Property: `ResetInputPort`
Type: character vector
Value: A valid identifier in the target language
Default: `'reset'`

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

See Also

ResetInputPort

Clock inputs

Specify generation of single or multiple clock inputs.

Settings

Default: `Single`

`Single`

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

`Multiple`

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

Command-Line Information

Property: `ClockInputs`
Type: character vector
Value: `'Single'` | `'Multiple'`
Default: `'Single'`

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

See Also

ClockInputs

Oversampling factor

Specify frequency of global oversampling clock as a multiple of the model's base rate.

Settings

Default: 1.

Oversampling factor specifies the *oversampling factor* of a global oversampling clock. The oversampling factor expresses the desired rate of the global oversampling clock as a multiple of your model's base rate. By default, HDL Coder does not generate a global oversampling clock.

If you want to generate a global oversampling clock:

- The **Oversampling factor** must be an integer greater than or equal to 1.
- In a multirate DUT, other rates in the DUT must divide evenly into the global oversampling rate.

Command-Line Information

Property: `Oversampling`
Type: int
Value: integer greater than or equal to 1
Default: 1

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

See Also

- "Generate a Global Oversampling Clock" on page 13-9
- `Oversampling`

Clock edge

Specify active clock edge.

Settings

Default: Rising.

Rising

The rising edge, or 0-to-1 transition, is the active clock edge.

Falling

The falling edge, or 1-to-0 transition, is the active clock edge.

Command-Line Information

Property: `ClockEdge`
Type: character vector
Value: `'Rising'` | `'Falling'`
Default: `'Rising'`

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

See Also

ClockEdge

Comment in header

Specify comment lines in header of generated HDL and test bench files.

Settings

Default: None

Text entered in this field generates a comment line in the header of generated model and test bench files. The code generator adds leading comment characters for the target language. When newlines or linefeeds are included, the code generator emits single-line comments for each newline.

Command-Line Information

Property: `UserComment`
Type: character vector

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

See Also

UserComment

Verilog file extension

Specify the file name extension for generated Verilog files.

Settings

Default: `.v`

This field specifies the file name extension for generated Verilog files.

Dependency

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

Command-Line Information

Property: `VerilogFileExtension`
Type: character vector
Default: `'.v'`

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

See Also

VerilogFileExtension

VHDL file extension

Specify the file name extension for generated VHDL files.

Settings

Default: `.vhd`

This field specifies the file name extension for generated VHDL files.

Dependencies

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

Command-Line Information

Property: `VHDLFileExtension`
Type: character vector
Default: `'.vhd'`

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

See Also

VHDLFileExtension

Entity conflict postfix

Specify the text as a character vector to resolve duplicate VHDL entity or Verilog module names in generated code.

Settings

Default: `_block`

The specified postfix resolves duplicate VHDL entity or Verilog module names. For example, in the default case, if HDL Coder detects two entities with the name `MyFilt`, the coder names the first entity `MyFilt` and the second instance `MyFilt_entity`.

Command-Line Information

Property: `EntityConflictPostfix`
Type: character vector
Value: A valid character vector in the target language
Default: `'_block'`

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

See Also

EntityConflictPostfix

Package postfix

Specify a text as a character vector to append to the model or subsystem name to form name of a package file.

Settings

Default: `_pkg`

HDL Coder applies this option only if a package file is required for the design.

Dependency

This option is enabled when:

The target language (specified by the **Language** option) is VHDL.

The target language (specified by the **Language** option) is Verilog, and the **Multi-file test bench** option is selected.

Command-Line Information

Property: `PackagePostfix`
Type: character vector
Value: A character vector that is legal in a VHDL package file name
Default: `'_pkg'`

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

See Also

PackagePostfix

Reserved word postfix

Specify a text as a character vector to append to value names, postfix values, or labels that are VHDL or Verilog reserved words.

Settings

Default: `_rsvd`

The reserved word postfix is applied to identifiers (for entities, signals, constants, or other model elements) that conflict with VHDL or Verilog reserved words. For example, if your generating model contains a signal named `mod`, HDL Coder adds the postfix `_rsvd` to form the name `mod_rsvd`.

Command-Line Information

Property: `ReservedWordPostfix`
Type: character vector
Default: `'_rsvd'`

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

See Also

ReservedWordPostfix

Split entity and architecture

Specify whether generated VHDL entity and architecture code is written to a single VHDL file or to separate files.

Settings

Default: Off

☑ On

VHDL entity and architecture definitions are written to separate files.

☐ Off

VHDL entity and architecture code is written to a single VHDL file.

Tips

The names of the entity and architecture files derive from the base file name (as specified by the generating model or subsystem name). By default, postfix strings identifying the file as an entity (`_entity`) or architecture (`_arch`) are appended to the base file name. You can override the default and specify your own postfix as a character vector.

For example, instead of all generated code residing in `MyFIR.vhd`, you can specify that the code reside in `MyFIR_entity.vhd` and `MyFIR_arch.vhd`.

Dependencies

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

Selecting this option enables the following parameters:

- **Split entity file postfix**
- **Split architecture file postfix**

Command-Line Information
Property: `SplitEntityArch`
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

SplitEntityArch

Clocked process postfix

Specify a character vector to append to HDL clock process names.

Settings

Default: `_process`

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

Command-Line Information

Property: `ClockProcessPostfix`
Type: character vector
Default: `'_process'`

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

See Also

ClockProcessPostfix

Complex real part postfix

Specify the character vector to append to real part of complex signal names.

Settings

Default: `'_re'`

Enter a text to be appended to the names generated for the real part of complex signals.

Command-Line Information

Property: `ComplexRealPostfix`
Type: character vector
Default: `'_re'`

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

See Also

ComplexRealPostfix

Complex imaginary part postfix

Specify character vector to append to imaginary part of complex signal names.

Settings

Default: `'_im'`

Enter a character vector to be appended to the names generated for the imaginary part of complex signals.

Command-Line Information

Property: `ComplexImagPostfix`
Type: character vector
Default: `'_im'`

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

See Also

ComplexImagPostfix

Split entity file postfix

Enter a character vector to be appended to the model name to form the name of a generated VHDL entity file.

Settings

Default: `_entity`

Dependencies

This parameter is enabled by **Split entity and architecture**.

Command-Line Information

Property: `SplitEntityFilePostfix`
Type: character vector
Default: `'_entity'`

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

See Also

SplitEntityFilePostfix

Split arch file postfix

Enter a character vector to be appended to the model name to form the name of a generated VHDL architecture file.

Settings

Default: `_arch`

Dependency

This parameter is enabled by **Split entity and architecture**.

Command-Line Information

Property: `SplitArchFilePostfix`
Type: character vector
Default: `'_arch'`

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

See Also

SplitArchFilePostfix

VHDL architecture name

Specify the architecture name for your DUT in the generated HDL code.

Settings

Default: `'rtl'`

Specify the VHDL architecture name for your DUT in the generated HDL code as a character vector.

Command-Line Information

Property: `VHDLArchitectureName`
Type: character vector
Default: `'rtl'`

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

See Also

`VHDLArchitectureName`

Module name prefix

Specify a prefix for every module or entity name in the generated HDL code.

Settings

Default: `''`

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

You can specify the module name prefix to avoid name collisions if you plan to instantiate the generated HDL code multiple times in a larger system.

Command-Line Information

Property: `ModulePrefix`
Type: character vector
Default: `''`

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

See Also

`ModulePrefix`

Enable prefix

Specify the base name as a character vector for internal clock enables and other flow control signals in generated code.

Settings

Default: `'enb'`

Where only a single clock enable is generated, **Enable prefix** specifies the signal name for the internal clock enable signal.

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

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

Command-Line Information

Property: `EnablePrefix`
Type: character vector
Default: `'enb'`

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

See Also

EnablePrefix

Pipeline postfix

Specify the postfix as a character vector to append to names of input or output pipeline registers generated for pipelined block implementations.

Settings

Default: `'_pipe'`

You can specify a generation of input and/or output pipeline registers for selected blocks. The **Pipeline postfix** option defines a character vector that HDL Coder appends to names of input or output pipeline registers.

Command-Line Information

Property: `PipelinePostfix`
Type: character vector
Default: `'_pipe'`

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

See Also

PipelinePostfix

Timing controller postfix

Specify the suffix appended to the DUT name to form the timing controller name.

Settings

Default: `'_tc'`

A timing controller file is generated if the design uses multiple rates, for example:

- When code is generated for a multirate model.
- When an area or speed optimization, or block architecture, introduces local multirate.

The timing controller name is based on the name of the DUT. For example, if the name of your DUT is `my_test`, by default, HDL Coder adds the postfix `_tc` to form the timing controller name, `my_test_tc`.

Command-Line Information

Property: `TimingControllerPostfix`
Type: character vector
Default: `'_tc'`

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

See Also

`TimingControllerPostfix`

Generate VHDL code for model references into a single library

Specify whether VHDL code generated for model references is in a single library, or in separate libraries.

Settings

Default: Off

☑ On

Generate VHDL code for model references into a single library.

☐ Off

For each model reference, generate a separate VHDL library.

Dependencies

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

Command-Line Information

Property: `UseSingleLibrary`
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

UseSingleLibrary

VHDL library name

Specify the target library name for the generated VHDL code.

Settings

Default: `'work'`

Target library name for generated VHDL code.

Command-Line Information
Property: `VHDLLibraryName`
Type: character vector
Default: `'work'`

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

See Also

- `VHDLLibraryName`
- `HDLCompileInit`
- `UseSingleLibrary`

Input data type

Specify the HDL data type for the model's input ports.

Settings

For VHDL, the options are:

Default: `std_logic_vector`

`std_logic_vector`
    Specifies VHDL type `STD_LOGIC_VECTOR`.

`signed/unsigned`

Specifies VHDL type `SIGNED` or `UNSIGNED`.

For Verilog, the options are:

Default: `wire`

In generated Verilog code, the data type for all ports is `'wire'`. Therefore, **Input data type** is disabled when the target language is Verilog.

Dependency

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

Command-Line Information

Property: `InputType`
Type: character vector
Value: (for VHDL)`'std_logic_vector'` | `'signed/unsigned'`
(for Verilog) `'wire'`
Default: (for VHDL) `'std_logic_vector'`
(for Verilog) `'wire'`

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

See Also

InputType

Output data type

Specify the HDL data type for the model's output ports.

Settings

For VHDL, the options are:

Default: `Same as input data type`

`Same as input data type`
: Specifies that output ports have the same type specified by **Input data type**.

`std_logic_vector`
: Specifies VHDL type `STD_LOGIC_VECTOR`.

`signed/unsigned`
: Specifies VHDL type `SIGNED` or `UNSIGNED`.

For Verilog, the options are:

Default: `wire`

In generated Verilog code, the data type for all ports is `'wire'`. Therefore, **Output data type** is disabled when the target language is Verilog.

Dependency

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

Command-Line Information
Property: `OutputType`
Type: character vector
Value: (for VHDL)`'std_logic_vector' | 'signed/unsigned'`
(for Verilog) `'wire'`
Default: If the property is left unspecified, output ports have the same type specified by `InputType`.

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

See Also

OutputType

Clock enable output port

Specify the name for the generated clock enable output.

Settings

Default: `ce_out`

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

Command-Line Information

Property: `ClockEnableOutputPort`
Type: character vector
Default: `'ce_out'`

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

See Also

ClockEnableOutputPort

Minimize clock enables

Omit generation of clock enable logic for single-rate designs.

Settings

Default: Off

☑ On

For single-rate models, omit generation of clock enable logic wherever possible. The following VHDL code example does not define or examine a clock enable signal. When the clock signal (`clk`) goes high, the current signal value is output.

```
Unit_Delay_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      Unit_Delay_out1 <= to_signed(0, 32);
    ELSIF clk'EVENT AND clk = '1' THEN
      Unit_Delay_out1 <= In1_signed;
    END IF;
  END PROCESS Unit_Delay_process;
```

☐ Off

Generate clock enable logic. The following VHDL code extract represents a register with a clock enable (`enb`)

```
Unit_Delay_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      Unit_Delay_out1 <= to_signed(0, 32);
    ELSIF clk'EVENT AND clk = '1' THEN
      IF enb = '1' THEN
        Unit_Delay_out1 <= In1_signed;
      END IF;
    END IF;
  END PROCESS Unit_Delay_process;
```

Exceptions

In some cases, HDL Coder emits clock enables even when **Minimize clock enables** is selected. These cases are:

- Registers inside Enabled, State-Enabled, and Triggered subsystems.

- Multirate models.
- The coder always emits clock enables for the following blocks:
  - commseqgen2/PN Sequence Generator
  - dspsigops/NCO

    Note: HDL support for the NCO block will be removed in a future release. Use the NCO HDL Optimized block instead.
  - dspsrcs4/Sine Wave
  - hdldemolib/HDL FFT
  - built-in/DiscreteFir
  - dspmlti4/CIC Decimation
  - dspmlti4/CIC Interpolation
  - dspmlti4/FIR Decimation
  - dspmlti4/FIR Interpolation
  - dspadpt3/LMS Filter
  - dsparch4/Biquad Filter

Command-Line Information

Property: `MinimizeClockEnables`
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

`MinimizeClockEnables`

Minimize global resets

Omit generation of reset logic in the HDL code.

Settings

Default: Off

☑ On

When you enable this setting, the code generator tries to minimize or remove the global reset logic from the HDL code. This code snippet corresponds to the Verilog code generated for a Delay block in the Simulink model. The code snippet shows that HDL Coder removed the reset logic.

```
always @(posedge clk)
   begin : Delay_Synchronous_process
     if (enb) begin
       Delay_Synchronous_out1 <= DataIn;
     end
   end
```

☐ Off

When you disable this parameter, HDL Coder generates the global reset logic in the HDL code. This Verilog code snippet shows the reset logic generated for the Delay block.

```
always @(posedge clk or posedge reset)
  begin : Delay_Synchronous_process
    if (reset == 1'b1) begin
      Delay_Synchronous_out1 <= 1'b0;
    end
    else begin
      if (enb) begin
        Delay_Synchronous_out1 <= DataIn;
      end
    end
  end
```

Dependencies

If you enable `MinimizeGlobalResets`, the generated HDL code contains registers that do not have a reset port. If you do not initialize these registers, there can be potential numerical mismatches in the HDL simulation results. To avoid simulation mismatches, you can initialize the registers by using the “No-reset registers initialization” on page 11-73 setting.

By default, the **No-reset registers initialization** setting generates an external script to initialize the registers. To initialize registers with the script, use a zero initial value for the blocks in your Simulink model. If these blocks have a non-zero initial value, set **No-reset registers initialization** to `Generate initialization inside module`.

Exceptions

Sometimes, when you set `MinimizeGlobalResets` to `'on'`, HDL Coder generates the reset logic, if you have:

- Blocks with state that have a non-zero initial value, such as a Delay block with non-zero **Initial Condition**.
- Enumerated data types for blocks with state.
- Subsystem blocks with `BlackBox` HDL architecture where you request a reset signal.
- Multirate models with `TimingControllerArch` set to `default`.

  If you set `TimingControllerArch` to `resettable`, HDL Coder generates a reset port for the timing controller. If you set `MinimizeGlobalResets` to `'on'`, the code generator removes this reset port.
- Truth Table
- Chart
- MATLAB Function block

Command-Line Information

Property: `MinimizeGlobalResets`
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

`MinimizeGlobalResets`

Use trigger signal as clock

Enable to use trigger input signal as clock in generated HDL code.

Settings

Default: Off

☑ On

For triggered subsystems, use the trigger input signal as a clock in the generated HDL code.

☐ Off

For triggered subsystems, do not use the trigger input signal as a clock in the generated HDL code.

Command-Line Information

Property: `TriggerAsClock`
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

`TriggerAsClock`

Represent constant values by aggregates

Specify whether constants in VHDL code are represented by aggregates, including constants that are less than 32 bits.

Settings

Default: Off

☑ On

HDL Coder represents constants as aggregates. The following VHDL constant declarations show a scalar less than 32 bits represented as an aggregate:

```
GainFactor_gainparam <= (14 => '1',  OTHERS => '0');
```

☐ Off

The coder represents constants less than 32 bits as scalars and constants greater than or equal to 32 bits as aggregates. The following VHDL code was generated by default for a value less than 32 bits:

```
GainFactor_gainparam <= to_signed(16384, 16);
```

Dependencies

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

Command-Line Information

Property: `UseAggregatesForConst`
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

`UseAggregatesForConst`

Use “rising_edge/falling_edge” style for registers

Specify whether generated code uses the VHDL `rising_edge` or `falling_edge` function to detect clock transitions.

Settings

Default: Off

☑ On

Generated code uses the VHDL `rising_edge` or `falling_edge` function.

☐ Off

Generated code uses the `'event` syntax.

Dependencies

This option is enabled when the target language is VHDL.

Command-Line Information

Property: `UseRisingEdge`
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

`UseRisingEdge`

Loop unrolling

Specify whether VHDL `FOR` and `GENERATE` loops are unrolled and omitted from generated VHDL code.

Settings

Default: Off

✓ On

Unroll and omit `FOR` and `GENERATE` loops from the generated VHDL code. (In Verilog code, loops are always unrolled.)

☐ Off

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.

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`.

See Also

`LoopUnrolling`

Use Verilog `` `timescale `` directives

Specify use of compiler `` `timescale `` directives in generated Verilog code.

Settings

Default: On

☑ On

Use compiler `` `timescale `` directives in generated Verilog code.

☐ Off

Suppress the use of compiler `` `timescale `` directives in generated Verilog code.

Tip

The `` `timescale `` directive provides a way of specifying different delay values for multiple modules in a Verilog file. This setting does not affect the generated test bench.

Dependency

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

Command-Line Information

Property: `UseVerilogTimescale`
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

`UseVerilogTimescale`

Inline VHDL configuration

Specify whether generated VHDL code includes inline configurations.

Settings

Default: On

☑ On

Include VHDL configurations in files that instantiate a component.

☐ Off

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

Tip

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

Dependencies

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

Command-Line Information

Property: `InlineConfigurations`
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

`InlineConfigurations`

Concatenate type safe zeros

Specify use of syntax for concatenated zeros in generated VHDL code.

Settings

Default: On

☑ On

Use the type-safe syntax, `'0' & '0'`, for concatenated zeros. Typically, this syntax is preferred.

☐ Off

Use the syntax "`000000...`" for concatenated zeros. This syntax can be easier to read and more compact, but it can lead to ambiguous types.

Dependencies

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

Command-Line Information

Property: `SafeZeroConcat`
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

`SafeZeroConcat`

Emit time/date stamp in header

Specify whether or not to include time and date information in the generated HDL file header.

Settings

Default: On

☑ On

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

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

☐ Off

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

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

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

Command-Line Information

Property: `DateComment`
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

`DateComment`

Scalarize vector ports

Flatten vector ports into a structure of scalar ports in VHDL code

Settings

Default: Off

☑ On

When generating code for a vector port, generate a structure of scalar ports.

☐ Off

When generating code for a vector port, generate a type definition and port declaration for the vector port.

Dependencies

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

Command-Line Information

Property: `ScalarizePorts`
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

`ScalarizePorts`

Minimize intermediate signals

Specify whether to optimize HDL code for debuggability or code coverage.

Settings

Default: Off

☑ On

Optimize for code coverage by minimizing intermediate signals. For example, suppose that the generated code with this setting *off* is:

```
const3 <= to_signed(24, 7);
subtractor_sub_cast <= resize(const3, 8);
subtractor_sub_cast_1 <= resize(delayout, 8);
subtractor_sub_temp <= subtractor_sub_cast - subtractor_sub_cast_1;
```

With this setting *on*, HDL Coder optimizes the output to:

```
subtractor_sub_temp <= 24 - (resize(delayout, 8));
```

The coder removes the intermediate signals `const3`, `subtractor_sub_cast`, and `subtractor_sub_cast_1`.

☐ Off

Optimize for debuggability by preserving intermediate signals.

Command-Line Information

Property: `MinimizeIntermediateSignals`
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

`MinimizeIntermediateSignals`

Include requirements in block comments

Enable or disable generation of requirements comments as comments in code or code generation reports

Settings

Default: On

☑ On

If the model contains requirements comments, include them as comments in code or code generation reports. See “Requirements Comments and Hyperlinks” on page 16-38.

☐ Off

Do not include requirements as comments in code or code generation reports.

Command-Line Information

Property: `RequirementComments`
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

`RequirementComments`

Inline MATLAB Function block code

Inline HDL code for MATLAB Function blocks.

Settings

Default: Off

☑ On

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

☐ Off

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

Command-Line Information

Property: `InlineMATLABBlockCode`
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

`InlineMATLABBlockCode`

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

☑ On

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

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

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

☐ Off

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 18-20

- `MaskParameterAsGeneric`

Initialize all RAM blocks

Enable or suppress generation of initial signal value for RAM blocks.

Settings

Default: On

☑ On

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

☐ Off

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

Command-Line Information

Property: `InitializeBlockRAM`
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

`InitializeBlockRAM`

RAM Architecture

Select RAM architecture with clock enable, or without clock enable, for all RAMs in DUT subsystem.

Settings

Default: `RAM with clock enable`

Select one of the following options from the menu:

- `RAM with clock enable`: Generate RAMs with clock enable.
- `Generic RAM without clock enable`: Generate RAMs without clock enable.

Command-Line Information

Property: `RAMArchitecture`
Type: character vector
Value: `'WithClockEnable'` | `'WithoutClockEnable'`
Default: `'WithClockEnable'`

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

See Also

`RAMArchitecture`

No-reset registers initialization

Specify whether you want to initialize registers without reset and the mode of initialization.

Settings

Default: `Generate an external script`

The options are:

`Do not initialize`

Select this option if you do not want HDL Coder to initialize the registers that do not have an external reset port in the generated code.

`Generate an external script`

Select this option to generate an external script to initialize the No-reset registers for ModelSim® simulation. The initialization script is compliant with ModelSim 10.2c or later. This initialization mode does not support enumeration types.

`Generate initialization inside module`

Select this option to generate the code for initializing the registers as part of the HDL code generated for the DUT. In Verilog, an `initial` construct in the corresponding module definition initializes the No-reset registers. In VHDL, the initialization code is part of the signal declaration statements.

See Also

`NoResetInitializationMode`

HDL coding standard

Specify an HDL coding standard.

Settings

Default: `None`

`None`

Generate generic synthesizable HDL code.

`Industry`

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

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`.

See Also

`HDLCodingStandard`

Do not show passing rules in coding standard report

Specify whether to show rules without errors or warnings in the coding standard report.

Settings

Default: Off

☑ On

Show only rules with errors or warnings.

☐ Off

Show all rules.

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

`HDL Coding Standard Customization`

Check for duplicate names

Specify whether to check for duplicate names in the design (CGSL-1.A.A.5).

Settings

Default: On

☑ On

Check for duplicate names.

☐ Off

Do not check for duplicate names.

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

`HDL Coding Standard Customization`

Check for HDL keywords in design names

Specify whether to check for HDL keywords in design names (CGSL-1.A.A.3).

Settings

Default: On

☑ On

Check for HDL keywords in design names.

☐ Off

Do not check for HDL keywords in design names.

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

`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 (CGSL-2.C.D.1).

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.

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

`HDL Coding Standard Customization`

Check module, instance, and entity name length

Specify whether to check module, instance, and entity name length (CGSL-1.A.C.3).

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.

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

`HDL Coding Standard Customization`

Check signal, port, and parameter name length

Specify whether to check signal, port, and parameter name length (CGSL-1.A.B.1).

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.

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

`HDL Coding Standard Customization`

Minimize use of clock enable signals

Specify whether to check for clock enable signals in the generated code (CGSL-2.C.C.4).

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.

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

- `HDL Coding Standard Customization`
- "Minimize Clock Enables" on page 5-53

Detect usage of reset signals

Specify whether to check for reset signals in the generated code (CGSL-2.C.C.5).

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.

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

`HDL Coding Standard Customization`

Detect usage of asynchronous reset signals

Specify whether to check for asynchronous reset signals in the generated code (CGSL-2.C.C.6).

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.

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

`HDL Coding Standard Customization`

Check for conditional statements in processes

Specify whether to check for length of conditional statements that are described separately within a process (CGSL-2.F.B.1).

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.

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

`HDL Coding Standard Customization`

Minimize use of variables

Specify whether to minimize use of variables (CGSL-2.G).

Settings

Default: Off

☑ On

Minimize use of variables.

☐ Off

Do not minimize use of variables.

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

`HDL Coding Standard Customization`

Check if-else statement chain length

Specify whether to check if-else statement chain length (CGSL-2.G.C.1c).

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.

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

`HDL Coding Standard Customization`

Check if-else statement nesting depth

Specify whether to check if-else statement nesting depth (CGSL-2.G.C.1a).

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.

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

`HDL Coding Standard Customization`

Check multiplier width

Specify whether to check multiplier bit width (CGSL-2.J.F.5).

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.

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

`HDL Coding Standard Customization`

Check for non-integer constants

Specify whether to check for non-integer constants (CGSL-3.B.D.1).

Settings

Default: On

☑ On

Check for non-integer constants.

☐ Off

Do not check for non-integer constants.

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

`HDL Coding Standard Customization`

Check line length

Specify whether to check line lengths in the generated HDL code (CGSL-3.A.D.5).

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.

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

`HDL Coding Standard Customization`

Highlight feedback loops inhibiting delay balancing and optimizations

Specify whether to generate a script to highlight feedback loops that are inhibiting delay balancing and optimizations.

Settings

Default: Off

☑ On

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

☐ Off

Do not generate a script to highlight feedback loops.

Command-Line Information

Property: `HighlightFeedbackLoops`
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

- `HighlightFeedbackLoops`
- `HighlightFeedbackLoopsFile`
- "Find Feedback Loops" on page 15-33

Highlight blocks inhibiting clock-rate pipelining

Specify whether to generate a script to highlight blocks that are inhibiting clock-rate pipelining.

Settings

Default: Off

☑ On

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

☐ 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: `'off'`

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

See Also

- `HighlightClockRatePipeliningDiagnostic`
- `HighlightClockRatePipeliningFile`

Highlight blocks inhibiting distributed pipelining

Specify whether to generate a script to highlight blocks that are inhibiting distributed pipelining.

Settings

Default: Off

☑ On

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

☐ Off

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

Command-Line Information

Property: `DistributedPipeliningBarriers`
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

- `DistributedPipeliningBarriers`
- `DistributedPipeliningBarriersFile`

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`.

See Also

- `DetectBlackBoxNameCollision`

Library

Specify the floating-point target library

Settings

Default: `NONE`

The options are:

`None`

Select this option if you do not want the design to map to floating-point target libraries.

`Native Floating Point`

Specify native floating-point as the library. You can specify the latency strategy and whether to handle denormal numbers in your design.

`Altera Megafunctions (ALTERA FP FUNCTIONS)`

Specify Altera Megafunctions (ALTERA FP FUNCTIONS) as the floating-point target library. You can provide the IP Target frequency.

`Altera Megafunctions (ALTFP)`

Specify Altera Megafunctions (ALTFP) as the floating-point target library. You can provide the objective and latency strategy for the IP.

`Xilinx LogiCORE`

Specify Xilinx LogiCORE® as the floating-point target library. You can provide the objective and latency strategy for the IP.

See Also

- `hdlcoder.createFloatingPointTargetConfig`
- "Generate Target-Independent HDL Code with Native Floating-Point" on page 10-52
- "Generate HDL Code for FPGA Floating-Point Target Libraries" on page 22-26

Initialize IP Pipelines To Zero

Inserts additional logic during HDL code generation to initialize the values of pipeline registers in the Altera floating-point target IP to zero. If you do not select the check box, HDL Coder reports a warning during HDL code generation.

Settings

Default: On

☑ On

Inserts additional logic to initialize pipeline registers in the floating-point target IP to zero.

☐ Off

Does not add additional logic to initialize pipeline registers in the floating-point target IP to zero.

See Also

- `hdlcoder.createFloatingPointTargetConfig`
- "Generate HDL Code for FPGA Floating-Point Target Libraries" on page 22-26

Latency Strategy

Specify whether you want the design to map to minimum or maximum latency with Xilinx LogiCORE or ALTFP Altera megafunction IPs.

Settings

Default: `MIN`

The options are:

`MIN`

Maps to minimum latency for the specified floating-point target IP.

`MAX`

Maps to maximum latency for the specified floating-point target IP.

See also

- `hdlcoder.createFloatingPointTargetConfig`
- "Generate HDL Code for FPGA Floating-Point Target Libraries" on page 22-26
- "Customize Floating-Point IP Configuration" on page 22-34

Objective

Specify whether you want to optimize the design for speed or area when mapping to floating-point target libraries.

Settings

Default: `SPEED`

The options are:

`NONE`
: Select this option if you do not want to optimize the design for speed or area.

`SPEED`
: Select this option to optimize the design for speed.

`AREA`
: Select this option to optimize the design for area.

See also

- `hdlcoder.createFloatingPointTargetConfig`
- “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 22-26
- “Customize Floating-Point IP Configuration” on page 22-34

IP Settings

The **IP Settings** section has an IP configuration table with the IP names and data types and additional options to specify a custom latency and any extra arguments.

The options in the IP configuration table depend on the library that you specify.

- If you specify the `ALTERA MEGAFUNCTION (ALTERA FP FUNCTIONS)` library, HDL Coder infers the latency value from the **Target Frequency (MHz)** value.
- If you specify the `ALTERA MEGAFUNCTION (ALTFP)` or `XILINX LOGICORE` libraries, HDL Coder infers the IP latency from the **Latency Strategy** setting. The IP configuration table has two additional columns, **MinLatency** and **MaxLatency**, that contain the minimum and maximum latency values for each IP in the table.

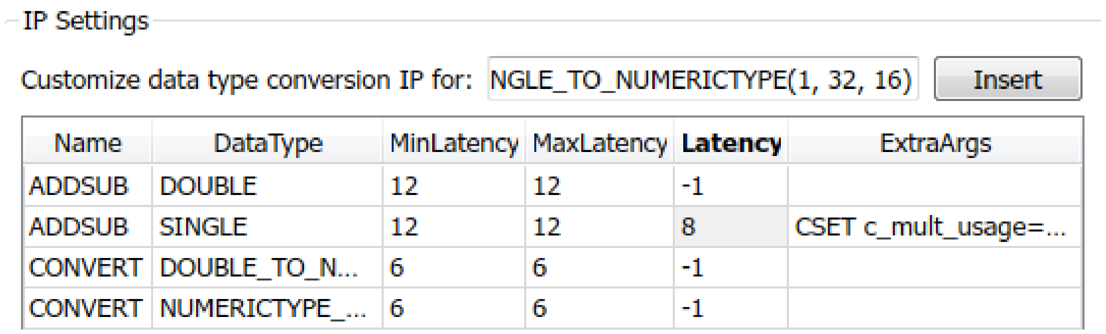
The IP configuration table has these sections:

- **Name**: Contains a list of IP names that HDL Coder map the Simulink blocks to, such as `ABS`, `ADDSUB`, and `CONVERT`.
- **DataType**: Contains a list of IP data types for each IP in the table. These are mostly `SINGLE` and `DOUBLE` data types. The `CONVERT` IP blocks can have `DOUBLE_TO_NUMERICTYPE`, `NUMERICTYPE_TO_DOUBLE` data types, and so on.
- **Latency**: The default latency value of `–1` means that the IP inherits the latency value from the target frequency or the latency strategy setting depending on the library

that you choose. To customize the latency of the IP that your Simulink blocks map to, enter your own custom value for the latency.

- **ExtraArgs**: Specify any additional settings that is specific to the IP.

For example, if you have an Add block with `Single` data types in your Simulink model, HDL Coder maps the block to the **ADDSUB** IP. If you want to specify a custom latency value, say 8, for the IP, enter the value in the **Latency** column for the IP.



Customize data type conversion IP for: NGLE_TO_NUMERICTYPE(1, 32, 16)

Name	DataType	MinLatency	MaxLatency	**Latency**	ExtraArgs
ADDSUB	DOUBLE	12	12	-1	
ADDSUB	SINGLE	12	12	8	CSET c_mult_usage=...
CONVERT	DOUBLE_TO_N...	6	6	-1	
CONVERT	NUMERICTYPE_...	6	6	-1	

`cmultusage` is a parameter that you can specify with the Xilinx LogiCORE libraries.

See also

- `hdlcoder.createFloatingPointTargetConfig`
- "Generate HDL Code for FPGA Floating-Point Target Libraries" on page 22-26
- "Customize Floating-Point IP Configuration" on page 22-34

Latency Strategy

Specify whether you want the design to map to minimum or maximum latency with native floating-point libraries.

Settings

Default: `MAX`

The options are:

`MIN`

Maps to minimum latency for the native floating-point libraries.

`MAX`

Maps to maximum latency for the native floating-point libraries.

`ZERO`

Does not use any latency for the native floating-point libraries.

See also

- "LatencyStrategy" on page 12-16
- `hdlcoder.createFloatingPointTargetConfig`
- "Generate Target-Independent HDL Code with Native Floating-Point" on page 10-52
- "Latency Customization with Native Floating-Point" on page 10-47

Handle Denormals

Specify whether you want to handle denormal numbers in your design. Denormal numbers are nonzero numbers that are smaller than the smallest normal number.

Settings

Default: Off

☑ On

Inserts additional logic to handle the denormal numbers in your design.

☐ Off

Does not add additional logic to handle the denormal numbers in your design. If the input is a denormal value, HDL Coder treats the value as zero before performing any computations.

See also

- "HandleDenormals" on page 12-15
- `hdlcoder.createFloatingPointTargetConfig`
- "Generate Target-Independent HDL Code with Native Floating-Point" on page 10-52
- "Numerical Considerations with Native Floating-Point" on page 10-42

Mantissa Multiplier Strategy

Specify how you want HDL Coder to implement the mantissa multiplication operation when you have Product blocks in your design.

Settings

Default: Full Multiplier

The options are:

`Full Multiplier`

Specify this option to use only multipliers for implementing the mantissa multiplication. The multipliers can utilize DSP units on the target device.

`Part Multiplier Part AddShift`

Specify this option to split 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.

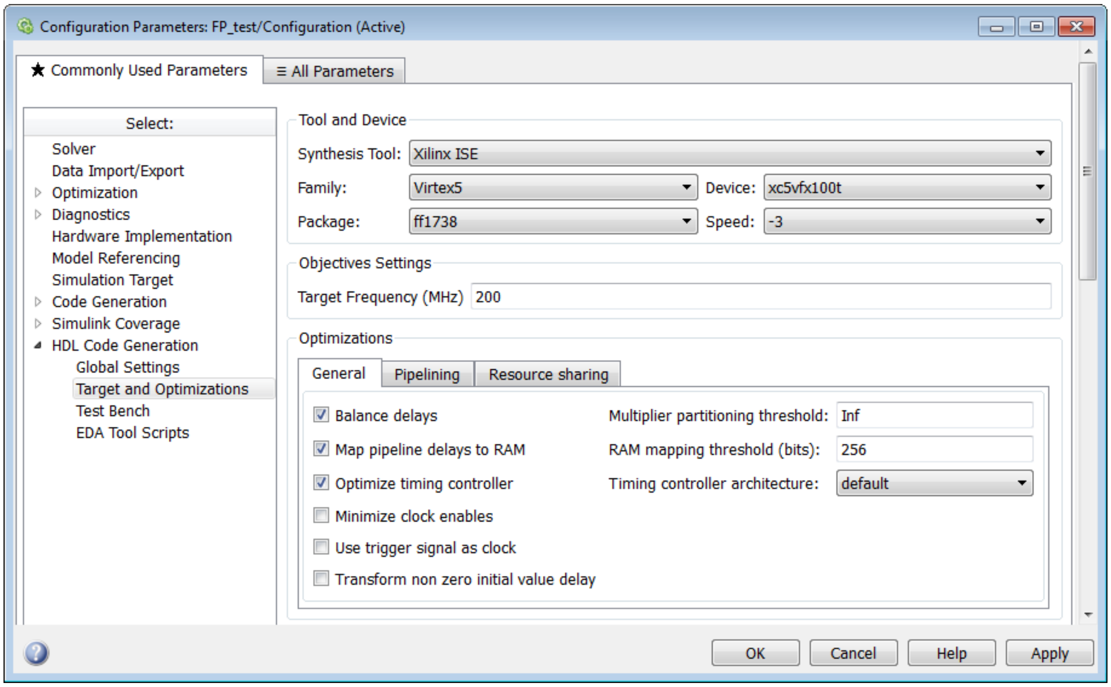
`No Multiplier Full AddShift`

Select this option to use a combination of adders and multipliers 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.

See also

- "MantissaMultiplyStrategy" on page 12-17
- `hdlcoder.createFloatingPointTargetConfig`
- "Generate Target-Independent HDL Code with Native Floating-Point" on page 10-52

HDL Code Generation Pane: Target and Optimizations



Tool and Device

Synthesis Tool: Xilinx ISE

Family: Virtex5 Device: xc5vfx100t

Package: ff1738 Speed: -3

Objectives Settings

Target Frequency (MHz) 200

Optimizations

[✓] Balance delays

[✓] Map pipeline delays to RAM

[✓] Optimize timing controller

[ ] Minimize clock enables

[ ] Use trigger signal as clock

[ ] Transform non zero initial value delay

Multiplier partitioning threshold: Inf

RAM mapping threshold (bits): 256

Timing controller architecture: default

In this section...

In this section...

Target and Optimizations Overview

The **Target and Optimizations** pane enables you to specify the target hardware and apply certain optimizations to the generated HDL code.

Tool and Device

Select a synthesis tool, then select the **Family**, **Device**, **Package**, and **Speed** for your synthesis target.

Settings

Default: `No synthesis tool specified`

The options are:

`No synthesis tool specified`
: Select this option if you do not want to perform logic synthesis. You can generate HDL code from your design.

`Xilinx Vivado`
: Specify Xilinx Vivado as the synthesis tool.

`Xilinx ISE`
: Specify Xilinx ISE as the synthesis tool.

`Altera Quartus II`
: Specify Altera Quartus II as the synthesis tool.

If your synthesis tool is not one of the **Synthesis tool** options, see “Synthesis Tool Path Setup”.

Target Frequency

Specify the target frequency in MHz for multiple features and workflows.

Settings

Default: `0`

This setting is the target frequency in MHz for multiple features and workflows that HDL Coder supports. The supported features are:

- FPGA floating-point target library mapping: Specify the target frequency that you want the IP to achieve when you use `ALTERA MEGAFUNCTION (ALTERA FP FUNCTIONS)`. If you do not specify the target frequency, HDL Coder sets the target frequency to a default value of `200 MHz`.
- Adaptive pipelining: If your design uses multipliers, specify the synthesis tool and the target frequency. Based on these settings, HDL Coder estimates the number of pipelines that can be inserted to improve area and timing on the target platform. If you do not specify the target frequency, HDL Coder uses a target frequency of `0 MHz` and cannot insert pipelines.

You can also set the target frequency by using the **Target Frequency (MHz)** setting in the **Set Target Frequency** task in the HDL Workflow Advisor.

Specify the target frequency for these workflows-

- `Generic ASIC/FPGA`: To specify the target frequency that you want your design to achieve. HDL Coder generates a timing constraint file for that clock frequency. It adds the constraint to the FPGA synthesis tool project that you create in the **Create Project** task. If the target frequency is not achievable, the synthesis tool generates an error.
- `IP Core Generation`: To specify the target frequency for HDL Coder to modify the clock module setting in the reference design to produce the clock signal with that frequency. Enter a target frequency value that is within the **Frequency Range (MHz)**. If you do not specify the target frequency, HDL Coder uses the **Default (MHz)** target frequency.
- `Simulink Real-Time FPGA I/O`: For Speedgoat boards that are supported with `Xilinx ISE`, specify the target frequency to generate the clock module to produce the clock signal with that frequency.

  The Speedgoat boards that are supported with `Xilinx Vivado` use the `IP Core Generation` workflow infrastructure. Specify the target frequency for HDL Coder to modify the clock module setting in the reference design to produce the clock signal with that frequency. Enter a target frequency value that is within the **Frequency Range (MHz)**. If you do not specify the target frequency, HDL Coder uses the **Default (MHz)** target frequency.
- `FPGA Turnkey`: To generate the clock module to produce the clock signal with that frequency automatically.

Command-Line Information

Property: `TargetFrequency`

Type: integer
Value: integer greater than or equal to 0
Default: 0

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

See Also

- `TargetFrequency`
- "Set Target Frequency" on page 25-9
- "Customize Floating-Point IP Configuration" on page 22-34

Balance delays

Enable delay balancing.

Settings

Default: On

☑ On

If HDL Coder detects introduction of new delays along one path, matching delays are inserted on the other paths. When delay balancing is enabled, the generated model is functionally equivalent to the original model.

☐ Off

The latency along signal paths might not be balanced, and the generated model might not be functionally equivalent to the original model.

Command-Line Information

Property: `BalanceDelays`
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

"Delay Balancing" on page 15-26

Map pipeline delays to RAM

Map pipeline registers in the generated HDL code to RAM.

Settings

Default: Off

☑ On

Map pipeline registers in the generated HDL code to RAM.

☐ Off

Do not map pipeline registers in the generated HDL code to RAM.

Command-Line Information

Property: `MapPipelineDelaysToRAM`
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

`MapPipelineDelaysToRAM`

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`.

See Also

`OptimizeTimingController`

Transform non zero initial value delay

Enable this option to optimize Delay blocks with non zero initial condition.

Settings

Default: On

☑ On

Transform Delay blocks with nonzero **Initial condition** in your Simulink model to Delay blocks with zero **Initial condition** and some additional logic in the generated HDL code.

By using this transformation, HDL Coder can perform optimizations such as sharing, distributed pipelining, and clock-rate pipelining more effectively, and prevent an assertion from being triggered in the validation model.

☐ Off

Do not transform Delay blocks with nonzero **Initial condition** in your Simulink model.

Command-Line Information

Property: `TransformNonZeroInitValDelay`
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

`TransformNonZeroInitDelay`

Multiplier partitioning threshold

Specify the maximum input bit width for multipliers in your design.

Settings

Default: `Inf`

N, where N is an integer greater than or equal to 2

Partition multipliers so that N is the maximum multiplier input bit width.

To improve hardware mapping results, set the multiplier partitioning threshold to the input bit width of the DSP or multiplier hardware on your target device.

Inf

Do not partition multipliers.

Command-Line Information

Property: `MultiplierPartitioningThreshold`
Type: integer
Value: integer greater than or equal to 0
Default: `Inf`

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

See Also

`MultiplierPartitioningThreshold`

RAM mapping threshold (bits)

Specify the minimum RAM size for mapping to block RAMs instead of to registers.

Settings

Default: 256

The RAM mapping threshold must be an integer greater than or equal to zero. HDL Coder uses the threshold to determine whether or not to map the following elements to block RAMs instead of to registers:

- Delay blocks
- Persistent arrays in MATLAB Function blocks

Command-Line Information

Property: `RAMMappingThreshold`
Type: integer
Value: integer greater than or equal to 0
Default: 256

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

See Also

- `RAMMappingThreshold`
- "UseRAM" on page 12-30
- "MapPersistentVarsToRAM" on page 12-22

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`.

See Also

`TimingControllerArch`

Hierarchical distributed pipelining

Specify that retiming be applied across a subsystem hierarchy.

Settings

Default: Off

☑ On

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

☐ Off

Distribute pipelining only within a subsystem.

Command-Line Information
Property: `HierarchicalDistPipelining`
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

- `HierarchicalDistPipelining`
- "DistributedPipelining" on page 12-10

Clock-rate pipelining

Insert pipeline registers at the clock rate instead of the data rate for multi-cycle paths in your design.

Settings

Default: On

☑ On

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

☐ Off

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

Command-Line Information
Property: `ClockRatePipelining`

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

- `ClockRatePipelining`
- "Clock-Rate Pipelining" on page 15-66

Allow clock-rate pipelining of DUT output ports

For DUT output ports, insert pipeline registers at the clock rate instead of the data rate.

Settings

Default: Off

☑ On

At DUT output ports, insert pipeline registers at clock rate.

☐ Off

At DUT output ports, insert pipeline registers at data rate.

Command-Line Information

Property: `ClockRatePipelineOutputPorts`
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

- `ClockRatePipelineOutputPorts`
- `ClockRatePipelining`

- “Clock-Rate Pipelining” on page 15-66

Preserve design delays

Enable to prevent distributed pipelining from moving design delays.

Settings

Default: Off

☑ On

Prevent distributed pipelining from moving design delays.

☐ Off

Do not prevent distributed pipelining from moving design delays.

Command-Line Information

Property: `PreserveDesignDelays`
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

- `PreserveDesignDelays`
- “DistributedPipelining” on page 12-10

Adaptive pipelining

Insert adaptive pipeline registers in your design.

Settings

Default: On

☑ On

Insert adaptive pipeline registers in your design. For HDL Coder to insert adaptive pipelines, you must specify the synthesis tool. If your design has multipliers, specify the synthesis tool and the target frequency for adaptive pipeline insertion.

☐ Off

Do not insert adaptive pipeline registers.

Command-Line Information

Property: `AdaptivePipelining`
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

- "Adaptive pipelining" on page 11-112
- `AdaptivePipelining`

Distributed pipelining priority

Specify priority for distributed pipelining algorithm.

Settings

Default: `Numerical Integrity`

`Numerical Integrity`

Prioritize numerical integrity when distributing pipeline registers.

This option uses a conservative retiming algorithm that does not move registers across a component if the functional equivalence to the original design is unknown.

`Performance`

Prioritize performance over numerical integrity.

Use this option if your design requires a higher clock frequency and the Simulink behavior does not need to strictly match the generated code behavior. This option uses a more aggressive retiming algorithm that moves registers across a component even if the modified design’s functional equivalence to the original design is unknown.

Command-Line Information

Property: `DistributedPipeliningPriority`
Type: character vector
Value: `'NumericalIntegrity'` | `'Performance'`
Default: `'NumericalIntegrity'`

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

See Also

- `DistributedPipeliningPriority`
- “DistributedPipelining” on page 12-10

Adders

Enable to share adders with the resource sharing optimization.

Settings

Default: Off

☑ On

When resource sharing is enabled, share adders with a bit width greater than or equal to **Adder sharing minimum bitwidth**.

☐ Off

Do not share adders.

Command-Line Information

Property: `ShareAdders`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

- `ShareAdders`

- `AdderSharingMinimumBitwidth`
- "Resource Sharing" on page 15-20

Multipliers

Share multipliers with the resource sharing optimization.

Settings

Default: On

☑ On

When resource sharing is enabled, share multipliers with a bit width greater than or equal to `MultiplierSharingMinimumBitwidth`. For successfully sharing multipliers, the input fixed-point data types must have the same wordlength. The fraction lengths and signs of the fixed-point data types can be different.

☐ Off

Do not share multipliers.

Command-Line Information

Property: `ShareMultipliers`
Type: character vector
Value: `'on' | 'off'`
Default: `'on'`

See Also

- `ShareMultipliers`
- `MultiplierSharingMinimumBitwidth`
- "Resource Sharing" on page 15-20

Multiply-Add blocks

Share Multiply-Add blocks with the resource sharing optimization.

Settings

Default: On

☑ On

When resource sharing is enabled, share Multiply-Add blocks with a bit width greater than or equal to **Multiply-Add block sharing minimum bitwidth**.

☐ Off

Do not share Multiply-Add blocks.

Command-Line Information

Property: `ShareMultiplyAdds`
Type: character vector
Value: `'on' | 'off'`
Default: `'on'`

See Also

- "Resource Sharing" on page 15-20

Atomic subsystems

Share atomic subsystems with the resource sharing optimization.

Settings

Default: On

☑ On

When resource sharing is enabled, share atomic subsystems.

☐ Off

Do not share atomic subsystems.

Command-Line Information

Property: `ShareAtomicSubsystems`
Type: character vector
Value: `'on' | 'off'`
Default: `'on'`

See Also

- `ShareAtomicSubsystems`

- “Resource Sharing” on page 15-20

MATLAB Function blocks

Share MATLAB Function blocks with the resource sharing optimization.

Settings

Default: On

☑ On

When resource sharing is enabled, share MATLAB Function blocks.

☐ Off

Do not share MATLAB Function blocks.

Command-Line Information

Property: `ShareMATLABBlocks`
Type: character vector
Value: `'on' | 'off'`
Default: `'on'`

See Also

- `ShareMATLABBlocks`
- “Resource Sharing” on page 15-20

Floating-point IPs

Share floating-point IP blocks in the target hardware with the resource sharing optimization.

Settings

Default: On

☑ On

When you enable resource sharing, HDL Coder shares floating-point IP blocks. The number of floating-point IP blocks that get shared depends on the **SharingFactor** that you specify for the subsystem.

☐ Off

Do not share floating-point IP blocks.

Command-Line Information
Property: `ShareFloatingPointIP`
Type: character vector
Value: `'on' | 'off'`
Default: `'on'`

See Also

- "Resource Sharing" on page 15-20
- `ShareFloatingPointIP`

Adder sharing minimum bitwidth

Minimum bit width for shared adders.

To share only larger adders with the resource sharing optimization, specify the minimum adder bit width.

Settings

Default: 0

01

No minimum bit width for shared adders.

N, where N is an integer greater than 1

When resource sharing and adder sharing are enabled, share adders with a bit width greater than or equal to N.

Command-Line Information
Property: `AdderSharingMinimumBitwidth`
Type: integer
Value: integer greater than or equal to 0
Default: 0

See Also

- `AdderSharingMinimumBitwidth`

- `ShareAdders`
- "Resource Sharing" on page 15-20

Multiplier sharing minimum bitwidth

Minimum bit width for shared multipliers.

To share only larger multipliers with the resource sharing optimization, specify the minimum multiplier bit width.

Settings

Default: 0

01

No minimum bit width for shared multipliers.

N, where N is an integer greater than 1

When resource sharing and multiplier sharing are enabled, share multipliers with a bit width greater than or equal to N.

Command-Line Information

Property: `MultiplierSharingMinimumBitwidth`
Type: integer
Value: integer greater than or equal to 0
Default: 0

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

See Also

- `MultiplierSharingMinimumBitwidth`
- `ShareMultipliers`
- "Resource Sharing" on page 15-20

Multiplier promotion threshold

Maximum word-length by which HDL Coder promotes a multiplier for sharing with other multipliers.

To share smaller multipliers with other larger multipliers by using the resource sharing optimization, specify the multiplier promotion threshold.

Settings

Default: 0

0

: No difference in word-length between the multipliers. In other words, HDL Coder shares multipliers that have the same word-lengths.

N, where N is an integer greater than 0

: Maximum word-length by which HDL Coder promotes a multiplier for sharing with other multipliers. HDL Coder promotes and shares multipliers with different word-lengths, if the difference in word-lengths is less than or equal to N.

Command-Line Information

Property: `MultiplierPromotionThreshold`
Type: integer
Value: integer greater than or equal to 0
Default: 0

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

See Also

- `MultiplierPromotionThreshold`
- `ShareMultipliers`
- "Resource Sharing" on page 15-20

Multiply-Add block sharing minimum bitwidth

Minimum bit width for shared Multiply-Add blocks.

To share only larger Multiply-Add blocks with the resource sharing optimization, specify the minimum multiplier bit width.

Settings

Default: 0

01

No minimum bit width for shared Multiply-Add blocks.

N, where N is an integer greater than 1

When resource sharing and Multiply-Add block sharing are enabled, share Multiply-Add blocks with a bit width greater than or equal to N.

Command-Line Information

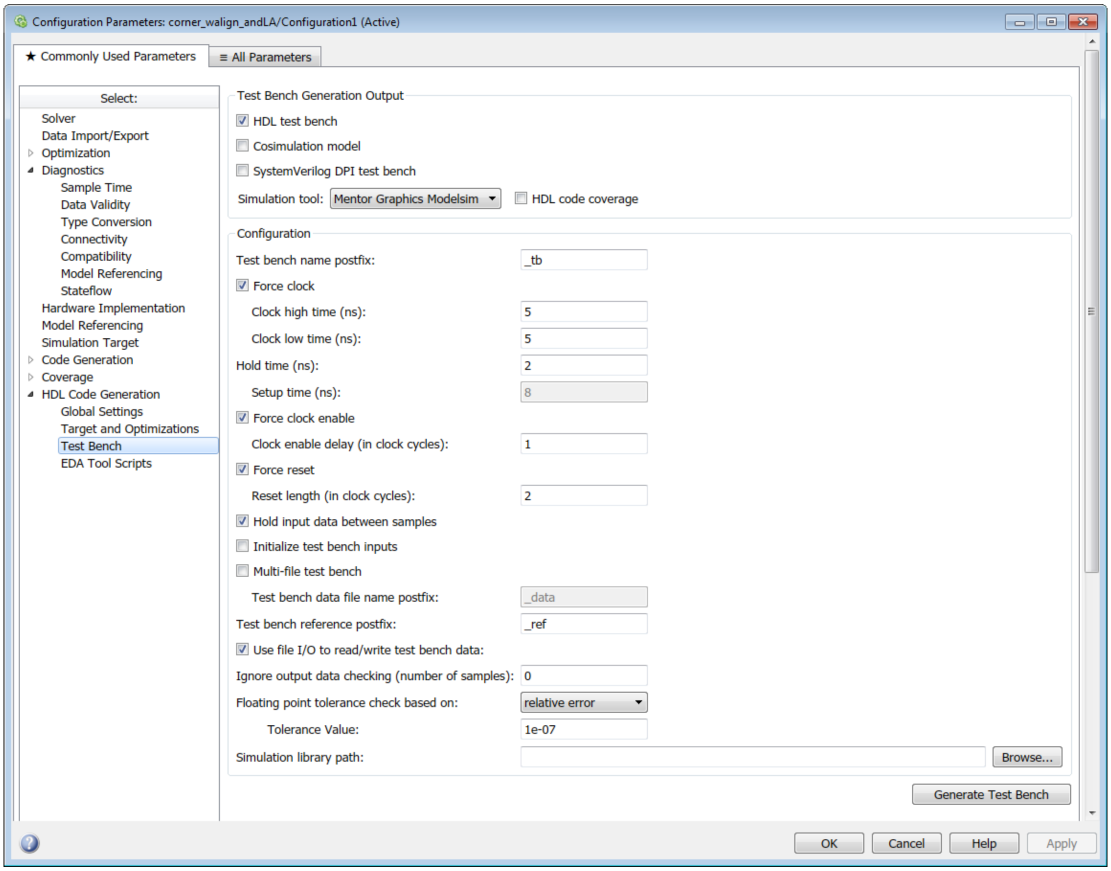
Property: `MultiplierAddSharingMinimumBitwidth`
Type: integer
Value: integer greater than or equal to 0
Default: 0

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

See Also

- "Resource Sharing" on page 15-20

HDL Code Generation Pane: Test Bench



Test Bench Generation Output

- [✓] HDL test bench
- [ ] Cosimulation model
- [ ] SystemVerilog DPI test bench

Simulation tool: Mentor Graphics Modelsim   [ ] HDL code coverage

Configuration

Test bench name postfix:	_tb
[✓] Force clock	
Clock high time (ns):	5
Clock low time (ns):	5
Hold time (ns):	2
Setup time (ns):	8
[✓] Force clock enable	
Clock enable delay (in clock cycles):	1
[✓] Force reset	
Reset length (in clock cycles):	2
[✓] Hold input data between samples	
[ ] Initialize test bench inputs	
[ ] Multi-file test bench	
Test bench data file name postfix:	_data
Test bench reference postfix:	_ref
[✓] Use file I/O to read/write test bench data:	
Ignore output data checking (number of samples):	0
Floating point tolerance check based on:	relative error
Tolerance Value:	1e-07
Simulation library path:	

In this section...

In this section...

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. See also `makehdltb`.

HDL test bench

Enable or disable HDL test bench generation.

Settings

Default: selected

☑ On

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

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

☐ Off

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

Dependencies

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 `makehdl`. To view the property value, use `hdlget_param`.

See Also

`GenerateHDLTestbench`

“Generating VHDL Test Bench Code”

Cosimulation model

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

Settings

Default: not selected

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 `makehdl`. To view the property value, use `hdlget_param`.

See Also

`GenerateCoSimBlock` | `GenerateCoSimModel`

“Generate a Cosimulation Model” on page 18-40

SystemVerilog DPI test bench

Enable or disable generation of the SystemVerilog DPI test bench. Select your HDL simulator at **Simulation tool**. For System Verilog 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 coder generates a direct programming interface (DPI) component for your entire Simulink model, including your DUT and data sources. Your entire model must support C code generation with Simulink Coder. The coder generates a SystemVerilog test bench that compares the output of the DPI component with the output of the HDL implementation of your DUT. The coder also builds shared libraries and generates a simulation script for the simulator you select.

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

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

Settings

Default: not selected

Command-Line Information

Property: `GenerateSVDPITestBench`
Type: character vector
Value: `'ModelSim' | 'Incisive'|'Custom'|'VCS'|'Vivado'`
Default: `'ModelSim'`

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

See Also

`GenerateSVDPITestbenchSimulationTool`

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**.

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 `makehdl`. To view the property value, use `hdlget_param`.

See Also

`SimulationTool`

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

Command-Line Information

Property: `HDLCodeCoverage`
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

`SimulationTool`

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`.

Command-Line Information

Property: `TestBenchPostFix`
Type: character vector
Default: `'_tb'`

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

See Also

`TestBenchPostFix`

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 high time** options.

Command-Line Information

Property: `ForceClock`
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

`ForceClock`

Clock high time (ns)

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

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: `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`.

See Also

`ClockHighTime`

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 `makehdl`. To view the property value, use `hdlget_param`.

See Also

`ClockLowTime`

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.

Command-Line Information

Property: `HoldTime`
Type: integer
Value: positive integer
Default: 2

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

See Also

`HoldTime`

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.

Command-Line Information

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

See Also

`HoldTime`

Force clock enable

Specify whether the test bench forces clock enable input signals.

Settings

Default: On

☑ 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.

☐ Off

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

Dependencies

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

Command-Line Information

Property: `ForceClockEnable`
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

`ForceClockEnable`

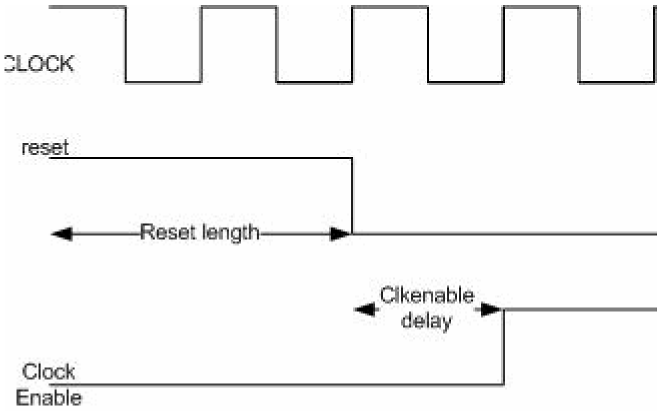
Clock enable delay (in clock cycles)

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).



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 `makehdl`. To view the property value, use `hdlget_param`.

See Also

`TestBenchClockEnableDelay`

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.

Command-Line Information

Property: `ForceReset`
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

`ForceReset`

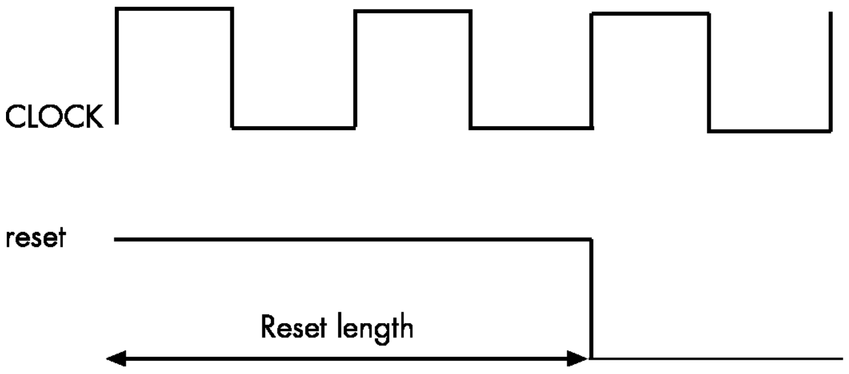
Reset length (in clock cycles)

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 `makehdl`. To view the property value, use `hdlget_param`.

See Also

`ResetLength`

Hold input data between samples

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

Settings

Default: On

☑ On

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

☐ Off

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

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 subrate signals are held valid through each base-rate clock period.

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

Command-Line Information

Property: `HoldInputDataBetweenSamples`
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

`HoldInputDataBetweenSamples`

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).

Command-Line Information

Property: `InitializeTestBenchInputs`
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

`InitializeTestBenchInputs`

Multi-file test bench

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

Settings

Default: Off

☑ On

Write separate files for test bench code, helper functions, and test bench data. The file names are derived from the name of the DUT, the **Test bench name postfix** property, and the **Test bench data file name postfix** property as follows:

`DUTname_TestBenchPostfix_TestBenchDataPostfix`

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

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

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

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

☐ Off

Write a single test bench file containing the HDL test bench code, helper functions, and test bench data.

Dependency

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

Command-Line Information

Property: `MultifileTestBench`

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

`MultifileTestBench`

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.

Command-Line Information

Parameter: `TestBenchReferencePostFix`
Type: character vector
Default: `'_ref'`

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

See Also

TestBenchReferencePostFix

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 `makehdl`. To view the property value, use `hdlget_param`.

See Also

`TestBenchDataPostFix`

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.

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`.

See Also

`UseFileIOInTestBench`

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 20-41)
- When you set the `ResetType` property to `'None'` for the following blocks:
  - commcnvintrlv2/Convolutional Deinterleaver
  - commcnvintrlv2/Convolutional Interleaver
  - commcnvintrlv2/General Multiplexed Deinterleaver
  - commcnvintrlv2/General Multiplexed Interleaver
  - dspsigops/Delay
  - simulink/Additional Math & Discrete/Additional Discrete/Unit Delay Enabled
  - simulink/Commonly Used Blocks/Unit Delay
  - simulink/Discrete/Delay
  - simulink/Discrete/Memory
  - simulink/Discrete/Tapped Delay
  - simulink/User-Defined Functions/MATLAB Function
  - sflib/Chart
  - sflib/Truth Table
- When generating a black box interface to existing manually written HDL code

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`.

See Also

`IgnoreDataChecking`

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.

Command-Line Information

Property: `FPToleranceStrategy`
Type: character vector
Value: `'relative' | 'ULP'`
Default: `'relative'`

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

See Also

`FPToleranceStrategy`

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`.

See Also

`FPToleranceValue`

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.

Command-Line Information

Property: `SimulationLibPath`

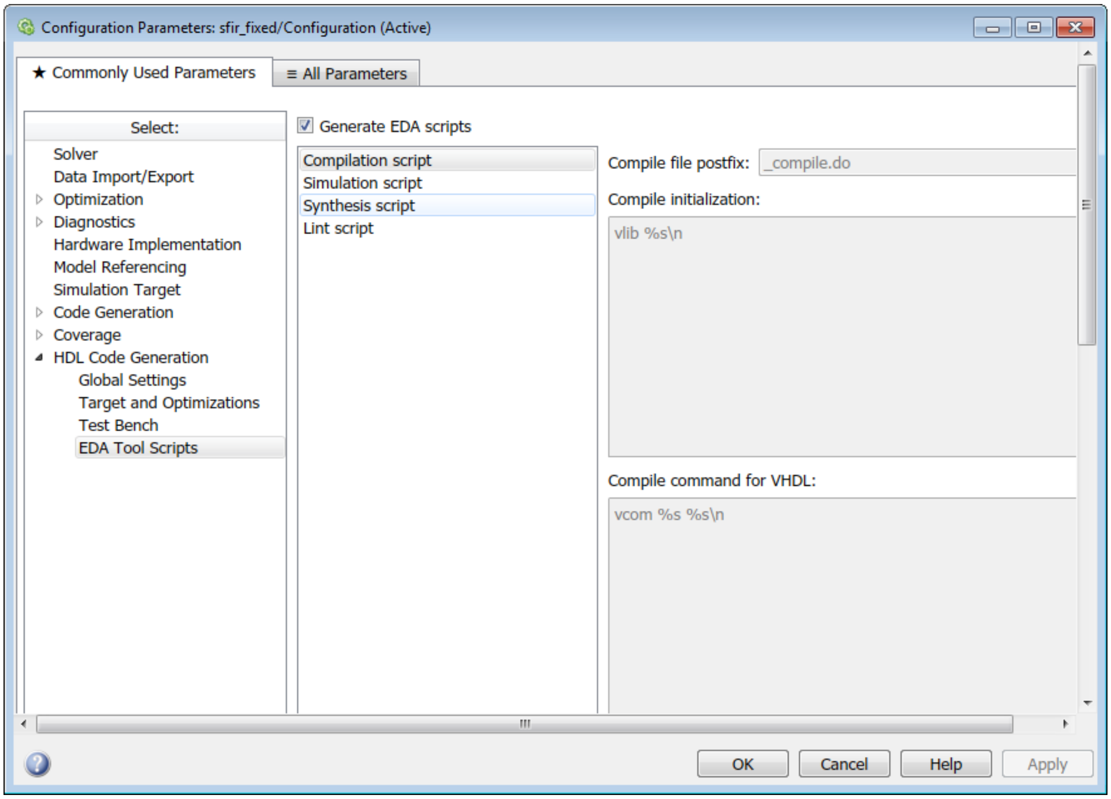
Type: character vector
Default: `''`

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

See Also

`SimulationLibPath`

HDL Code Generation Pane: EDA Tool Scripts


Configuration Parameters: sfir_fixed/Configuration (Active)

★ Commonly Used Parameters | ≡ All Parameters

Select:
Solver
Data Import/Export
Optimization
Diagnostics
Hardware Implementation
Model Referencing
Simulation Target
Code Generation
Coverage
HDL Code Generation
Global Settings
Target and Optimizations
Test Bench
EDA Tool Scripts

[✓] Generate EDA scripts

Compilation script
Simulation script
Synthesis script
Lint script

Compile file postfix: _compile.do

Compile initialization:
vlib %s\n

Compile command for VHDL:
vcom %s %s\n

OK | Cancel | Help | Apply

In this section...

In this section...

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

- "Configure Compilation, Simulation, Synthesis, and Lint Scripts" on page 21-8
- EDAScriptGeneration

Generate multicycle path information

Generate a file that reports multicycle path constraint information.

Settings

Default: Off

☑ On

Generate a text file that reports multicycle path constraint information, for use with synthesis tools.

☐ Off

Do not generate a multicycle path information file.

Command-Line Information

Parameter: `MulticyclePathInfo`
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 Multicycle Path Information Files" on page 13-17
- `MulticyclePathInfo`

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

- "Configure Compilation, Simulation, Synthesis, and Lint Scripts" on page 21-8
- HDLCompileFilePostfix

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 `'VHDLLibraryName'` property, which defaults to`'work'`. You can override the default `Init` string (`'vlib work\n'` by changing the value of `'VHDLLibraryName'`.

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

- "Configure Compilation, Simulation, Synthesis, and Lint Scripts" on page 21-8
- HDLCompileInit

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

- "Configure Compilation, Simulation, Synthesis, and Lint Scripts" on page 21-8
- HDLCompileVHDLCmd

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

- "Configure Compilation, Simulation, Synthesis, and Lint Scripts" on page 21-8
- HDLCompileVerilogCmd

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

- "Configure Compilation, Simulation, Synthesis, and Lint Scripts" on page 21-8
- HDLCompileTerm

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

- "Configure Compilation, Simulation, Synthesis, and Lint Scripts" on page 21-8
- HDLSimFilePostfix

Simulation initialization

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

Settings

Default: The default is

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

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

See Also

- "Configure Compilation, Simulation, Synthesis, and Lint Scripts" on page 21-8
- HDLSimInit

Simulation command

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

Settings

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

If your target language is VHDL, the first implicit argument is the value of the `VHDLLibraryName` property. If your target language is Verilog, the first implicit argument is `'work'`.

The second implicit argument is the top-level module or entity name.

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

- "Configure Compilation, Simulation, Synthesis, and Lint Scripts" on page 21-8
- HDLSimCmd

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

- "Configure Compilation, Simulation, Synthesis, and Lint Scripts" on page 21-8
- HDLSimViewWaveCmd

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

- "Configure Compilation, Simulation, Synthesis, and Lint Scripts" on page 21-8
- HDLSimTerm

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

`HDLSynthTool`

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`.

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

- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 21-8
- HDLSynthFilePostfix

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.

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

- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 21-8
- HDLSynthInit

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** pulldown 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 filename of the entity or module.

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

- "Configure Compilation, Simulation, Synthesis, and Lint Scripts" on page 21-8
- HDLSynthCmd

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.

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

- "Configure Compilation, Simulation, Synthesis, and Lint Scripts" on page 21-8
- HDLSynthTerm

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.

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 17-54
- HDLLintTool

Lint initialization

Enter an initialization text for your HDL lint 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 17-54
- HDLLintInit

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`.

See Also

- "Generate an HDL Lint Tool Script" on page 17-54
- HDLLintCmd

Lint termination

Enter a termination character vector for your HDL lint 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 17-54
- HDLLintTerm

Supported Blocks Library and Block Properties

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 `hdlsupported` library and the following HTML reports:

   ```
   ### HDL supported block list hdlblklist.html
   ### HDL implementation list hdlsupported.html
   ```
2. To see the generated list of blocks, click the `hdlblklist.html` link.

See Also

Functions
`hdllib`

Related Examples

- “Show Blocks Supported for HDL Code Generation” on page 16-45
- “View HDL-Specific Block Documentation” on page 12-3

More About

- “Supported Blocks”

View HDL-Specific Block Documentation

To view HDL-specific documentation for a block in your model:

1. Right-click the block and select **HDL Code** > **HDL Block Properties**.
2. To view the block documentation, click **Help**.

You can also find HDL-specific block documentation in “Supported Blocks”.

See Also

`hdllib`

Related Examples

- “Set and View HDL Block Parameters” on page 12-55
- “Show Blocks Supported for HDL Code Generation” on page 16-45

More About

- “HDL Block Properties” on page 12-4
- “HDL Filter Block Properties” on page 12-36

HDL Block Properties

In this section...

Overview

Block implementation parameters enable you to control details of the code generated for specific block implementations. See “Set and View HDL Block Parameters” on page 12-55 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.

AdaptivePipelining

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 `AdaptivePipelining`.

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.

2 Select **HDL Code** > **HDL Block Properties** .

3 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'`.

To learn how to set model-level delay balancing, see `BalanceDelays`.

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.
`'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 `ClockRatePipelining`.

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.

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 15-57

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 (*`nStages`* defaults to 0)	HDL Coder does not insert pipeline registers.
	`nStages` > 0	The coder inserts *`nStages`* output registers at the output of the subsystem, MATLAB Function block, or Stateflow chart.
`'on'`	Unspecified (*`nStages`* defaults to 0)	The coder does not insert pipeline registers. `DistributedPipelining` has no effect.
	`nStages` > 0	The coder distributes *`nStages`* 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 11-147
- `IgnoreDataChecking`

See Also

- "Distributed Pipelining and Hierarchical Distributed Pipelining" on page 15-47
- "Specify Distributed Pipelining" on page 15-50
- "Distributed Pipeline Insertion for MATLAB Function Blocks" on page 20-41

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_dsp48` 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, Model Variants 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_dsp48 = "yes" *) wire signed [32:0] m4_out1;`

	`'VHDL'`	`attribute use_dsp48 : string ;`
		
		`attribute multstyle of m4_out1 : signal is “dsp“ ;`
			`attribute use_dsp48 : string ;`
			
			`attribute use_dsp48 of m4_out1 : signal is "yes“ ;`

`'off'`	`'Verilog'`	`(* multstyle = "logic“ *) wire signed [32:0] m4_out1;`	`(* use_dsp48 = "no" *) wire signed [32:0] m4_out1;`
	`'VHDL'`	`attribute use_dsp48 : string ;`	`attribute use_dsp48 : string ;`

DSPStyle Value	TargetLanguage Value	SynthesisTool Value	
		`'Altera Quartus II'`	`'Xilinx ISE'` `'Xilinx Vivado'`
		`attribute multstyle of m4_out1 : signal is “logic“ ;`	`attribute use_dsp48 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.

Prerequisites For Hierarchy Flattening

To flatten hierarchy, a subsystem must have the following block properties.

Property	Required value
`DistributedPipelining`	`'off'`
`ClockRatePipelining`	`'off'`
`StreamingFactor`	0
`SharingFactor`	0

To flatten hierarchy, you must also have the `MaskParameterAsGeneric` global property set to `'off'`. For more information, see MaskParameterAsGeneric.

How To Flatten Hierarchy

To set hierarchy flattening using the HDL Block Properties dialog box:

1. Right-click the subsystem.
2. Select **HDL Code** > **HDL Block Properties** .
3. 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 black box implementation or model reference.

- A triggered subsystem when `TriggerAsClock` is enabled.
- A masked subsystem that contains any of the following:
  - Bus.
  - Enumerated data type.
  - Lookup table block: 1-D Lookup Table, 2-D Lookup Table, Cosine, Direct LookupTable (n-D), Prelookup, Sine, n-D Lookup Table.
  - MATLAB Function block.
  - MATLAB System block.
  - Stateflow block: Chart, State Transition Table, Message Viewer.
  - Block with a pass-through or no-op implementation. See “Pass through, No HDL, and Cascade Implementations” on page 12-61.

Note: This option removes subsystem boundaries before code generation. It does not necessarily generate HDL code with a completely flat hierarchy.

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 10-43.

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.
`'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

HandleDenormals Setting	Description
	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.

LatencyStrategy

You can use the `LatencyStrategy` property for certain blocks that support HDL code generation in native floating-point mode. The property specifies whether you want the blocks in your design to map to minimum or maximum latency of the native floating-point 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.

LatencyStrategy Setting	Description
`'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.

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 “Specify the Latency Strategy Setting” on page 10-47.

Set Latency 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. For **LatencyStrategy**, select **inherit**, **Max**, **Min**, or **Zero**.

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', 'LatencyStategy', 'MIN')
```
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, Reciprocal, and so on. By using this setting, you can specify how you want HDL Coder to implement the mantissa multiplication operation for the blocks.

LatencyStrategy Setting	Description
`'inherit'` (default)	Use the mantissa multiply strategy setting of the parent subsystem. If this subsystem

LatencyStrategy Setting	Description
	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. 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', 'MantissaMultiplyStategy', 'PartMultiplierPart
```

See also hdlset_param.

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;
```

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`. See `PipelinePostfix` for an example.

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.
`'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.

If you enable `InstantiateFunctions`, `UseMatrixTypesInHDL` has no effect.

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`.

InstantiateStages Setting	Description
`'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.

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.
`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:

- NCO HDL Optimized
- Gamma Correction
- Lookup Table

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 12-22.

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 "HDL Code Generation Pane: Global Settings" on page 11-21.

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 15-39.

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;
```

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`. See `PipelinePostfix` for an example.

See also "Distributed Pipeline Insertion for MATLAB Function Blocks" on page 20-41.

ResetType

Use the `ResetType` block parameter to suppress reset logic generation.

ResetType Value	Description
`default`	Generate reset logic.
`none`	Do not generate reset logic. 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. 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 IgnoreDataChecking.

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 `ResetType`.

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 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 12-42.

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 15-20.

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.

SoftReset Value	Description
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 15-7.

UseMatrixTypesInHDL

The `UseMatrixTypesInHDL` block property specifies whether to generate 2-D matrices in HDL code when you have MATLAB matrices in your MATLAB Function block.

UseMatrixTypesInHDL Setting	Description
`off` (default)	Generate HDL vectors with index computation logic for MATLAB matrices. This option can use more area in the synthesized hardware.
`on`	Generate HDL matrices for MATLAB matrices. This option can save area in the synthesized hardware. The following requirements apply: • You cannot use matrices at the block input or output ports. • Matrix elements cannot be complex or `struct` data types. • You cannot use linear indexing to specify matrix elements. For example, if you have a 3x3 matrix, `A`, you cannot use `A(4)`. Instead, use `A(2,1)`.

UseMatrixTypesInHDL Setting	Description
	You can also use a colon operator in either the row or column subscript, but not both. For example, you can use `A(3,1:3)` and `A(2:3,1)`, but not `A(2:3, 1:3)`. • `InstantiateFunctions` must be set to `'off'`. If you enable `InstantiateFunctions`, `UseMatrixTypesInHDL` has no effect. • `UseMatrixTypesInHDL` has no effect if you have System objects in your MATLAB code.

To generate 2-D matrices in HDL code:

1 Right-click the MATLAB Function block and select **HDL Code** > **HDL Block Properties**.
2 For **UseMatrixTypesInHDL**, select **on**.

Alternatively, at the command line, use `makehdl` or `hdlset_param` to set the `UseMatrixTypesInHDL` block property to `'on'`.

For example, suppose you have a model, `myModel`, with a subsystem, `dutSubsys`, that contains a MATLAB Function block, `myMLFcn`. To generate 2-D matrices in HDL code for myMLFcn, enter:

```
hdlset_param('myModel/dutSubsys/myMLFcn', 'UseMatrixTypesInHDL', 'on')
```

UseRAM

The `UseRAM` implementation parameter enables using RAM-based mapping for a block instead of mapping to a shift register.

UseRAM Setting	Mapping Behavior
`off`	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 12-34.
`on`	The delay maps to a dual-port RAM block when all of the following conditions are true:

UseRAM Setting	Mapping Behavior
	• Initial value of the delay is zero. • Delay length > 4. • Delay has one of the following set of numeric and data type attributes: • (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) • `RAMSize` is greater than or equal to the `RAMMappingThreshold` value. `RAMSize` is the product `DelayLength * WordLength * ComplexLength`. • `DelayLength` is the number of delays that the Delay block specifies. • `WordLength` is the number of bits that represent the data type of the delay. • `ComplexLength` is 2 for complex signals; 1 otherwise. 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 12-31.

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:

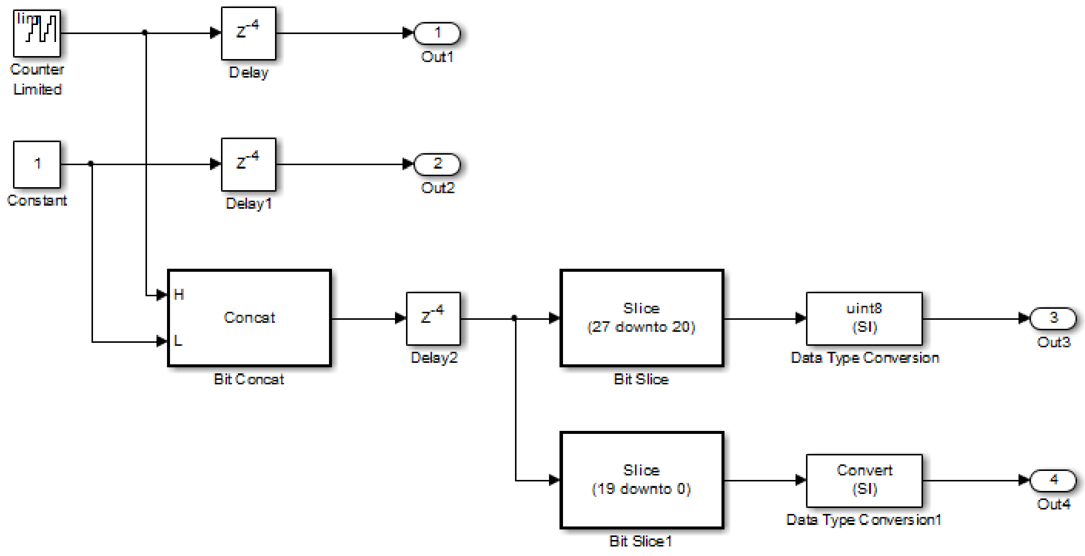
- Increased occupancy on a single RAM
- Sharing of address generation logic, which minimizes duplication of identical HDL code
- Mapping of delays to a RAM when the *individual* delays do not satisfy the threshold

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

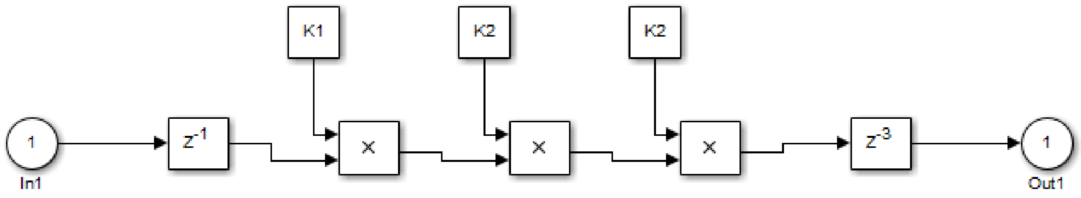
When `UseRAM` is `off` for a Delay block, HDL Coder maps the delay to a shift register by default. However, the coder changes the `UseRAM` setting to `on` and tries to map the delay to a RAM under the following conditions:

- 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	No

`UseRAM` Setting for the Delay Block	Optimizations Enabled for Subsystem with Delay Block		
	Distributed Pipelining Only	Streaming Only	Both Distributed Pipelining and Streaming
		of a shift register can provide an area-efficient design.	

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 8-12.

HDL Filter Block Properties

In this section...

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 12-49.

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 12-44.

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 (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 12-52.

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	`FL = length(find(Hd.numerator ~= 0))`
Direct-form asymmetrical FIR, direct-form symmetrical FIR	`FL = ceil(length(find(Hd.numerator ~= 0))/2)`

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) Calculation** column to the right. Specify the LUT partition as a vector of integers whose elements sum to `FL`.	`FL = size(polyphase(Hm), 2)`

Matrix: Determine the subfilter length `FL`*i* based on the polyphase decomposition of the filter, as shown in the **Filter Length (FL) Calculation** column to the right. Specify the LUT partition for each subfilter as a row vector whose elements sum to `FL`*i*.	`p = polyphase(Hm);`
	`FL`*i* `= length(find(p(`*i*`,:)));`
	
	where *i* is the index to the *i*th row of the polyphase matrix of the multirate filter. The *i*th row of the matrix `p` represents the *i*th subfilter.

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 `mod(W, log2(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 12-53.

See “Distributed Arithmetic for HDL Filters” on page 12-52.

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 12-41. 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 12-44.

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 12-44.

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 12-40 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 12-44

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
Partly serial	`[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: • 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

To Generate This Architecture...	Set SerialPartition to...
	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 12-44.

This property is also used for Min/Max blocks with cascade-serial architectures. For how to configure Min/Max cascades, see “SerialPartition” on page 12-27.

More About

- “Set and View HDL Block Parameters” on page 12-55
- “HDL Block Properties” on page 12-4

HDL Filter Architectures

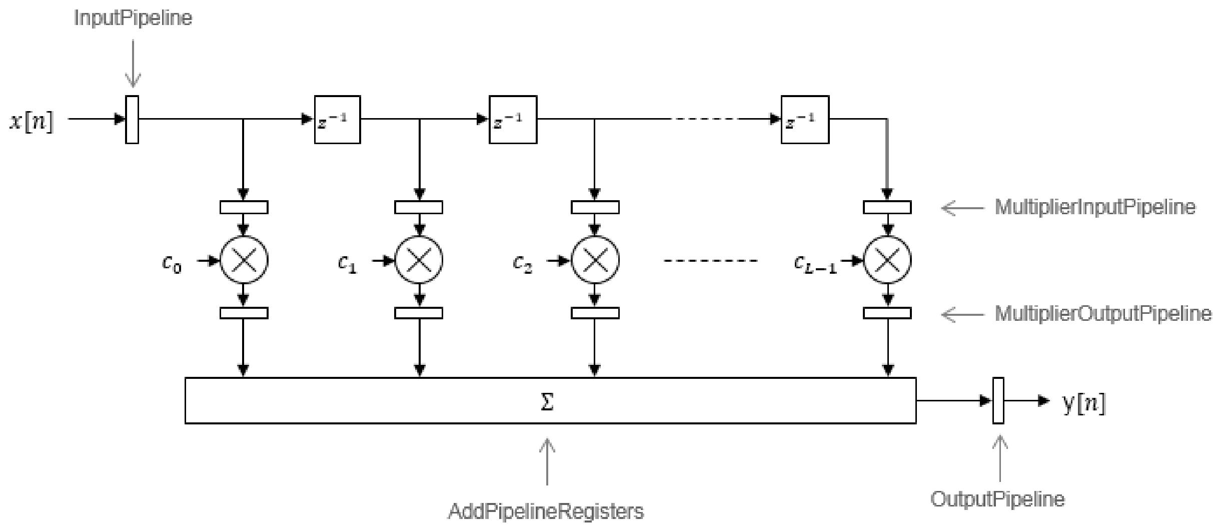
The HDL Coder software provides architecture options that extend your control over speed vs. area tradeoffs in the realization of filter designs. To achieve the desired tradeoff for generated HDL code, you can either specify a fully parallel architecture, or choose one of several serial architectures. Configure a serial architecture using the “SerialPartition” on page 12-42 and “ReuseAccum” on page 12-41 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 12-36 for scalar input filters, and “AdderTreePipeline” on page 12-36 for frame-based filters. Specify pipeline stages before and after each multiplier with MultiplierInputPipeline on page 12-41 and MultiplierOutputPipeline on page 12-41. Set the number of pipeline stages before and after the filter using “InputPipeline” on page 12-19 and “OutputPipeline” on page 12-24. The architecture diagrams show the locations of the various configurable pipeline stages.

Fully Parallel Architecture

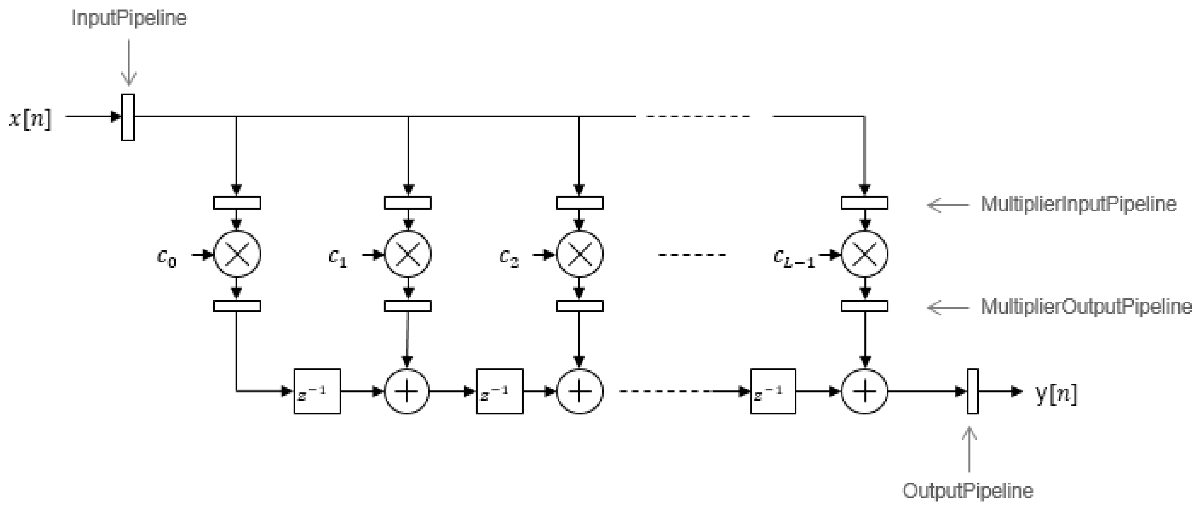
This option is the default architecture. A fully parallel architecture uses a dedicated multiplier and adder for each filter tap. The taps execute in parallel. A fully parallel architecture is optimal for speed. However, it requires more multipliers and adders than a serial architecture, and therefore 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 12-42 and "ReuseAccum" on page 12-41 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 would use 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 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 will be 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. For example, you could specify a four-tap filter with two partitions, each having two taps. The system clock would run 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 non-cascade 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.

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

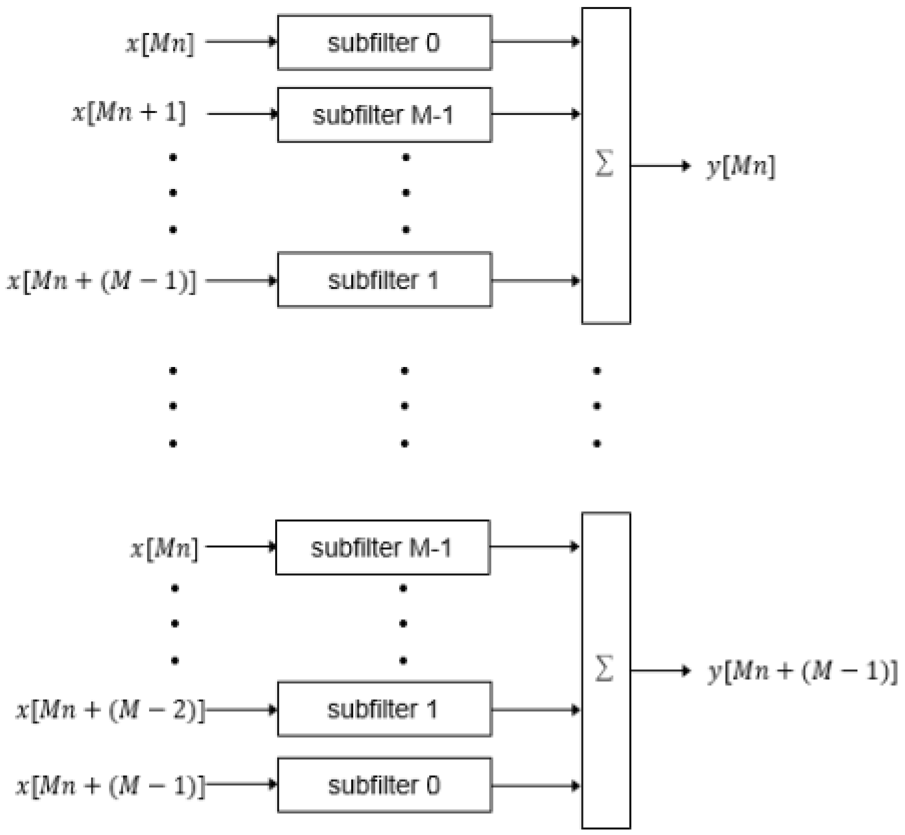
Serialization of a filter increases the total latency of the design by one clock cycle. The serial architectures use an accumulator (an adder with a register) to add the products sequentially. An additional final register is used to store the summed result of all the serial partitions, requiring an extra clock cycle for the operation. To handle 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.

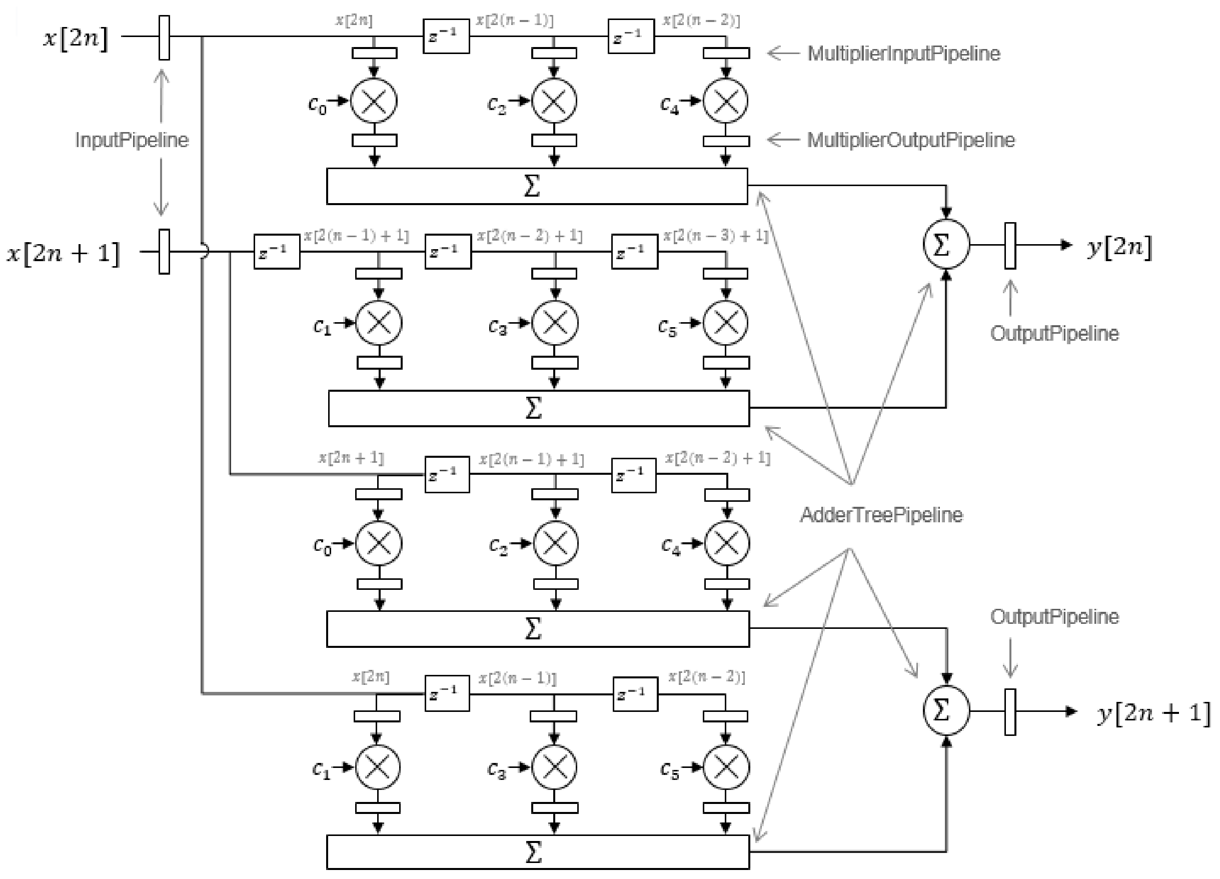


Each of the subfilters includes every *M*th coefficient. The subfilter results are added so that each output sample is the sum of each of the coefficients multiplied with one input sample.

$$\begin{aligned}
&\text{subfilter } 0 = c_0,\ c_M,\ \ldots\\
&\text{subfilter } 1 = c_1,\ c_{M+1},\ \ldots\\
&\ldots\\
&\text{subfilter } M\text{–}1 = c_{M-1},\ c_{2M-1},\ \ldots
\end{aligned}$$

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 12-36 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 12-41 and MultiplierOutputPipeline on page 12-41.



For symmetric or antisymmetric coefficients, the filter architecture reuses the coefficient multipliers and adds design delay between the multiplier and summation stages as required.

More About

- "HDL Filter Block Properties" on page 12-36
- "Distributed Arithmetic for HDL Filters" on page 12-52

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 12-38
- Improve Performance with Parallelism: “DARadix ” on page 12-39

For information on the theoretical foundations of DA, see “Further References” on page 12-53.

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 Block Parameters

In this section...

For a list of HDL block properties, see "HDL Block Properties" on page 12-4.

Set HDL Block Parameters from the GUI

You can view and set HDL-related block properties, such as implementation and implementation parameters, at the individual block level. To open the HDL Properties dialog box:

1 Right-click the block and select **HDL Code** > **HDL Block Properties**.

   The HDL Properties dialog box opens.

2 Modify the block properties as desired.

3 Click OK.

Set HDL Block Parameters from the Command Line

`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 architecture for the same block, enter the following at the command line:

```
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]
```

See also `hdlset_param` and `hdlget_param`.

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
```

See also `hdldispblkparams`.

Set HDL Block Parameters for Multiple Blocks

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.

See the Simulink documentation for detailed information about find_system (Simulink).

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.

1 Open the `sfir_fixed` model.
2 Click on the `sfir_fixed/symmetric_fir` subsystem to select it.
3 Locate all Product blocks within the subsystem as follows:

```
 prodblocks = find_system(gcb, 'BlockType', 'Product')

prodblocks = 

    'sfir_fixed/symmetric_fir/Product'
    'sfir_fixed/symmetric_fir/Product1'
    'sfir_fixed/symmetric_fir/Product2'
    'sfir_fixed/symmetric_fir/Product3'
```

4 Set the output pipeline depth to 2 for all selected blocks.

```
for ii=1:length(prodblocks), hdlset_param(prodblocks{ii}, 'OutputPipeline', 2), end;
```

5 To verify the settings, display the value of the `OutputPipeline` parameter for the blocks .

```
 for ii=1:length(prodblocks), hdlget_param(prodblocks{ii},  'OutputPipeline'), end;

ans =

     2


ans =

     2


ans =

     2


ans =
```

2

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'
```

Pass through, No HDL, and Cascade Implementations

Pass-through and No HDL Implementations

Implementation	Description
Pass-through implementations	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: • Convert 1-D to 2-D • Reshape • Signal Conversion • Signal Specification
No HDL	The `NoHDL` 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. You can also use this implementation as an alternative implementation for subsystems.

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
```

Generating HDL Code for Multirate Models

Code Generation from Multirate Models

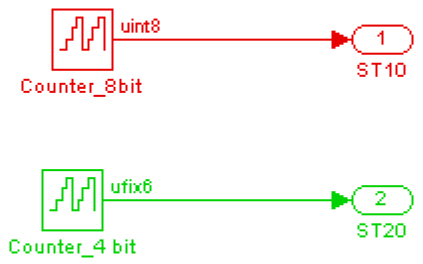
HDL Coder supports HDL code generation for single-clock and multiple clock multirate models. Your model can include blocks running at multiple sample rates:

- 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.

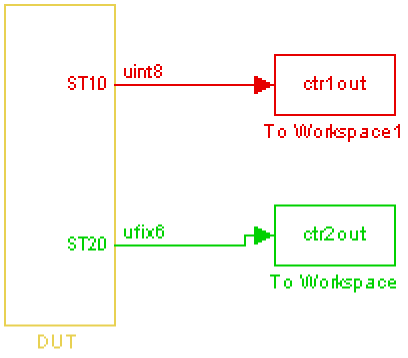
In general, generating HDL code for a multirate model does not differ greatly from generating HDL code for a single-rate model. However, there are a few requirements and restrictions on the configuration of the model and the use of specialized blocks (such as Rate Transitions) that apply to multirate models. For details, see “Multirate Model Requirements for HDL Code Generation” on page 13-7.

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 subrate 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.

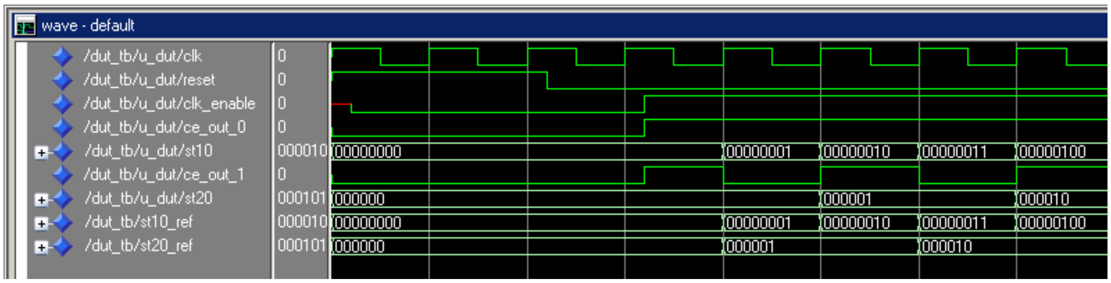


The following listing shows the VHDL entity declaration generated for the DUT.

```
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.


wave - default
/dut_tb/u_dut/clk 0
/dut_tb/u_dut/reset 0
/dut_tb/u_dut/clk_enable 0
/dut_tb/u_dut/ce_out_0 0
/dut_tb/u_dut/st10 000010 00000000 00000001 00000010 00000011 00000100
/dut_tb/u_dut/ce_out_1 0
/dut_tb/u_dut/st20 000101 000000 000001 000010
/dut_tb/st10_ref 000010 00000000 00000001 00000010 00000011 00000100
/dut_tb/st20_ref 000101 000000 000001 000010

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 subrate 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.

To learn more, see “Using Multiple Clocks in HDL Coder™” on page 13-26.

Generate Reset for Timing Controller

In this section...

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

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.

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?

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

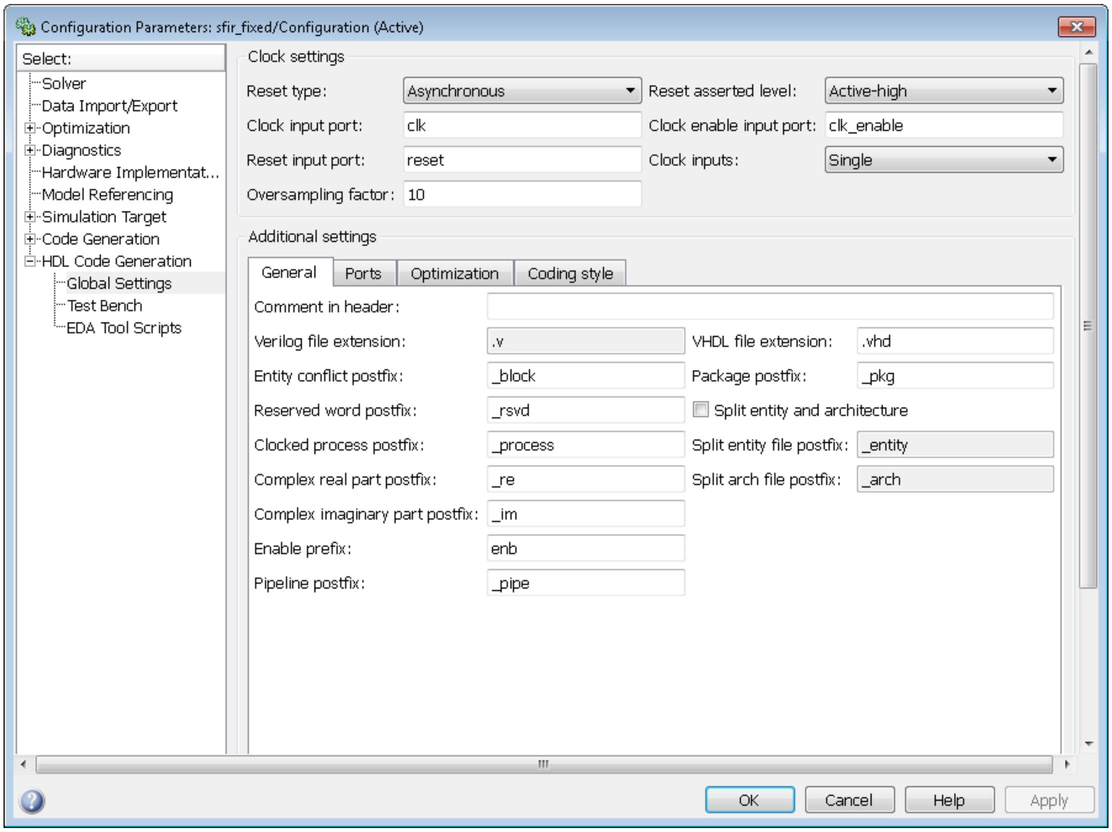
rate. For more information, see “Resolving Oversampling Rate Conflicts” on page 13-11 .

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.



Clock settings

Reset type:	Asynchronous	Reset asserted level:	Active-high
Clock input port:	clk	Clock enable input port:	clk_enable
Reset input port:	reset	Clock inputs:	Single
Oversampling factor:	10		

Additional settings

Comment in header:			
Verilog file extension:	.v	VHDL file extension:	.vhd
Entity conflict postfix:	_block	Package postfix:	_pkg
Reserved word postfix:	_rsvd	☐ Split entity and architecture	
Clocked process postfix:	_process	Split entity file postfix:	_entity
Complex real part postfix:	_re	Split arch file postfix:	_arch
Complex imaginary part postfix:	_im		
Enable prefix:	enb		
Pipeline postfix:	_pipe		

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

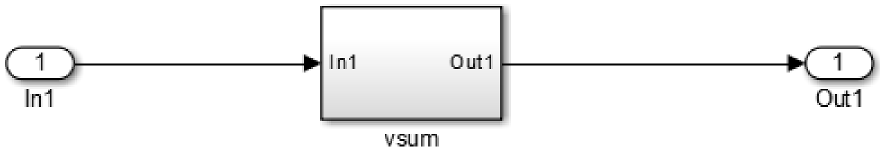
You can specify the oversampling factor for a global clock from the command line by setting the `Oversampling` property with `hdlset_param` or `makehdl`. The following example specifies an oversampling factor of 7:

```
>> 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.
```

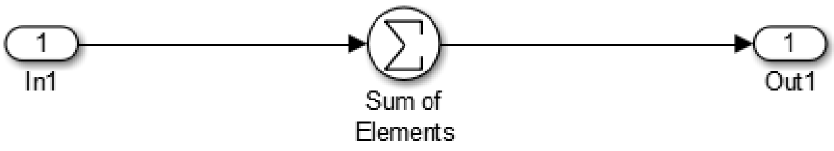
Resolving Oversampling Rate Conflicts

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 (`SumCascadeHDLEmission`) 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.
...
```

Use Trigger As Clock in Triggered Subsystems

In this section...

When To Use 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.

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.

Requirements For Using Trigger As Clock

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.

How To Specify Trigger As Clock

Using the Configuration Parameters Dialog Box

In **HDL Code Generation** > **Global Settings** > **Optimization** tab, select **Use trigger signal as clock**.

Using the HDL Workflow Advisor

In the **HDL Code Generation** > **Set Code Generation Options** > **Set Advanced Options** > **Optimization** tab, select **Use trigger signal as clock**.

At the Command Line

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 When Using Trigger As Clock

Using the trigger as clock for triggered subsystems can result in timing mismatches of one cycle during testbench simulation.

Generate Multicycle Path Information Files

In this section...

Overview

HDL Coder implements multirate systems in HDL by generating a master clock running at the model's base rate, and generating subrate timing signals from the master clock (see also "Code Generation from Multirate Models" on page 13-2). The propagation time between two subrate 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; TO : Sbs.booloreg; PATH_MULT : 2; RELATIVE_CLK : source,
   Sbs.clk;
FROM : Sbs.boolireg_v<0>; TO : Sbs.booloreg_v<0>; PATH_MULT : 2;
   RELATIVE_CLK : source, Sbs.clk;
FROM : Sbs.doubireg; TO : Sbs.douboreg; PATH_MULT : 2; RELATIVE_CLK : source,
   Sbs.clk;
FROM : Sbs.doubireg_v<0>; TO : Sbs.douboreg_v<0>; PATH_MULT : 2;
   RELATIVE_CLK : source, Sbs.clk;
FROM : Sbs.intireg(7:0); TO : Sbs.intoreg(7:0); PATH_MULT : 2;
   RELATIVE_CLK : source, Sbs.clk;
FROM : Sbs.intireg_v<0>(7:0);TO : 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
`FROM :` *`src_reg_path`*`;`	The source (or FROM) register of a multicycle path in the system. The value of *`src_reg_path`* is the HDL path of the source register's output signal. See also “Register Path Syntax for FROM : and TO : Fields” on page 13-20 .
`TO :` *`dst_reg_path`*`;`	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 *`dst_reg_path`* is the HDL path of the destination register's output signal. See also “Register Path Syntax for FROM : and TO : Fields” on page 13-20.

Keyword : field (or subfields)	Field Description
`PATH_MULT : N;`	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). The path multiplier value *N* indicates that the signal has N clock cycles of its relative clock to propagate from source to destination register. The coder does not report register-to-register paths where N = 1, because this is the default path multiplier.
`RELATIVE_CLK : relclock, sysclock;`	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: • *`relclock`*: Since HDL Coder currently generates only single-clock systems, this subfield takes the value `source`. 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 `source` or `destination`. This usage is reserved for future release of the coder. • *`sysclock`*: This subfield is intended for use with synthesis tools that require the actual propagation time for a multicycle path. `sysclock` provides the path to the system's top-level clock (e.g., `Sbs.clk`) 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.initreg`.
- `<`*`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
`FROM : Sbs.intireg(7:0); TO : Sbs.intoreg(7:0);`	Both signals are fixed point and eight bits wide.

Path	Description
`FROM : Sbs.intireg; TO : Sbs.intoreg;`	Both signals are either boolean or double.
`FROM : Sbs.intireg<0>(7:0); TO : Sbs.intoreg<1>(7:0);`	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.
`FROM : Sbs.u_H1.intireg(7:0); TO : Sbs.intoreg(7:0);`	The signal `intireg` is defined in the module `H1`, and `H1` is inside the module `Sbs`. `u_H1` is the instance name of `H1` in `Sbs`. 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: clock_name`	Each clock in the design has a CLOCK definition line.
`PERIOD: float_value`	The Simulink rate (floating point value) associated with this CLOCK.

Keyword : field (or subfields)	Field Description
`BASE_CLOCK: base_clock_name`	Names the master clock. This field does not appear on the master clock.
`MULTIPLIER: int_value`	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

By default, HDL Coder does not generate multicycle path information files. To enable generation of multicycle path information files, select **Generate multicycle path information** in the **HDL Code Generation** > **EDA Tool Scripts** pane of the Configuration Parameters dialog box.

When you select **Generate multicycle path information**, the coder generates a multicycle path information file each time you initiate code generation.

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
SubsystemBlackBoxHDLInstiation	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.

Using Multiple Clocks in HDL Coder™

This example shows how to instantiate multiple top-level synchronous clock input ports in HDL Coder.

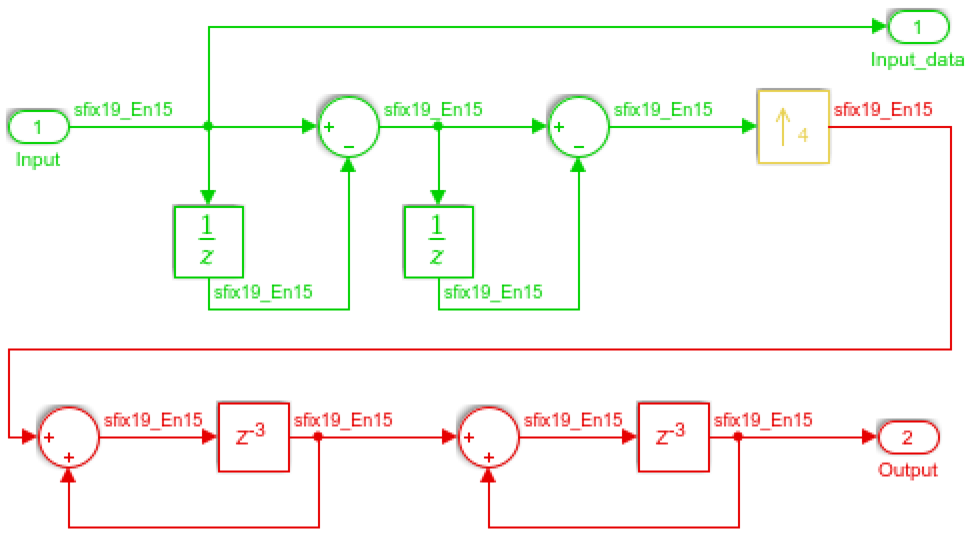
Overview of Clocking Modes

HDL Coder has two clocking modes: one that generates a single clock input to the Device Under Test (DUT), and one that will generate 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 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 corresponding to a separate rate in the model. Transitions between rates often 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 will have a set of three signals that do not appear in the Simulink diagram added to it. Collectively these are a clock bundle, containing 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'.
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_clockdemo'.
### Working on DUT_tc as hdlsrc\hdlcoder_clockdemo\DUT_tc.vhd.
### Working on hdlcoder_clockdemo/DUT as hdlsrc\hdlcoder_clockdemo\DUT.vhd.
### Generating package file hdlsrc\hdlcoder_clockdemo\DUT_pkg.vhd.
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\BF
### Creating HDL Code Generation Check Report file://C:\TEMP\BR2017ad_533001_8924\IB_CF
### HDL check for 'hdlcoder_clockdemo' complete with 0 errors, 0 warnings, and 1 messag
### 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 as shown here, 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' parameter must be set to 'multiple'. This may be done on the makehdl command line or by changing the "Clock inputs" setting to "Multiple" on the HDL Configuration Parameters Global Settings tab.

```
makehdl('hdlcoder_clockdemo/DUT', 'ClockInputs', 'multiple');
```

```
### Generating HDL for 'hdlcoder_clockdemo/DUT'.
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_clockdemo'.
### Working on DUT_tc_d1 as hdlsrc\hdlcoder_clockdemo\DUT_tc_d1.vhd.
### Working on hdlcoder_clockdemo/DUT as hdlsrc\hdlcoder_clockdemo\DUT.vhd.
### Generating package file hdlsrc\hdlcoder_clockdemo\DUT_pkg.vhd.
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\BR
### Creating HDL Code Generation Check Report file://C:\TEMP\BR2017ad_533001_8924\IB_CR
### HDL check for 'hdlcoder_clockdemo' complete with 0 errors, 0 warnings, and 1 messag
### HDL code generation complete.
```

Clock Summary Information in Multiclock Mode

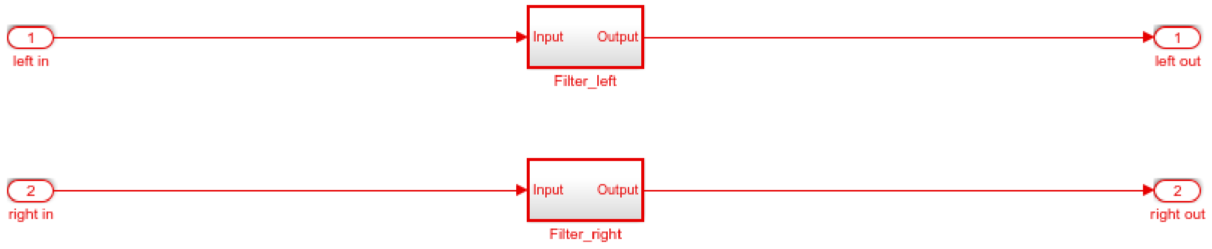
The contents of the Clock Summary are different in multiple clock mode. The report now contains a clock table. This table has one entry for each primary DUT clock. It describes the relative clock ratio between each clock and the fastest clock in the model. As with single clock mode, this information 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'.
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_audiofiltering'.
### Working on hdlcoder_audiofiltering/Audio filter/Filter_left as hdlsrc\hdlcoder_aud
### Working on hdlcoder_audiofiltering/Audio filter as hdlsrc\hdlcoder_audiofiltering\
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\BR
### Creating HDL Code Generation Check Report file://C:\TEMP\BR2017ad_533001_8924\IB_CR
### HDL check for 'hdlcoder_audiofiltering' complete with 0 errors, 0 warnings, and 0 r
### HDL code generation complete.
```

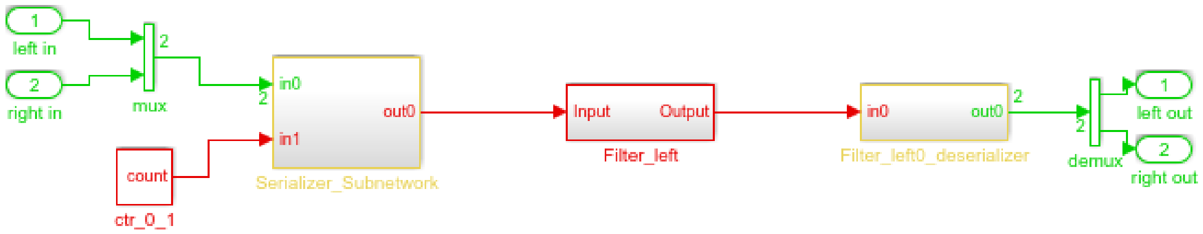


Using Multiple Clock Mode with Resource Sharing

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. The Clock Summary Report shows the timing information for the two clocks.

```
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'.
### Starting HDL check.
### The DUT requires an initial pipeline setup latency. Each output port experiences th
### 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 = (
### Working on hdlcoder_audiofiltering/Audio filter/Filter_left as hdlsrc\hdlcoder_audi
### Working on hdlcoder_audiofiltering/Audio filter as hdlsrc\hdlcoder_audiofiltering\A
### Generating package file hdlsrc\hdlcoder_audiofiltering\Audio_filter_pkg.vhd.
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\BR
### Creating HDL Code Generation Check Report file://C:\TEMP\BR2017ad_533001_8924\IB_CR
### HDL check for 'hdlcoder_audiofiltering' complete with 0 errors, 0 warnings, and 1 m
### HDL code generation complete.
```



Generating Bit-True Cycle-Accurate Models

Generated Model and Validation Model

In this section...
"Generated Model" on page 14-2 "Validation Model" on page 14-3

Generated Model

Before generating code, HDL Coder creates a behavioral model of the HDL code, called the *generated model*. The generated model uses HDL-specific block implementations, and it implements the area and speed optimizations that you specify in your Simulink model.

The generated model is an intermediate 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.

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`*.

Highlight Color	Delay Type
Cyan	Block implementation RAM mapping
Green	Constrained output pipelining
Orange	Distributed pipelining Input and output pipelining Delay balancing Clock-rate pipelining

Customize the Generated Model

To customize the generated model, use the following properties with `makehdl` or `hdlset_param`:

- `GeneratedModelName`

- `GeneratedModelNamePrefix`
- `CodeGenerationOutput`

If you do not want the code generator to create the generated model, set `GenerateModel` to `off` .

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 with 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, with compensating delays inserted.
- Original inputs, routed to both the original model and 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:

- The Newton-Raphson method is an approximation, so 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 14-5.

Generate A Validation Model

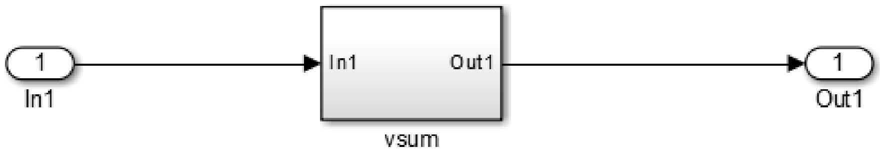
To generate a validation model:

- In the Configuration Parameters dialog box, in the **HDL Code Generation** pane, enable **Generate 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`.

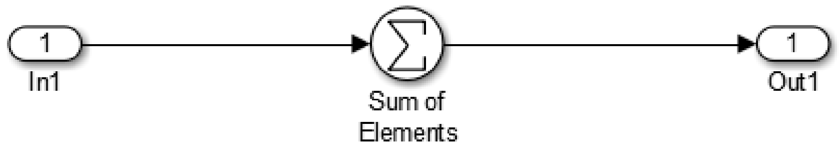
Locate Numeric Differences After Speed Optimization

This example first selects a speed-optimized Sum block implementation for simple model that computes a vector sum. It then examines a generated model and locates the numeric changes introduced by the optimization.

The model, `simplevectorsum_tree`, 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 model is configured to use the `Tree` implementation when generating HDL code for the Sum block within the `vsum` subsystem. This implementation, optimized for minimal latency, generates a tree-shaped structure of adders for the Sum block.

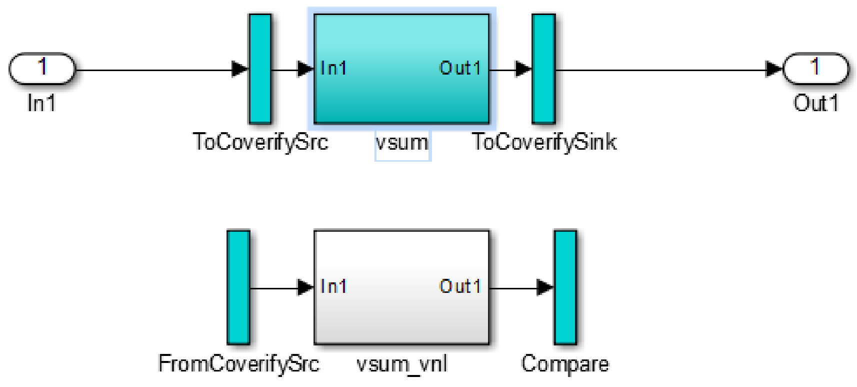
To select a nondefault implementation for an individual block:

1 Right-click the block and select **HDL Code** > **HDL Block Properties** .

2 In the HDL Properties dialog box, select the desired implementation from the **Architecture** menu.

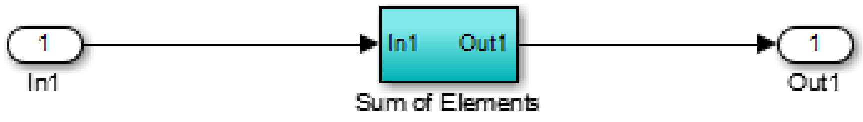
3 Click **Apply** and close the dialog box.

After code generation, you can view the validation model, `gm_simplevectorsum_tree_vnl`.

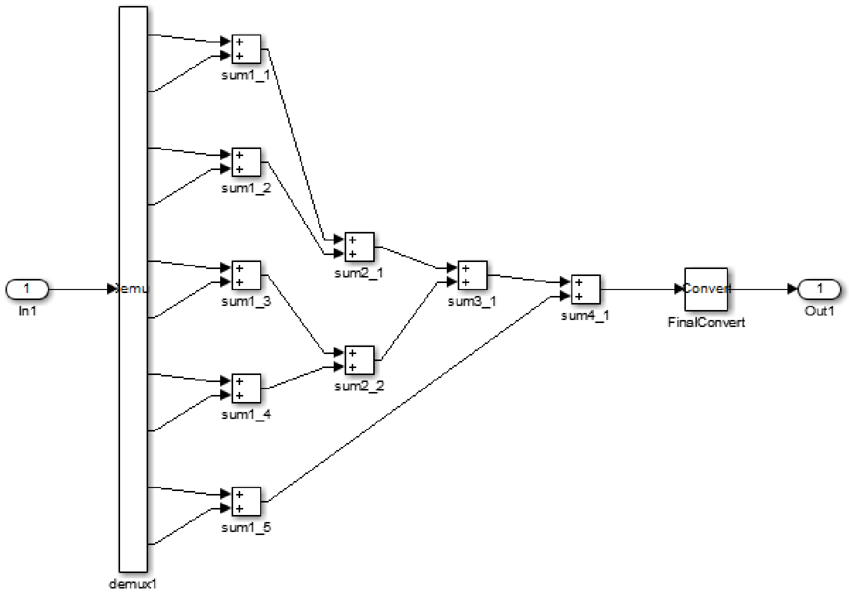


The `vsum` subsystem has been highlighted in cyan. This highlighting indicates that the subsystem differs in some respect from the `vsum` subsystem of the original model.

The following figure shows the `vsum` subsystem in the generated model. Observe that the Sum block is now implemented as a subsystem, which also appears highlighted.



The following figure shows the internal structure of the Sum subsystem.



The generated model implements the vector sum as a tree of adders (Sum blocks). The vector input signal is demultiplexed and connected, as five pairs of operands, to the five leftmost adders. The widths of the adder outputs increase from left to right, as required to avoid overflow in computing intermediate results.

Optimization

Automatic Iterative Optimization

In this section...

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`

Optimization with Constrained Overclocking

In this section...

Note: **Maximum Oversampling Ratio** and **Max computation latency** are not recommended. Use clock-rate pipelining with `Oversampling` instead.

Why Constrain Overclocking?

Overclocking can cause your design clock rate to exceed the maximum clock rate of your target hardware when your original clock rate is high. Without constrained overclocking, automated speed and area optimizations can modify the design implementation architecture and often result in local upsampling.

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

When to Use Constrained Overclocking

When using area and speed optimizations, you can specify constraints on overclocking using the `MaxOversampling` and `MaxComputationLatency` parameters. 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 `MaxOversampling`. You can use `MaxComputationLatency` to give 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.

Set Overclocking Constraints

You can use the `MaxOversampling` and `MaxComputationLatency` parameters to constrain overclocking when optimizing area and speed.

The following table shows how to set `MaxOversampling` and `MaxComputationLatency` for different design implementation results:

Desired implementation result	Without Optimizations	With Optimizations
Unlimited overclocking	`MaxOversampling` = 0	`MaxOversampling` = 0 `MaxComputationLatency` > 1
Overclocking with constraints	`MaxOversampling` > 1	`MaxOversampling` > 1 `MaxComputationLatency` > 1
No overclocking (single rate)	`MaxOversampling` = 1	`MaxOversampling` = 1 `MaxComputationLatency` > 1

Constrained Overclocking Limitations

When you constrain overclocking, the following limitations apply:

- If you set `MaxOversampling` = 1, your DUT must be single-rate.
- Loop streaming and RAM mapping are disabled when you set `MaxOversampling` = 1, even if `MaxComputationLatency` > 1.

Streaming

In this section...

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, the coder generates *fully parallel* implementations for vector computations. For example, the coder 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.

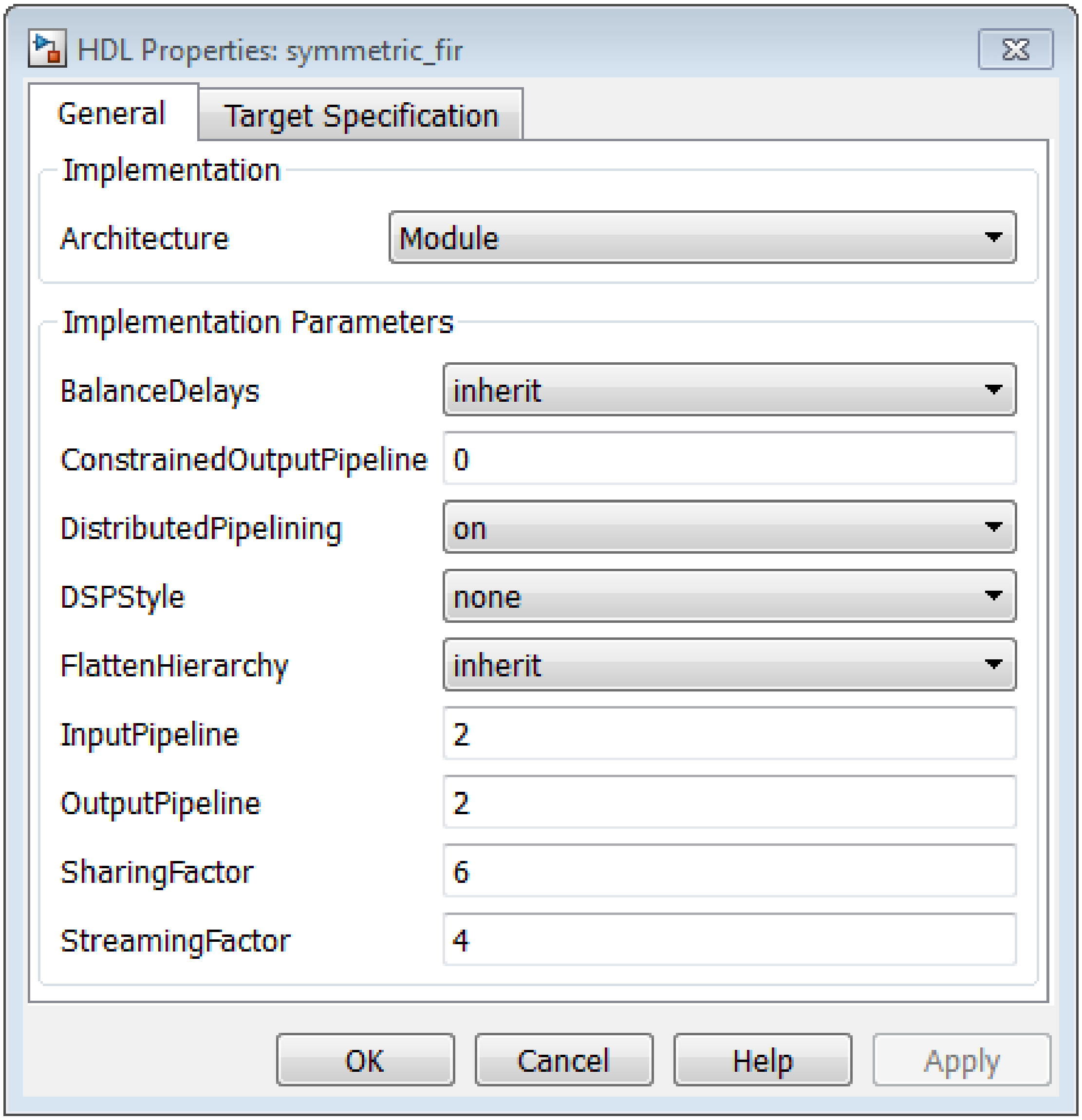
If you know the maximal vector dimensions and the sample rate for a subsystem, you can compute the possible streaming factors and resulting sample rates for the subsystem. However, even if the requested streaming factor is mathematically possible, the subsystem must meet other criteria for streaming. See "Requirements and Limitations for Streaming" on page 15-10 for details.

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 coder 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 15-66.

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 14-2.

Specify Streaming

You apply streaming at the subsystem level. Specify the streaming factor by setting the subsystem HDL parameter `StreamingFactor`. You can set `StreamingFactor` in the HDL Properties dialog box for a subsystem, as shown in the following figure.


HDL Properties: symmetric_fir

General | Target Specification

Implementation

Architecture	Module

Implementation Parameters

BalanceDelays	inherit
ConstrainedOutputPipeline	0
DistributedPipelining	on
DSPStyle	none
FlattenHierarchy	inherit
InputPipeline	2
OutputPipeline	2
SharingFactor	6
StreamingFactor	4

Alternatively, you can set `StreamingFactor` using the `hdlset_param` function, as in the following example.

```
dut = 'sfir_fixed/symmetric_fir';
```

```
hdlset_param(dut,'StreamingFactor', 4);
```

Requirements and Limitations for Streaming

This section describes the criteria for streaming that subsystems must meet.

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.

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} / `Oversampling`. Notice that the streamed sample time is independent of the original sample time, S.

Checks and 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.

If the requested streaming factor cannot be implemented, HDL Coder generates non-streaming code. It is good practice to generate an Optimization Report. The Streaming and Sharing page of the report provides information about:

- Success: Displays a link to highlight groups of blocks that belong to a streaming group in the Simulink model and in the generated model.
- Failure: Conditions that prevent streaming.

For more information on the streaming and sharing report, see "Optimization Report" on page 16-9.

More About

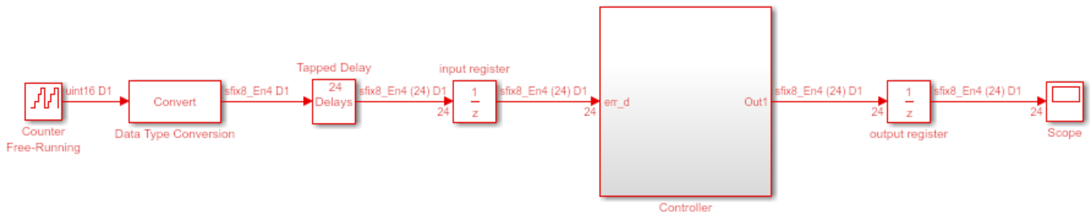
- "Resource Sharing" on page 15-20
- "Clock-Rate Pipelining" on page 15-66

Area Reduction with Streaming

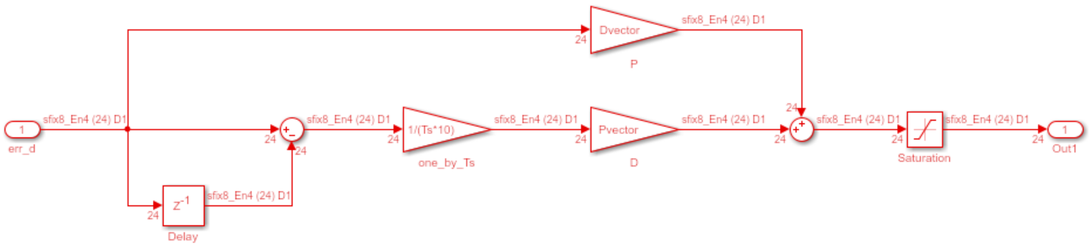
This example illustrates:

- Specification of a streaming factor for a subsystem
- Generation of HDL code and a validation model for the subsystem.

The following example is a single-rate model that drives the `Controller` subsystem with a vector signal of width 24.



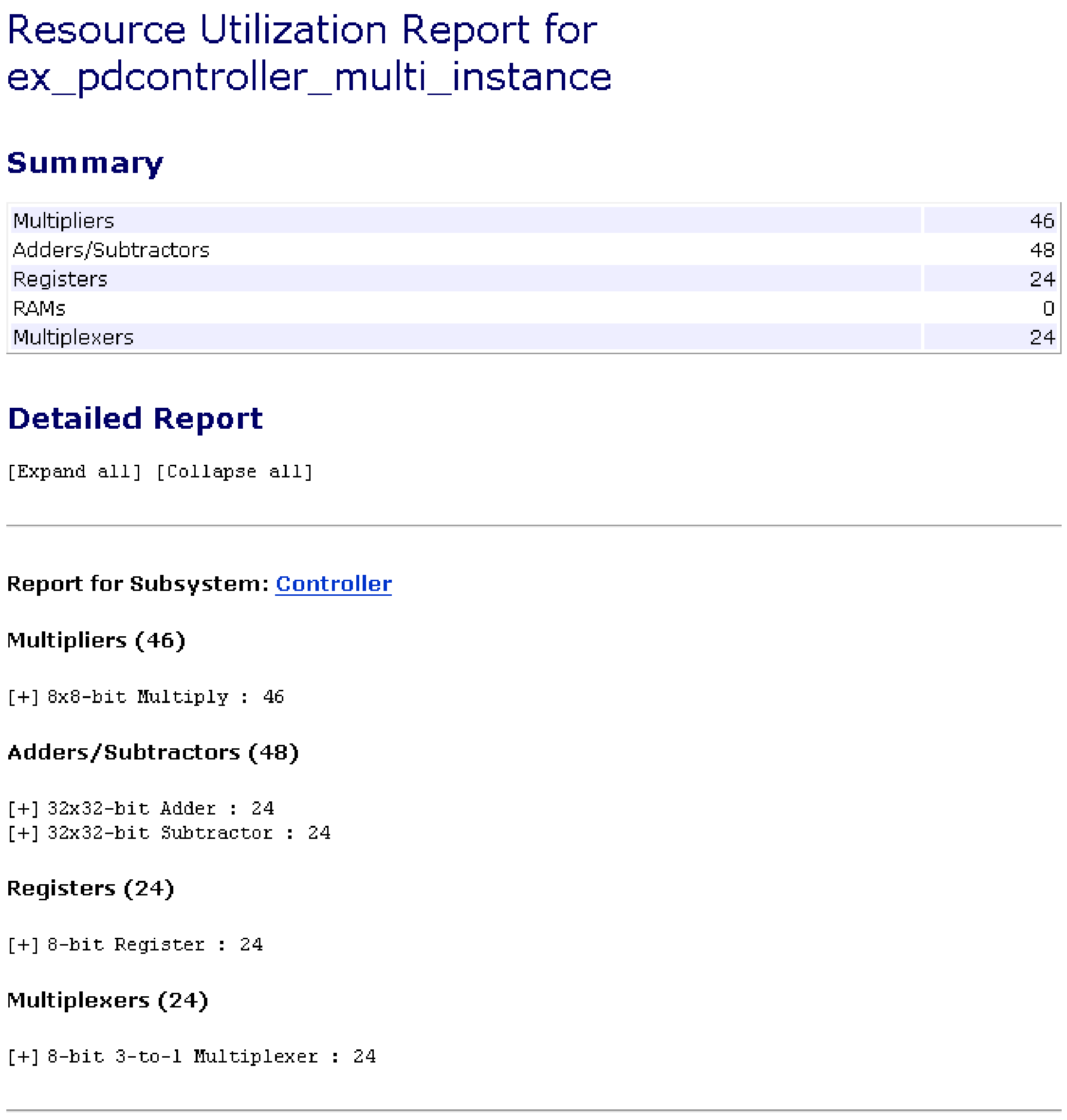
The following figure shows the `Controller` subsystem, which is the DUT in this example.



By generating HDL code and a report on resource utilization, you can determine how many multipliers, adders/subtractors, registers, RAMs, and multiplexers are generated from this DUT in the default case. To do so, type the following commands:

```
dut = 'ex_pdcontroller_multi_instance/Controller';
hdlset_param(dut,'StreamingFactor', 0);
makehdl(dut,'ResourceReport','on');
```

The following figure shows the Resource Utilization Report for the Controller subsystem. The report shows the number of multipliers, adders/subtractors, registers, RAMs, and multiplexers that the HDL Coder software generates.



Resource Utilization Report for ex_pdcontroller_multi_instance

Summary

Multipliers	46
Adders/Subtractors	48
Registers	24
RAMs	0
Multiplexers	24

Detailed Report

```
[Expand all] [Collapse all]
```

Report for Subsystem: Controller

Multipliers (46)

```
[+] 8x8-bit Multiply : 46
```

Adders/Subtractors (48)

```
[+] 32x32-bit Adder : 24
[+] 32x32-bit Subtractor : 24
```

Registers (24)

```
[+] 8-bit Register : 24
```

Multiplexers (24)

```
[+] 8-bit 3-to-1 Multiplexer : 24
```

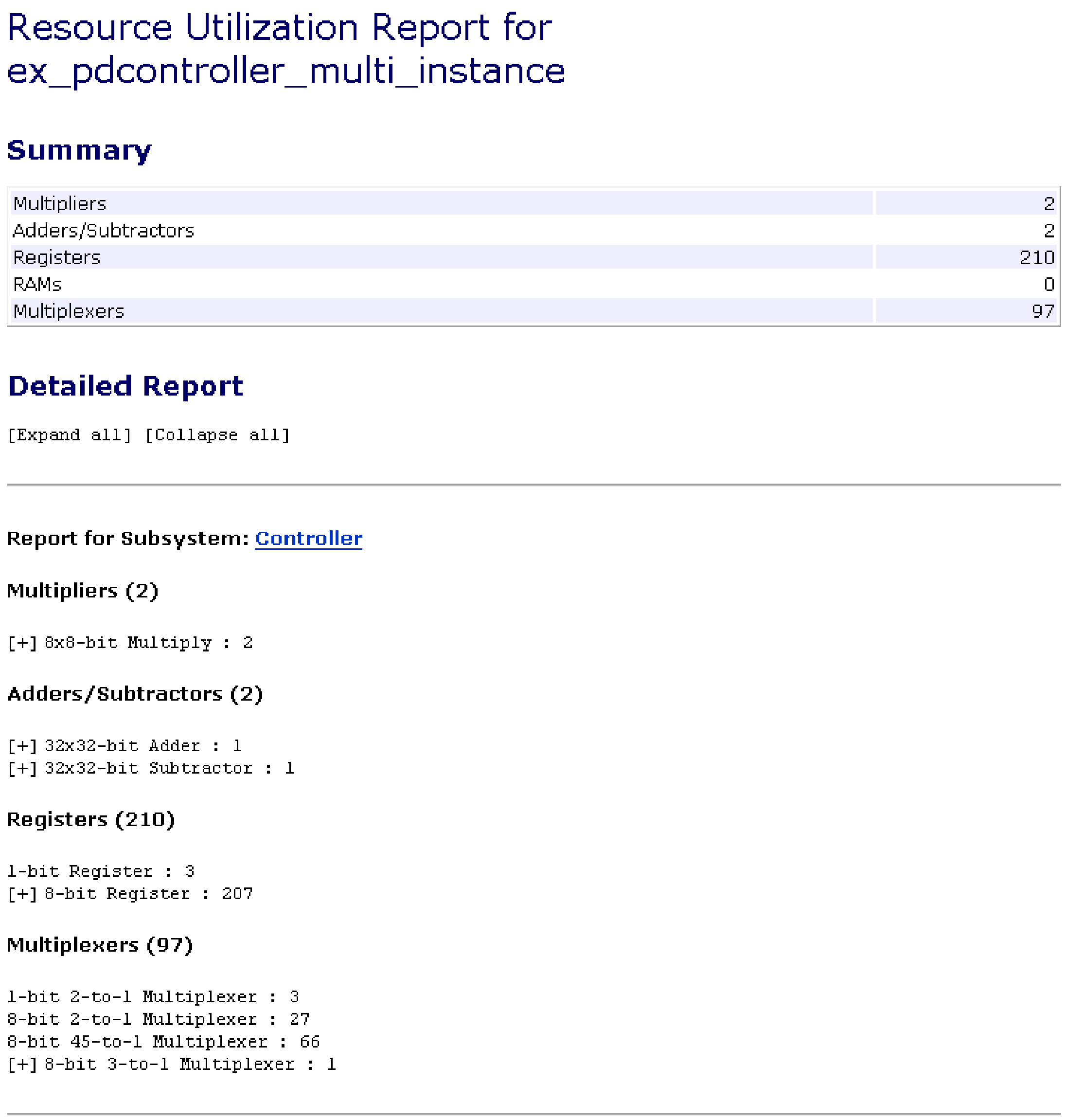
If you choose an optimal `StreamingFactor` for the DUT, you can achieve a drastic reduction in the number of multipliers and adders/subtractors generated. The following commands set `StreamingFactor` to the largest possible value for this subsystem and then generate VHDL code and a Resource Utilization Report.

```
dut = 'ex_pdcontroller_multi_instance/Controller';
hdlset_param(dut,'StreamingFactor', 24);
makehdl(dut,'ResourceReport','on', 'GenerateValidationModel','on');
```

During code generation, HDL Coder reports latency in the generated model. It also reports generation of the validation model.

```
### Starting HDL Check.
### HDL Check Complete with 0 errors, 0 warnings and 0 messages.

### The DUT requires an initial pipeline setup latency. Each output port experiences
    these additional delays
### Output port 0: 1 cycles


### Generating new validation model: gm_ex_pdcontroller_multi_instance4_vnl.mdl
### Validation Model Generation Complete.


### Begin VHDL Code Generation
### MESSAGE: The design requires 24 times faster clock with respect to the base rate = 2.
### Working on ex_pdcontroller_multi_instance/Controller/err_d_serializercomp
    as hdlsrc\err_d_serializercomp.vhd
### Working on ex_pdcontroller_multi_instance/Controller/Saturation_out1_serialcomp
    as hdlsrc\Saturation_out1_serialcomp.vhd
### Working on ex_pdcontroller_multi_instance/Controller/kconst_serializercomp
    as hdlsrc\kconst_serializercomp.vhd
### Working on ex_pdcontroller_multi_instance/Controller/kconst_serializercomp1
    as hdlsrc\kconst_serializercomp1.vhd
### Working on Controller_tc as hdlsrc\Controller_tc.vhd
### Working on ex_pdcontroller_multi_instance/Controller as hdlsrc\Controller.vhd
### Generating package file hdlsrc\Controller_pkg.vhd
### Generating HTML files for code generation report in
    C:\hdlsrc\html\ex_pdcontroller_multi_instance directory ...

### HDL Code Generation Complete.
```

After code generation completes, you can view the results of the `StreamingFactor` optimization. In the Resource Utilization Report, you can see that HDL Coder generates only two multipliers and two adders for the Controller subsystem.



Resource Utilization Report for ex_pdcontroller_multi_instance

Summary

Multipliers	2
Adders/Subtractors	2
Registers	210
RAMs	0
Multiplexers	97

Detailed Report

[Expand all] [Collapse all]

Report for Subsystem: Controller

Multipliers (2)

```
[+] 8x8-bit Multiply : 2
```

Adders/Subtractors (2)

```
[+] 32x32-bit Adder : 1
[+] 32x32-bit Subtractor : 1
```

Registers (210)

```
1-bit Register : 3
[+] 8-bit Register : 207
```

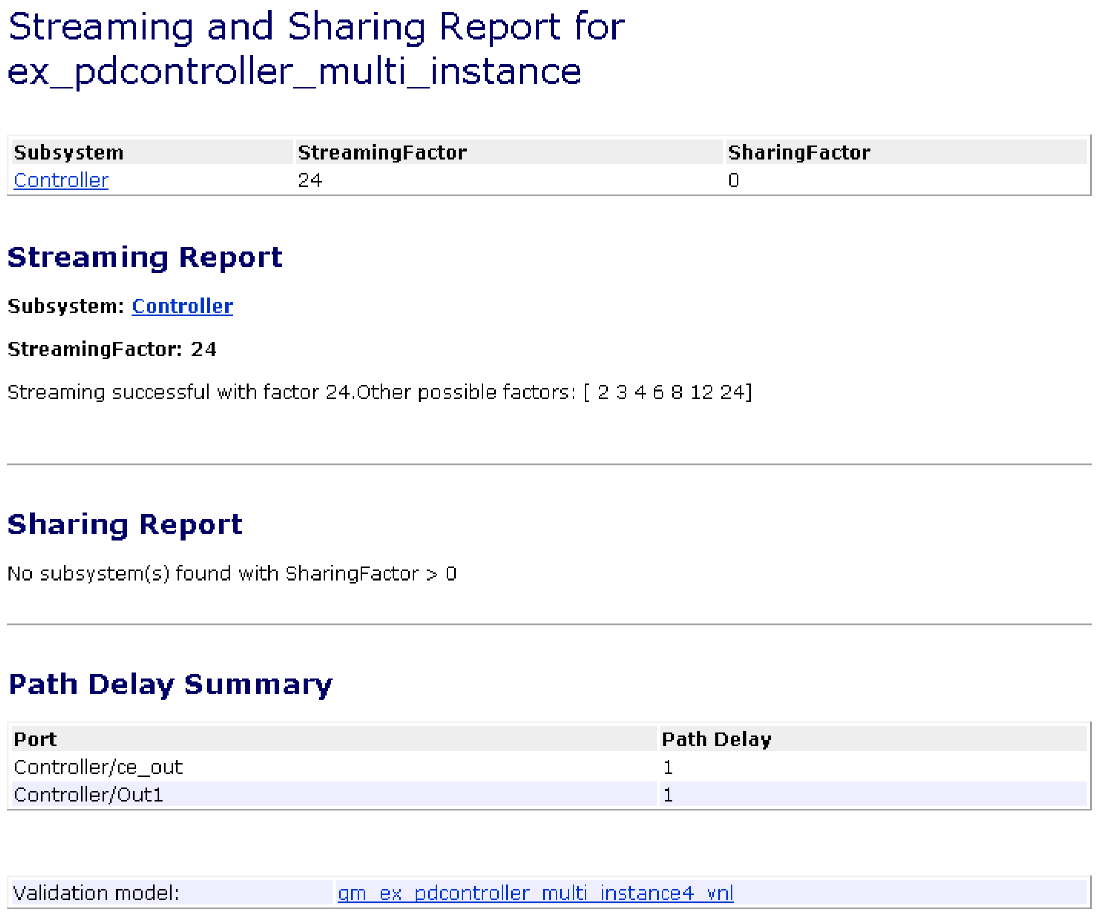
Multiplexers (97)

```
1-bit 2-to-1 Multiplexer : 3
8-bit 2-to-1 Multiplexer : 27
8-bit 45-to-1 Multiplexer : 66
[+] 8-bit 3-to-1 Multiplexer : 1
```

HDL Coder also produces a Streaming and Sharing report that shows:

- The `StreamingFactor` value that you specified

- The other usable `StreamingFactor` values for this subsystem
- Latency (delays) introduced in the generated model
- A hyperlink to the validation model, if generated



Streaming and Sharing Report for ex_pdcontroller_multi_instance

Subsystem	StreamingFactor	SharingFactor
Controller	24	0

Streaming Report

Subsystem: Controller

StreamingFactor: 24

Streaming successful with factor 24.Other possible factors: [2 3 4 6 8 12 24]

Sharing Report

No subsystem(s) found with SharingFactor > 0

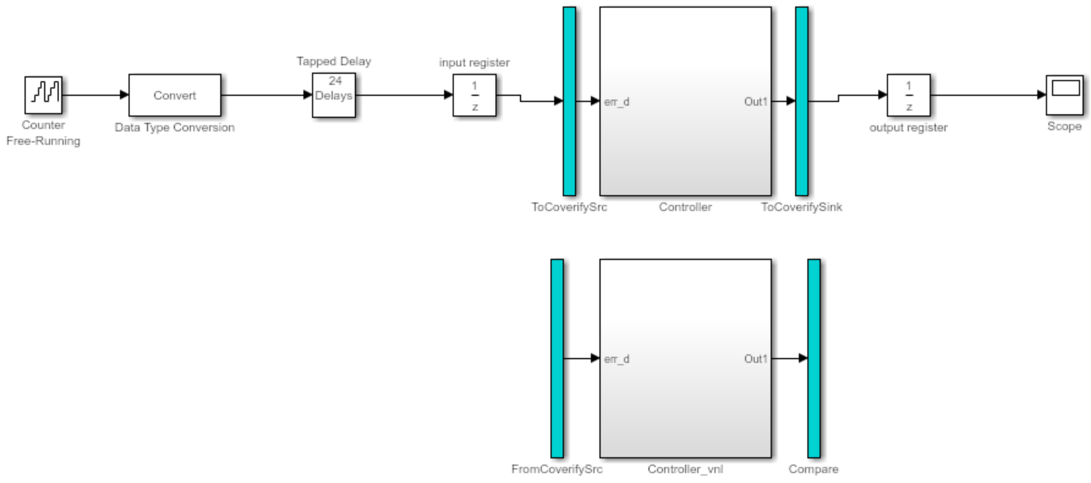
Path Delay Summary

Port	Path Delay
Controller/ce_out	1
Controller/Out1	1

Validation model:	gm_ex_pdcontroller_multi_instance4_vnl

The Validation Model

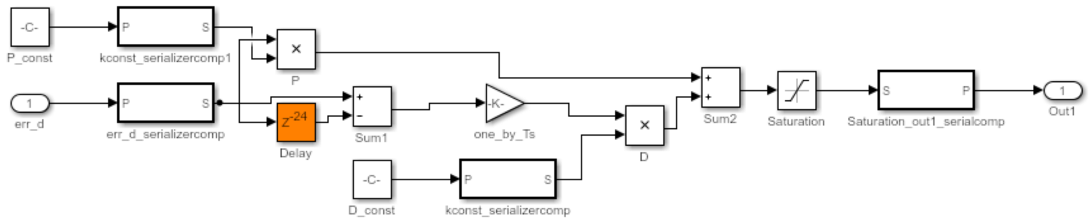
The following figure shows the validation model generated for the Controller subsystem.



The lower section of the validation model contains a copy of the original DUT (Controller_vnl). This single-rate subsystem runs at its original rate.

The upper section of the validation model contains the streaming version of the DUT (Controller). Internally, this subsystem runs at a different rate than the original DUT.

The following figure shows the interior of the Controller subsystem.

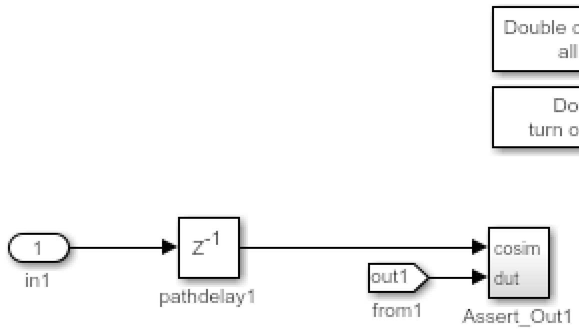


Inspection of the Controller subsystem shows that it is a multirate subsystem, having two rates that operate as follows:

- Inputs and outputs run at the same rate as the exterior model.
- Dual-rate Serializer blocks receive vector data at the original rate and output a stream of scalar values at the higher (24x) rate.

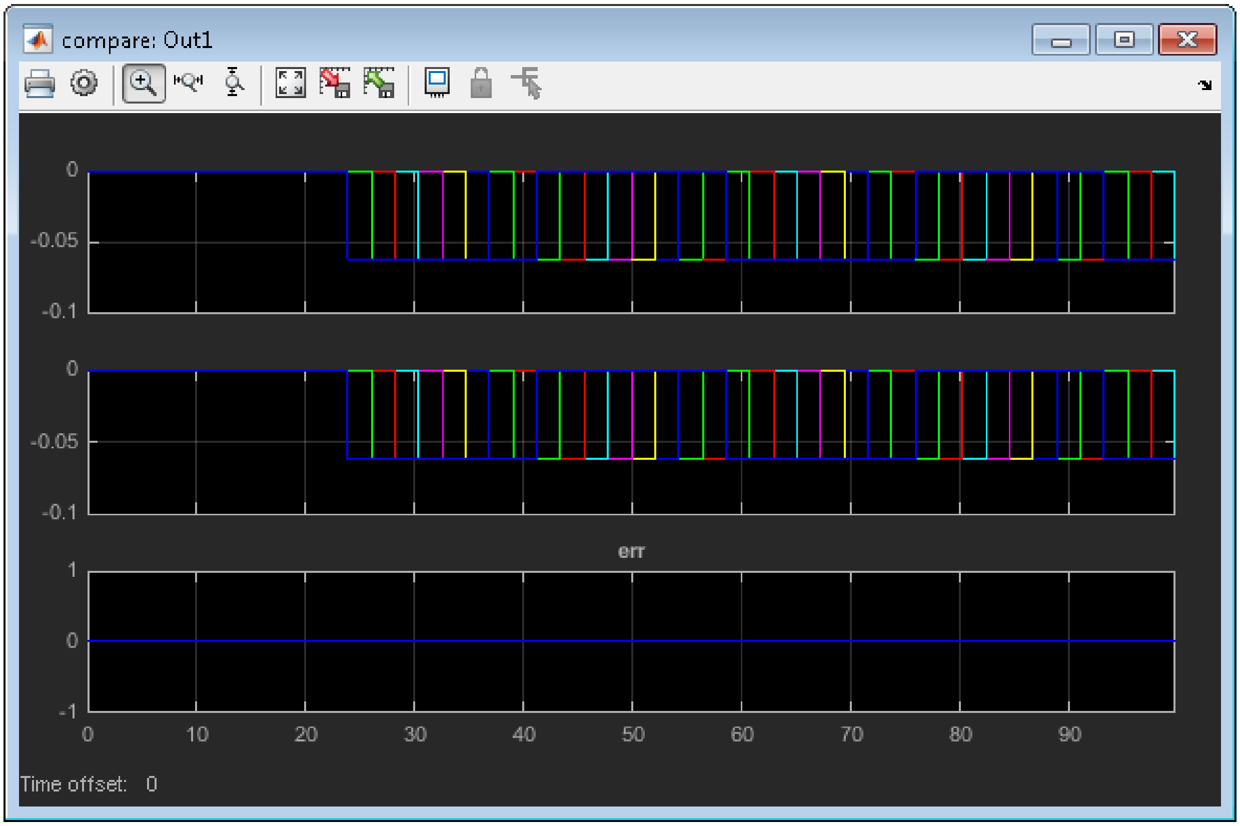
- Interior blocks between Serializers and Deserializer run at the higher rate.
- The Deserializer block receives scalar values at the higher rate and buffers values into a 24-element output vector running at the original rate.

The Compare subsystem (see following figure) receives and compares outputs from the Controller and Controller_vnl subsystems. To compensate for the latency of the Controller subsystem (reported during code generation), input from the Controller_vnl subsystem is delayed by one clock cycle. A discrepancy between the outputs of the two subsystems triggers an assertion.


Double click to turn OFF all Assertions
Double click to turn on/off all scopes
1
in1
z-1
pathdelay1
out1
from1
cosim
dut
Assert_Out1

To verify that a generated model with streaming is bit-true to its original counterpart in a validation model:

1. Open the Compare subsystem.
2. Double click the **Double click to turn on/off all scopes** button.
3. Run the validation model.
4. Observe the **compare:Out1** scope. The error signal display shows a line through zero, indicating that the data comparisons were equal.


compare: Out1
0
-0.05
-0.1
0
-0.05
-0.1
err
1
0
-1
0
10
20
30
40
50
60
70
80
90
Time offset: 0

Resource Sharing

In this section...

What Is Resource Sharing?

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.

You can specify a *sharing factor* N for a subsystem or a MATLAB Function block. HDL Coder tries to identify up to N shareable resources, and, by default, oversamples by a factor of N to generate an area-optimized implementation with the original latency. If the coder cannot identify N shareable resources, it shares as many as it can, but still oversamples by a factor of N.

You can limit the oversampling ratio to meet target hardware clock constraints. For details, see "Optimization with Constrained Overclocking" on page 15-5.

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 14-2.

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

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.

Specify Resource Sharing

Specify Resource Sharing from the UI

To specify resource sharing from the UI:

1. Right-click the subsystem, model reference, or MATLAB Function block.
2. Select **HDL Code** > **HDL Block Properties**.
3. In the **SharingFactor** field, enter the number of shareable resources.

Specify Resource Sharing from the Command Line

Set the `SharingFactor` using `hdlset_param`, as in the following example.

```
dut = 'ex_dimcheck/Channel';
hdlset_param(dut,'SharingFactor',3);
```

Requirements and Limitations for Resource Sharing

HDL Coder does not support model references for resource sharing.

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
```

Blocks to be shared have the following requirements:

- Single-rate.
- No bus inputs or outputs.
- If the block is within a feedback loop, at least one Unit Delay or Delay block connected to each output port.
- If you set the maximum oversampling ratio to 1, shared resources cannot be inside feedback loops.
- If you set the maximum oversampling ratio to 1, shared resources cannot have latency.

If you want to share Atomic Subsystem blocks:

- The only state elements that these blocks can contain are:

  - Delay
  - Unit Delay
  - Unit Delay Enabled
  - Unit Delay Resettable
  - Unit Delay Enabled Resettable

  These blocks must have the **Initial condition** parameter set to 0.
- These blocks must not contain a subsystem that does not meet the requirements for resource sharing.

If you want to share MATLAB Function blocks, they must not use:

- Persistent variables
- Loop streaming
- Output pipelining

For resource sharing, the Product blocks and Gain blocks can have different word-lengths. To share the blocks, specify the multiplier promotion threshold and the sharing factor. The multiplier promotion threshold is the maximum word-length by which HDL Coder promotes a multiplier for sharing with other multipliers.

To specify the multiplier promotion threshold, in the Configuration Parameters dialog box, on the **HDL Code Generation** > **Target and Optimizations** > **Resource sharing** tab, for **Multiplier promotion threshold**, enter an integer. For more information, see `MultiplierPromotionThreshold`.

HDL Coder can share Atomic Subsystem and MATLAB Function blocks that have:

- The same Simulink checksum. Use `Simulink.Subsystem.getChecksum` to determine the checksum.
- The same HDL block properties.

In addition, if you use the `DSPStyle` block property, HDL Coder does not share multipliers that have different synthesis attribute settings.

Limitations for Atomic Subsystem Sharing

You cannot apply resource sharing to a subset of instances for a particular atomic subsystem; you must share all instances. If you want to share a subset of atomic subsystem instances, change the remaining instances to virtual subsystems.

The only blocks with state elements that can be part of a shared atomic subsystem are:

- Delay
- Unit Delay
- Unit Delay Enabled
- Unit Delay Resettable
- Unit Delay Enabled Resettable

Therefore, 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, except Discrete FIR Filter
- Communications System 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 15-28.

Resource Sharing Information in Reports

If you generate a code generation report for each subsystem or MATLAB Function block that implements sharing, the report includes the following information:

- Success: Lists resource usage changes caused by sharing and displays a link to highlight groups of blocks in the Simulink model that shared resources.
- Failure: Identifies which criterion was violated and displays the link to any offending blocks in the Simulink model that caused resource sharing to fail.
- Latency changes.

For more information on the sharing report, see streaming and sharing report in "Optimization Report" on page 16-9

Related Examples

- Resource Sharing For Area Optimization
- Resource Sharing and Streaming with Oversampling Constraints

More About

- "Clock-Rate Pipelining" on page 15-66
- "Streaming" on page 15-7

Check Compatibility for Resource Sharing

To determine whether your model is compatible for resource sharing:

1. Before generating code, run `checkhdl` and eliminate general compatibility issues.
2. In the Configuration Parameters dialog box, on the **HDL Code Generation** pane, select **Generate optimization report**.
3. Set the sharing factor for the DUT and generate code.
4. After code generation completes, inspect the Optimization Report. The report shows incompatible blocks or other conditions that can cause a resource sharing request to fail.
5. If the Optimization Report shows problems, fix them and repeat these steps.

See also “Requirements and Limitations for Resource Sharing” on page 15-21.

Delay Balancing

In this section...

Why Use Delay Balancing

The HDL Coder software 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 inherently introduce delays in the generated code. "Resolve Numerical Mismatch with Delay Balancing" on page 15-30 discusses one such implementation.

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*. When you enable delay balancing and the coder detects introduction of new delays along one path, the coder inserts matching delays on the other paths. When delay balancing is enabled, the generated model is functionally equivalent to the original model.

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 `BalanceDelays`.
- `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 `GenerateValidationModel`.

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');
```

For more information about the validation model, see "Validation Model" on page 14-3.

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.

To learn how to set delay balancing for a subsystem, see "Set Delay Balancing For a Subsystem" on page 12-6.

Delay Balancing Limitations

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

The following block implementations do not support delay balancing:

- hdldefaults.ConstantSpecialHDLEmission
- hdldefaults.NoHDL

HDL Coder does not support delay balancing, if:

- There are multiple instances of an Atomic Subsystem in different conditional subsystems.
- The `BalanceDelays` 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.
- 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.

- The sample time is not discrete, or the ratio of sample times of the fastest to the slowest rate is too large.
- 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.

If delay balancing is unsuccessful, `hdlcoder.optimizeDesign` cannot optimize the generated HDL code.

Delay Balancing Information in Reports

To see the delay balancing information, select the **Delay Balancing** section of the optimization report when you generate the optimization report for each subsystem, model reference, or MATLAB Function block.

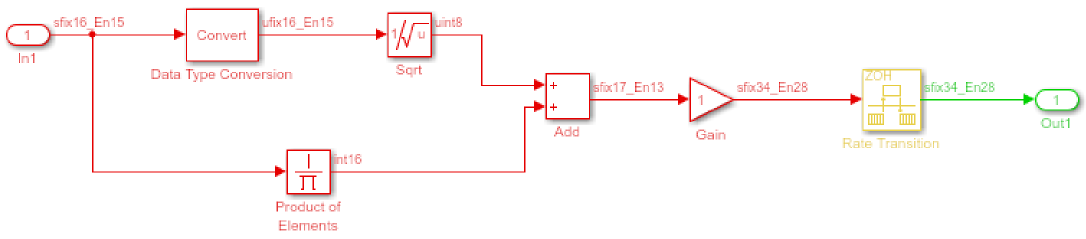
- Success: Lists the pipeline latency and phase delay at the output ports that shows the number of pipelines added at the output ports to match the delays.
- Failure: Identifies which criterion was violated and displays the link to the block or subsystem in the Simulink model that caused delay balancing to fail.
- Latency changes.

See also “Optimization Report” on page 16-9.

Resolve Numerical Mismatch with Delay Balancing

This example shows a simple case where the VHDL implementation of a block introduces delays that cause a numeric mismatch between the original DUT and the generated model and HDL code. The example then demonstrates how to use delay balancing to fix the mismatch.

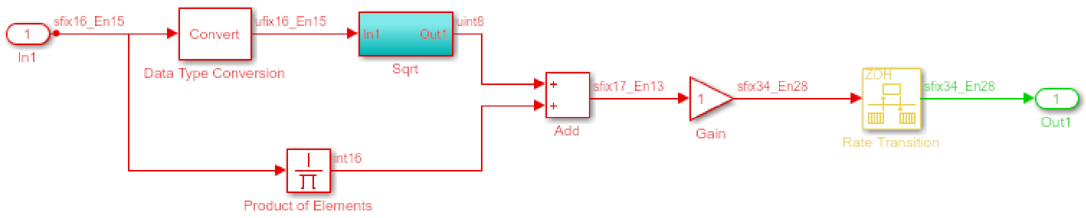
The following figure shows the DUT for the `ex_rsqrt_delaybalancing` model. The DUT is a simple multirate subsystem that includes a Reciprocal Square Root block, `Sqrt`. A Rate Transition block downsamples the output signal to a lower sample rate.



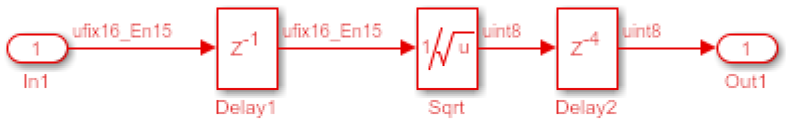
Generate HDL code without delay balancing and generate a validation model:

```
dut = 'ex_rsqrt_delaybalancing/Subsystem';
makehdl(dut,'BalanceDelays','off','GenerateValidationModel','on');
```

Examination of the generated model shows that HDL Coder has implemented the `Sqrt` block as a subsystem:



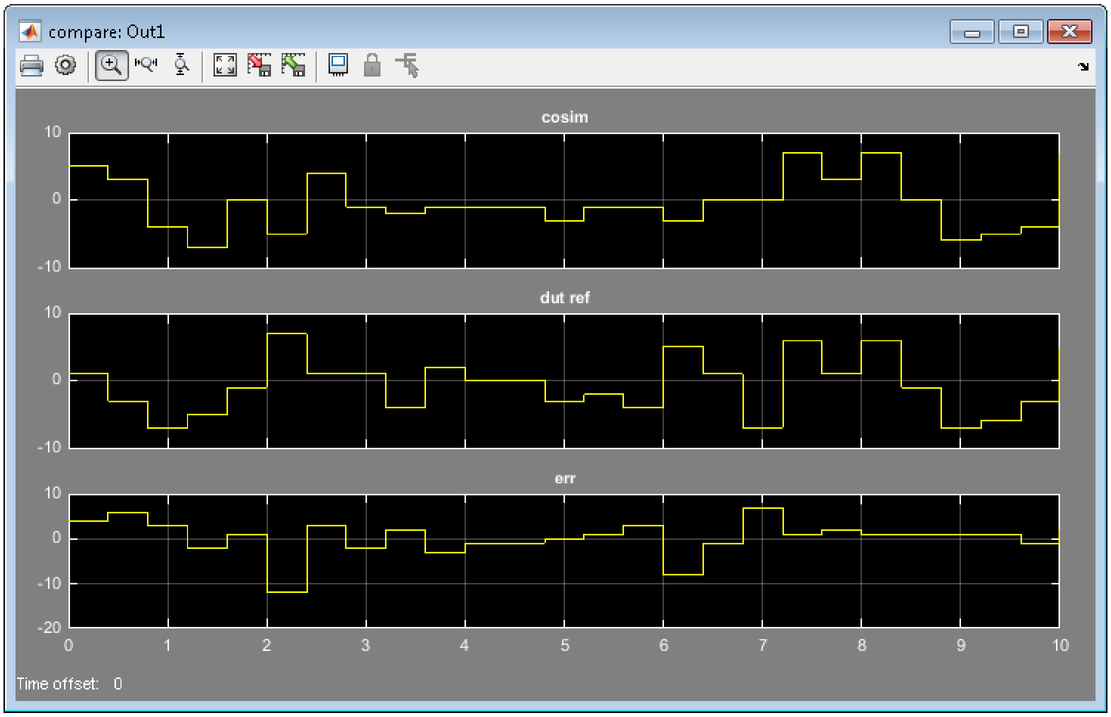
The following figure shows that the generated `Sqrt` subsystem introduces a total of 5 cycles of delay. (This behavior is inherent to the Reciprocal Square Root block implementation.) These delays map to registers in the generated HDL code when **UseRAM** is `off`.



The scope in the following figure shows the results of a comparison run between the original and generated models. The scope displays the following signals, in descending order:

- The outputs from the original model
- The outputs from the generated model
- The difference between the two

The difference is nonzero, indicating a numerical mismatch between the original and generated models.



Two factors cause this discrepancy:

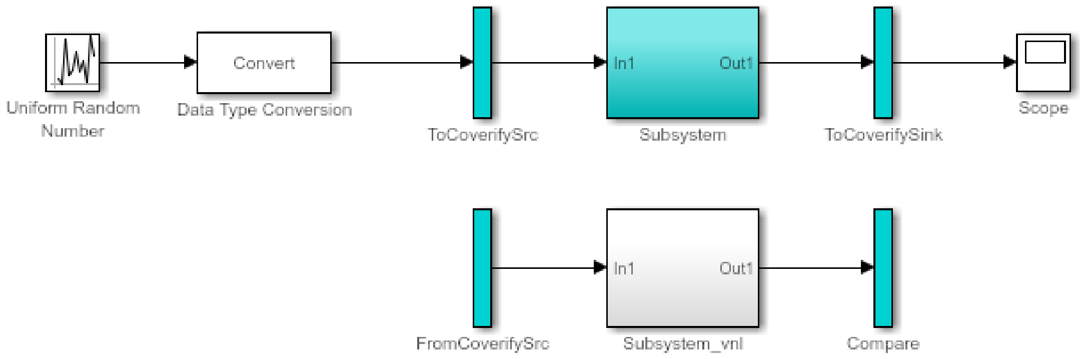
- The input signal branches into two parallel paths (to the `Sqrt` and product blocks) but only the branch to the `Sqrt` block introduces delays.
- The downsampling caused by the rate transition drops samples.

You can solve these problems by manually inserting delays in the generated model. However, using delay balancing produces more consistent results.

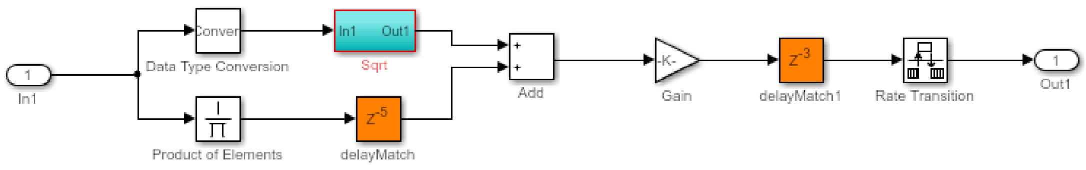
Generate HDL code with delay balancing and generate a validation model:

```
dut = 'ex_rsqrt_delaybalancing/Subsystem';
makehdl(dut,'BalanceDelays','on','GenerateValidationModel','on');
```

The following figure shows the validation model. The lower subsystem is identical to the original DUT. The upper subsystem represents the HDL implementation of the DUT.



The upper subsystem (shown in the following figure) represents the HDL implementation of the DUT. To balance the 5-cycle delay from the `Sqrt` subsystem, HDL Coder has inserted a 5-cycle delay on the parallel data path. The coder has also inserted a 3-cycle delay before the Rate Transition to offset the effect of downsampling.



Find Feedback Loops

In this section...

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

Using the HDL Workflow Advisor

In the **HDL Code Generation** > **Set Code Generation Options** > **Set Advanced Options** > **Diagnostics** tab, select **Highlight feedback loops inhibiting delay balancing and optimizations**.

To customize the script name, enter a file name in the **Feedback loop highlighting script file name** field.

Using the Configuration Parameters Dialog Box

In the **HDL Code Generation** > **Global Settings** > **Diagnostics** tab, select **Highlight feedback loops inhibiting delay balancing and optimizations**.

To customize the script name, enter a file name in the **Feedback loop highlighting script file name** field.

Using the Command Line

To generate a feedback loop highlighting script programmatically, use the `HighlightFeedbackLoops` and `HighlightFeedbackLoopsFile` properties with `makehdl` or `hdlset_param`.

For example:

- To generate a feedback loop highlight script for a model, *`myModel`*, enter:

  ```
  hdlset_param ('myModel', 'HighlightFeedbackLoops', 'on');
  ```
- To generate a feedback loop highlighting script with the file name, *`myHighlightScript`*, for a model, *`myModel`*, enter:

  ```
  hdlset_param ('myModel', 'HighlightFeedbackLoops', 'on');
  hdlset_param ('myModel', 'HighlightFeedbackLoopsFile', 'myHighlightScript');
  ```

Remove Highlighting

To turn off highlighting, in Simulink, select **Display** > **Remove Highlighting**.

Limitations

Feedback loop highlighting cannot highlight blocks that have names that contain a single quote (`'`).

See Also

`HighlightFeedbackLoops` | `HighlightFeedbackLoopsFile`

More About

- "Optimization Report" on page 16-9

Hierarchy Flattening

In this section...

What Is Hierarchy Flattening?

Hierarchy flattening enables you to remove subsystem hierarchy from the HDL code generated from your design.

The HDL Coder software 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

Flatten hierarchy to:

- Enable more extensive area and speed optimization.
- Reduce the number of HDL output files. For every subsystem flattened, HDL Coder generates one less HDL output file.

Avoid flattening hierarchy if you want to preserve one-to-one mapping from subsystem name to HDL `module` or `entity` name. Not flattening hierarchy makes the HDL code more readable.

Prerequisites for Hierarchy Flattening

To flatten hierarchy, a subsystem must have the following block properties.

Property	Required value
`DistributedPipelining`	`'off'`
`ClockRatePipelining`	`'off'`
`StreamingFactor`	0
`SharingFactor`	0

To flatten hierarchy, you must also have the `MaskParameterAsGeneric` global property set to `'off'`. For more information, see MaskParameterAsGeneric.

Options for Hierarchy Flattening

By default, a subsystem inherits its hierarchy flattening setting from the parent subsystem. However, you can enable or disable flattening for individual subsystems.

The 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.

How to Flatten Hierarchy

To set hierarchy flattening using the HDL Block Properties dialog box:

1. Right-click the subsystem.
2. Select **HDL Code** > **HDL Block Properties** .
3. 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 black box implementation or model reference.
- A triggered subsystem when `TriggerAsClock` is enabled.
- A masked subsystem that contains any of the following:
  - Bus.
  - Enumerated data type.
  - Lookup table block: 1-D Lookup Table, 2-D Lookup Table, Cosine, Direct LookupTable (n-D), Prelookup, Sine, n-D Lookup Table.
  - MATLAB Function block.
  - MATLAB System block.
  - Stateflow block: Chart, State Transition Table, Message Viewer.
  - Block with a pass-through or no-op implementation. See “Pass through, No HDL, and Cascade Implementations” on page 12-61.

Note: This option removes subsystem boundaries before code generation. It does not necessarily generate HDL code with a completely flat hierarchy.

RAM Mapping

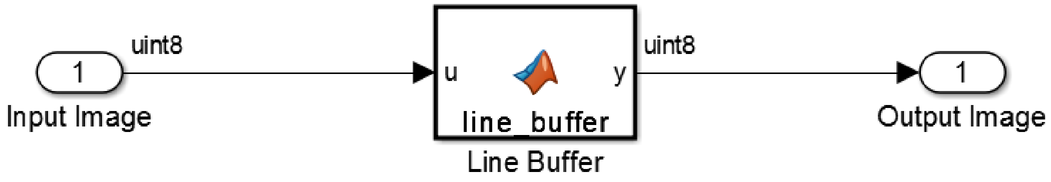
RAM mapping is an area optimization. You can map to RAMs in HDL code by using:

- `UseRAM` to map delays to RAM. For details, see “UseRAM” on page 12-30.
- `MapPersistentVarsToRAM` to map persistent arrays in a MATLAB Function block to RAM. For details, see “MapPersistentVarsToRAM” on page 12-22.
- RAM blocks from the HDL Operations library:
  - Dual Port RAM
  - Dual Rate Dual Port RAM
  - Simple Dual Port RAM
  - Single Port RAM
- Blocks with a RAM implementation.

RAM Mapping with the MATLAB Function Block

This example shows how to map persistent arrays to RAM 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.

1 In the Simulink editor, create a model, and open the Simulink Library Browser.
2 Add an Inport block, a MATLAB Function block, and an Outport block to your model, and name them as shown in figure.



3 Double-click the `Line Buffer` MATLAB Function block. In the MATLAB editor, copy this MATLAB code for a `line_buffer` function.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Line buffer: Uses a presistent array to store the image
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function y = line_buffer(u)
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
```

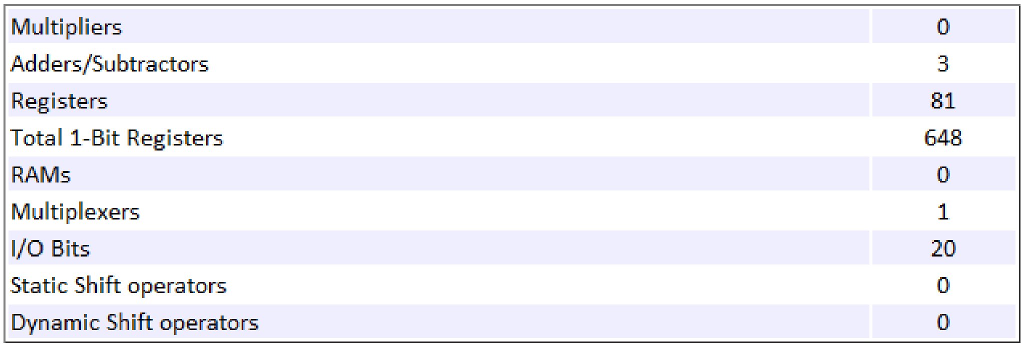
4 To use this model as your design under test (DUT) for generating HDL code, select all blocks and lines, and create a subsystem. Save your model as

`RAM_Mapping_Using_MATLAB_Function`. By default, RAM mapping is disabled, as **MapPersistentVarsToRAM** is set to **off**.

5 In the **Simulation** > **Model Configuration Parameters** > **HDL Code Generation** pane, enable **Generate resource utilization report** and click **Apply**.

6 Click **Generate** to generate HDL code.

7 In the Code Generation Report, select **High-level Resource Report**.



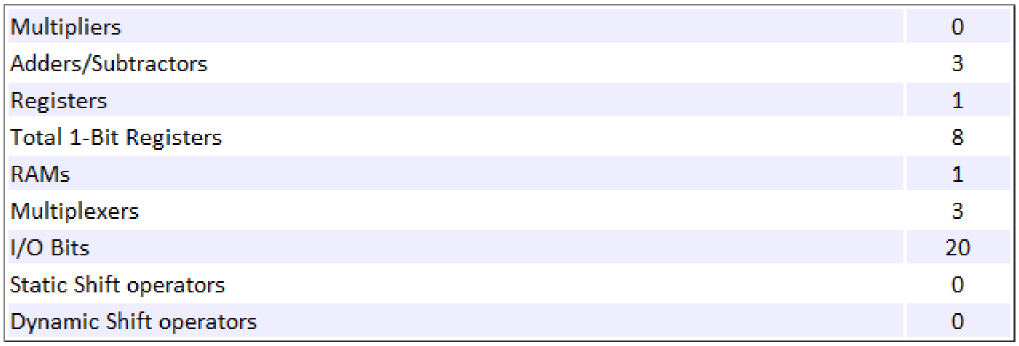
Multipliers	0
Adders/Subtractors	3
Registers	81
Total 1-Bit Registers	648
RAMs	0
Multiplexers	1
I/O Bits	20
Static Shift operators	0
Dynamic Shift operators	0

The design uses 81 registers, 648 1–bit Registers, and no RAM.

8 To enable RAM mapping, right-click the `Line Buffer` block, select **HDL Code** > **HDL Block Properties**, and set **MapPersistentVarsToRAM** to **on**. Click **OK**.

9 In the **Simulation** > **Model Configuration Parameters** > **HDL Code Generation** pane, click **Generate** to generate HDL code.

10 In the Code Generation Report, select **High-level Resource Report**.



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

The design now uses one register, eight 1–bit registers, and one RAM.

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

`RAMMappingThreshold`

More About

- “MATLAB Function Block Design Patterns for HDL” on page 20-24
- “MapPersistentVarsToRAM” on page 12-22

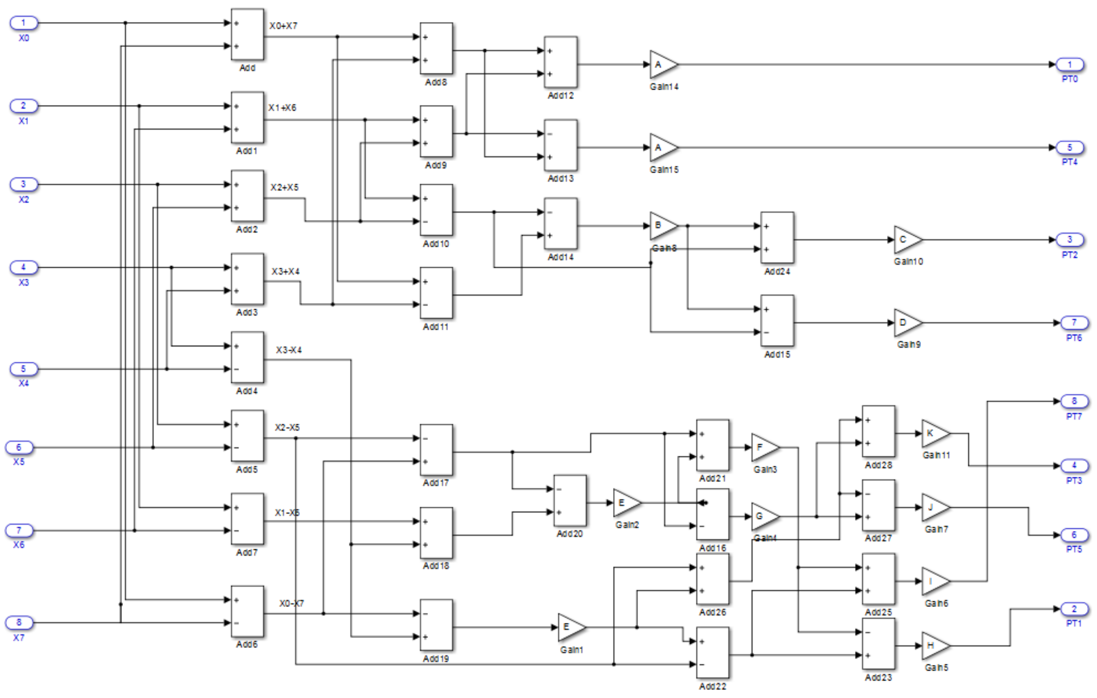
Insert Distributed Pipeline Registers in a Subsystem

This example shows how to use distributed pipelining with the `dct8_fixed` model.

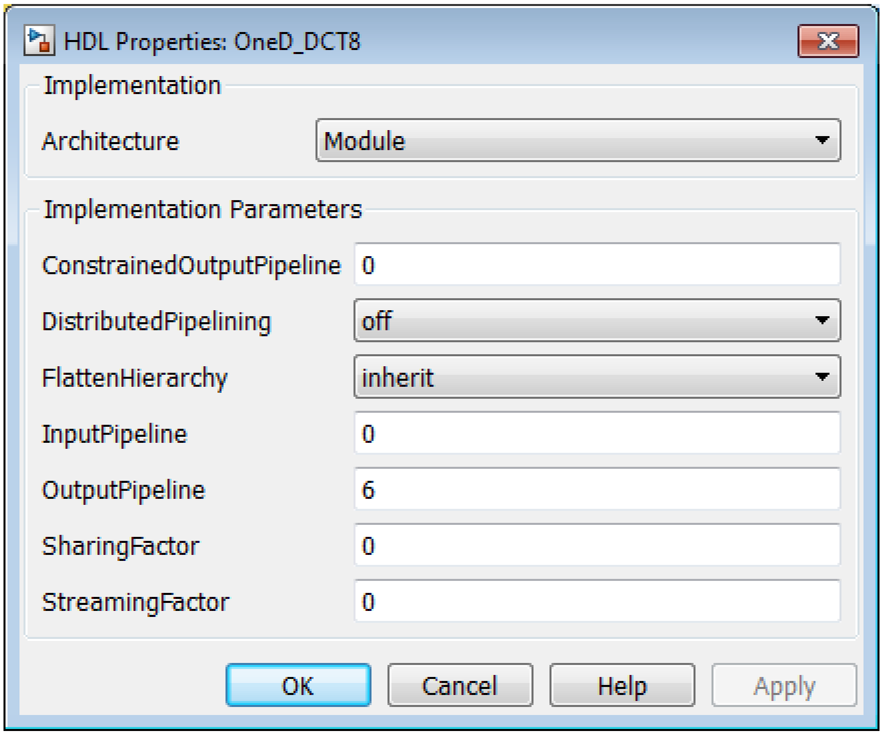
This example uses the following optimizations:

- Output pipelining
- Distributed pipelining

Open the model by typing `dct8_fixed` at the MATLAB prompt. The DUT is the `dct8_fixed/OneD_DCT8` subsystem.



Set **DistributedPipelining** to `off` and **OutputPipeline** to `6` to insert six pipeline stages at the outputs of the DUT.


HDL Properties: OneD_DCT8

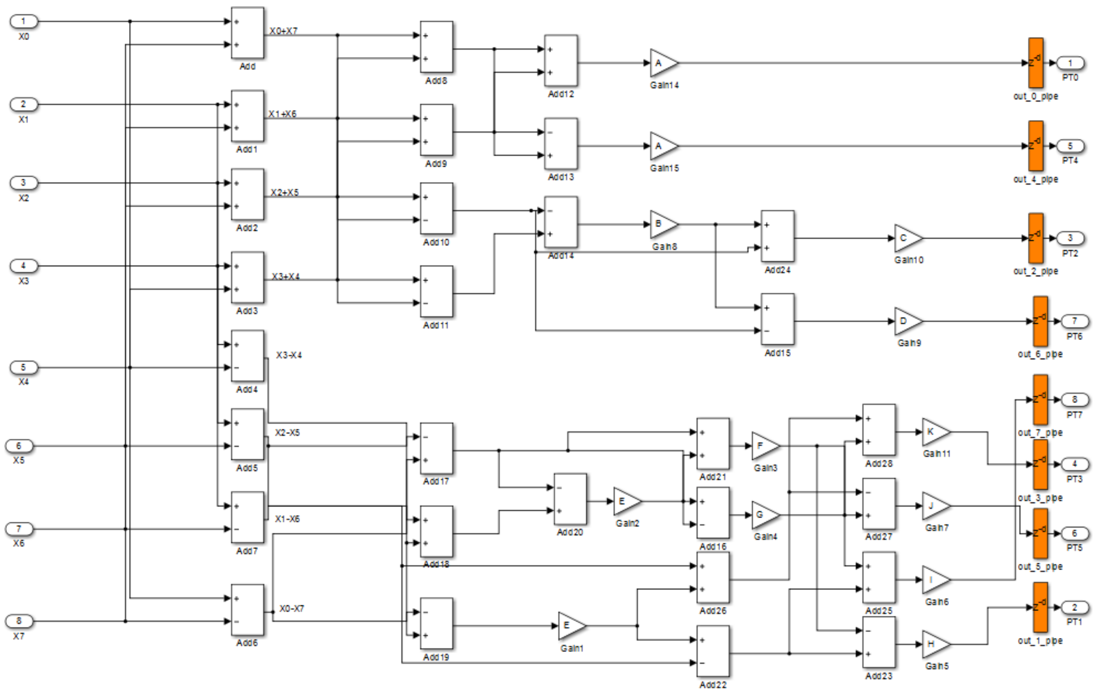
Implementation

Architecture: Module

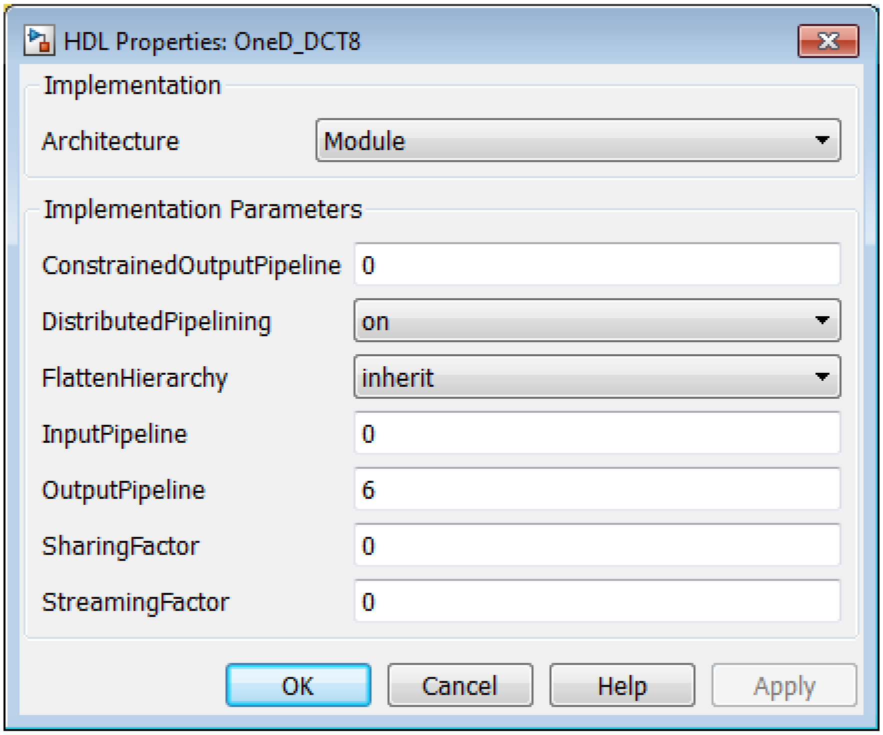
Implementation Parameters

ConstrainedOutputPipeline	0
DistributedPipelining	off
FlattenHierarchy	inherit
InputPipeline	0
OutputPipeline	6
SharingFactor	0
StreamingFactor	0

The generated model shows the placement of pipeline registers as highlighted delays at the outputs of the DUT. For more information about generated models, see “Generated Model and Validation Model” on page 14-2.

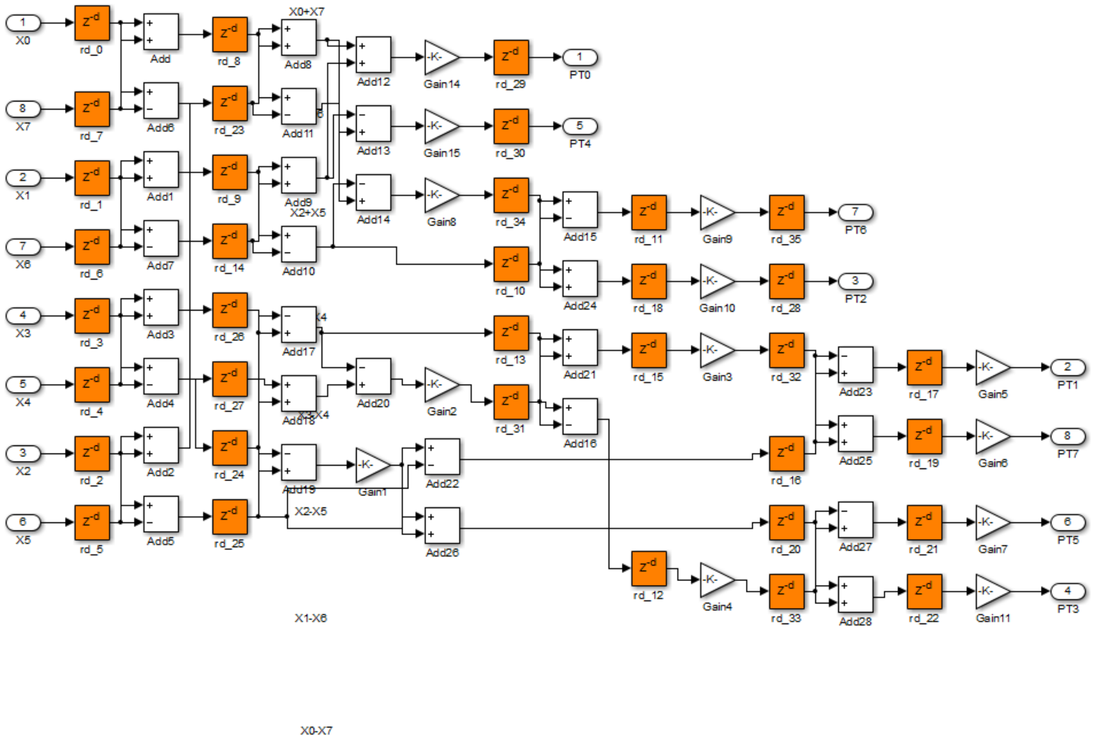


Set **DistributedPipelining** to `on` and **OutputPipeline** to `6` to distribute six pipeline stages for each signal path in the DUT.



Architecture	Module
ConstrainedOutputPipeline	0
DistributedPipelining	on
FlattenHierarchy	inherit
InputPipeline	0
OutputPipeline	6
SharingFactor	0
StreamingFactor	0

The generated model shows the distribution of pipeline registers as highlighted delays within each signal path. There are six pipeline registers for each path.



See Also

`DistributedPipeliningBarriers` | `DistributedPipeliningPriority` | `HierarchicalDistPipelining`

More About

- "Distributed Pipelining and Hierarchical Distributed Pipelining" on page 15-47

Distributed Pipelining and Hierarchical Distributed Pipelining

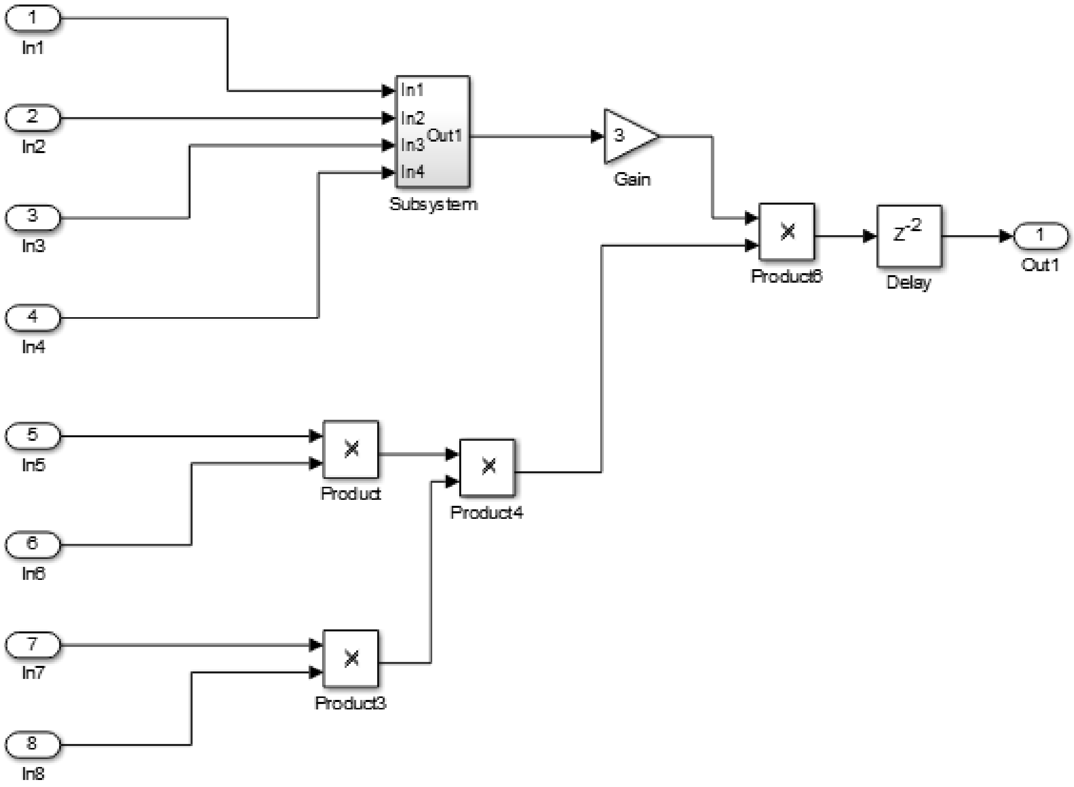
In this section...

What Is Distributed Pipelining?

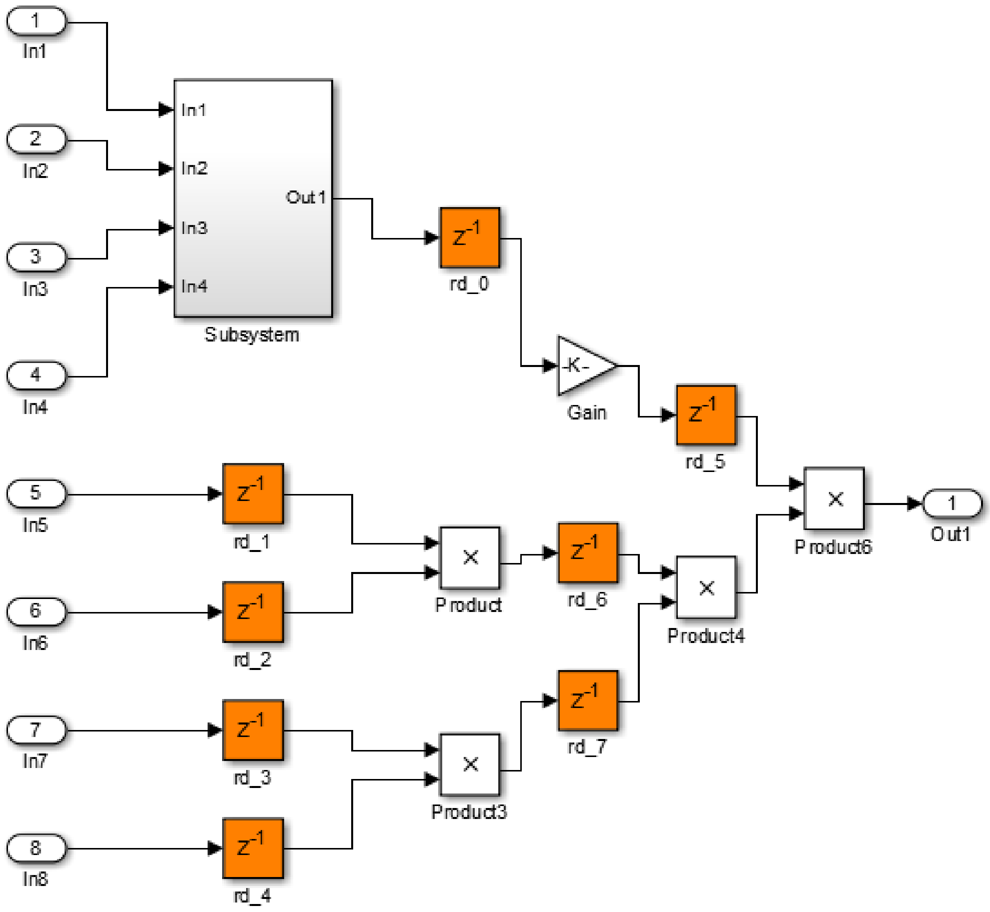
Distributed pipelining, or register retiming, is a speed optimization that moves existing delays in a design to reduce the critical path while preserving functional behavior.

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.
- MATLAB Function block within a subsystem. For details, see “Distributed Pipeline Insertion for MATLAB Function Blocks” on page 20-41.
- Stateflow chart within a subsystem.

To specify distributed pipelining using the UI:

1. Right-click the block and select **HDL Code** > **HDL Block Properties**.
2. Set **DistributedPipelining** to **on** and click **OK**.

To enable distributed pipelining, on the command line, enter:

```
hdlset_param('path/to/block', 'DistributedPipelining', 'on')
```

To disable distributed pipelining, on the command line, enter:

```
hdlset_param('path/to/block', 'DistributedPipelining', 'off')
```

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 11-147
- `IgnoreDataChecking`

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:
  - A MATLAB Function block or Stateflow chart contains a matrix with a statically unresolvable index.
  - A Stateflow chart contains a state or local variable.
- 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

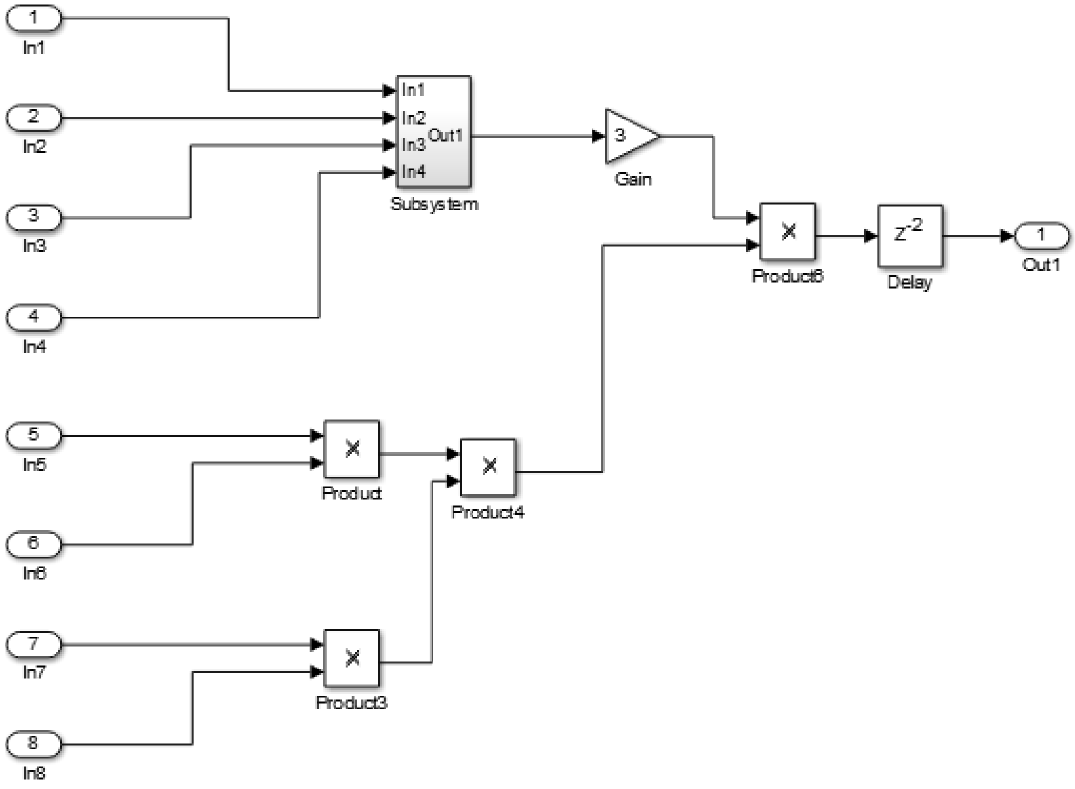
- QPSK Modulator Baseband
- BPSK Demodulator Baseband
- BPSK Modulator Baseband
- PN Sequence Generator
- Repeat
- HDL Counter
- LMS Filter
- Sine Wave
- Viterbi Decoder
- Triggered Subsystem
- Counter Limited
- Counter Free-Running
- Frame Conversion

What Is Hierarchical Distributed Pipelining?

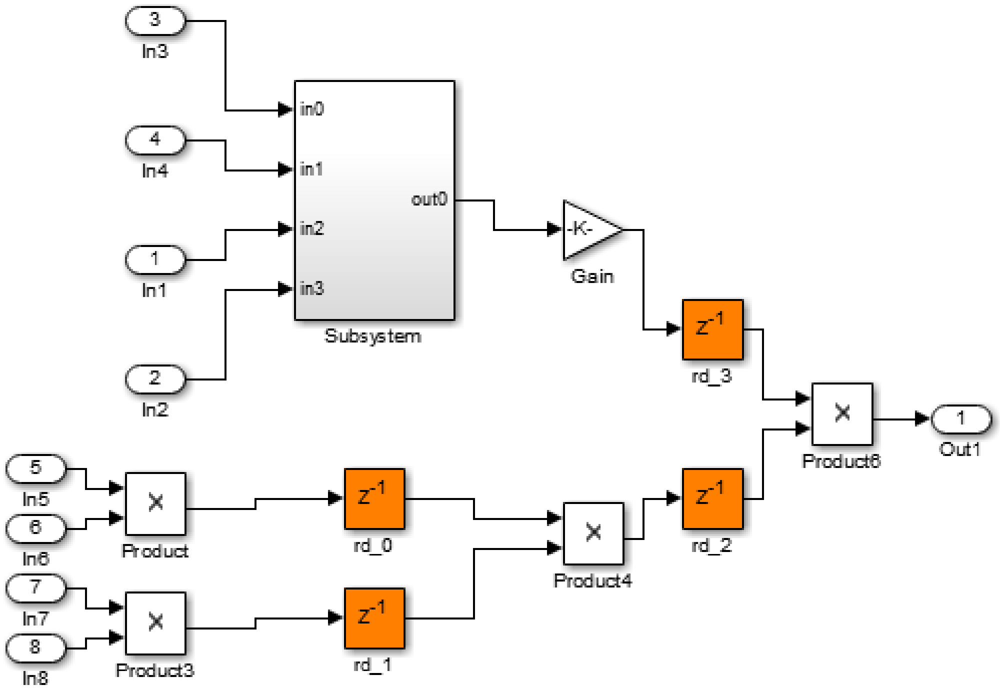
Hierarchical distributed pipelining extends the scope of distributed pipelining by moving delays across hierarchical boundaries within a subsystem while preserving subsystem hierarchy.

If a subsystem in the hierarchy does not have distributed pipelining enabled, HDL Coder does not move delays across that subsystem.

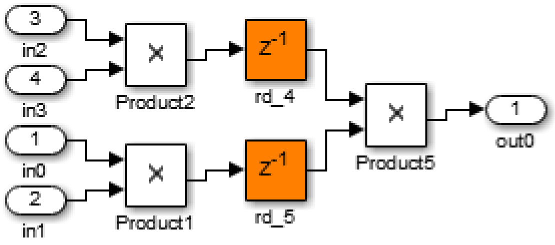
For example, the following model has one level of subsystem hierarchy:



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 distributed pipelining using the UI:

1 Right-click the DUT subsystem and select **HDL Code** > **HDL Coder Properties**.
2 In the **HDL Code Generation** > **Global Settings** pane, select the **Optimization** tab.
3 Select **Hierarchical distributed pipelining** and click **OK**.

To enable hierarchical distributed pipelining, on the command line, enter:

```
hdlset_param('modelname', 'HierarchicalDistPipelining', 'on')
```

To disable hierarchical distributed pipelining, on the command line, enter:

```
hdlset_param('modelname', 'HierarchicalDistPipelining', 'off')
```

Limitations of Hierarchical Distributed Pipelining

Hierarchical distributed pipelining must be disabled if your DUT subsystem contains a model reference.

Distributed Pipelining Workflow

For an example that shows how to use distributed pipelining to reduce your critical path, including delay insertion, see “Reduce Critical Path with Distributed Pipelining” on page 15-59.

Selected Bibliography

Leiserson, C.E, and James B. Saxe. “Retiming Synchronous Circuitry.” *Algorithmica.* Vol. 6, Number 1, 1991, pp. 5-35.

See Also

`DistributedPipeliningBarriers` | `DistributedPipeliningPriority` | `HierarchicalDistPipelining`

More About

- “Insert Distributed Pipeline Registers in a Subsystem” on page 15-42

Constrained Output Pipelining

In this section...

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:

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, 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.

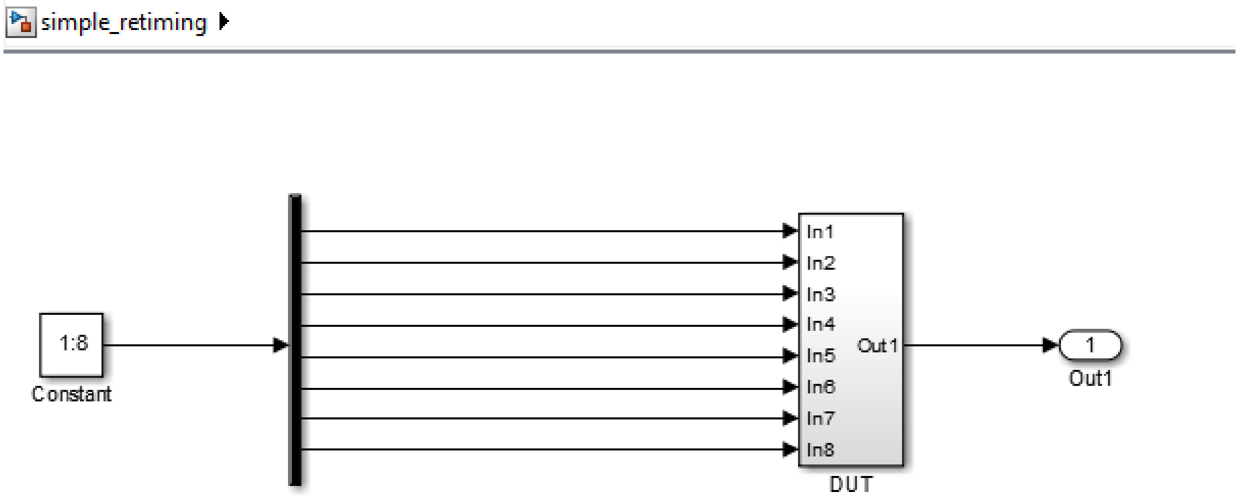
Reduce Critical Path with Distributed Pipelining

This example shows how to reduce your critical path using distributed pipelining, hierarchical distributed pipelining, output pipelining, and constrained output pipelining.

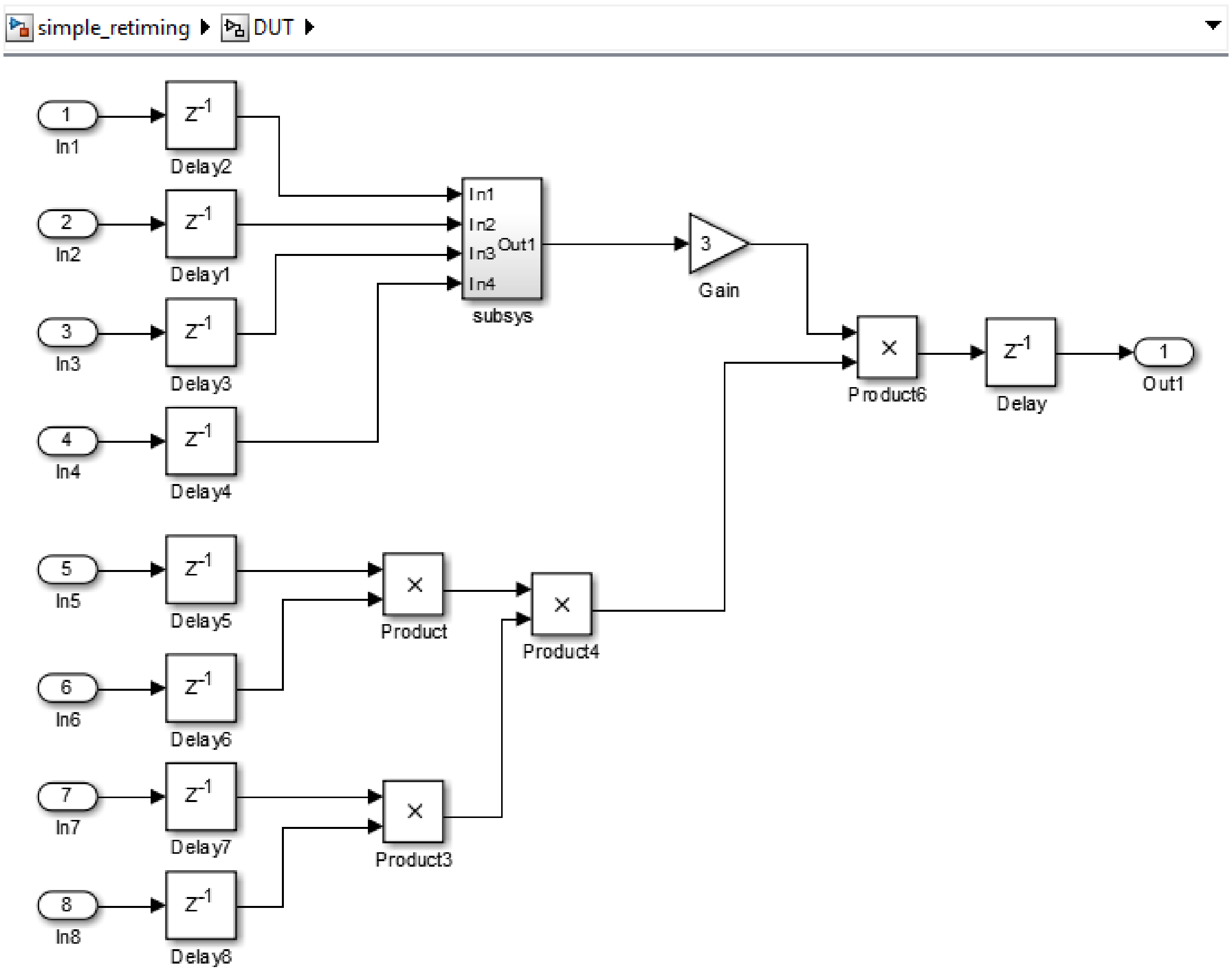
Before you begin, make sure that you have a synthesis tool set up. If you do not have a synthesis tool set up, you can follow this example, but you will not see maximum clock period results in the Result subpane. This example uses Xilinx ISE, but you can use another synthesis tool.

Open the `simple_retiming` model.

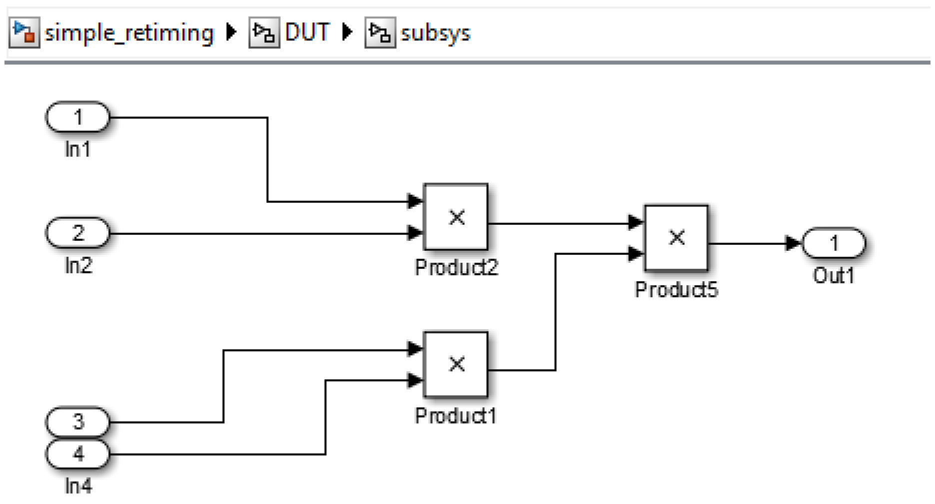
```
addpath(fullfile(docroot,'toolbox','hdlcoder','examples'));
simple_retiming
```



The top-level subsystem is the design under test (DUT). The `DUT` subsystem contains one subsystem, `subsys`, and other blocks.



The `subsys` block in `DUT` contains a copy of the three Product blocks in the lower half of the diagram.



Right-click the DUT subsystem and select **HDL Code** > **HDL Workflow Advisor** to open the HDL Workflow Advisor.

In the **Set Target** > **Set Target Device and Synthesis Tool** pane, for **Synthesis tool**, select **Xilinx ISE**.

The **FPGA Synthesis and Analysis** task appears.

On the left, expand the **FPGA Synthesis and Analysis** > **Perform Synthesis and P/R** item.

Right-click **Perform Logic Synthesis** and select **Run to Selected Task**.

```
Minimum period: 36.042ns (Maximum Frequency: 27.745MHz)
```

When HDL Coder finishes, you see the minimum clock period near the bottom of the **Results** subpane.

Next, use distributed pipelining to improve your timing results.

In the model, right-click the DUT subsystem and select **HDL Code** > **HDL Block Properties**.

In the HDL Block Properties dialog box, for **DistributedPipelining**, select **on** to enable distributed pipelining, and click **OK**.

In the `DUT` subsystem, right-click the `subsys` block, select **HDL Code** > **HDL Block Properties**, and for **DistributedPipelining**, select **on**. Click **OK**.

In the HDL Workflow Advisor, in the **HDL Code Generation** > **Set Code Generation Options** > **Set Basic Options** pane, enable **Generate optimization report**. Click **Apply**.

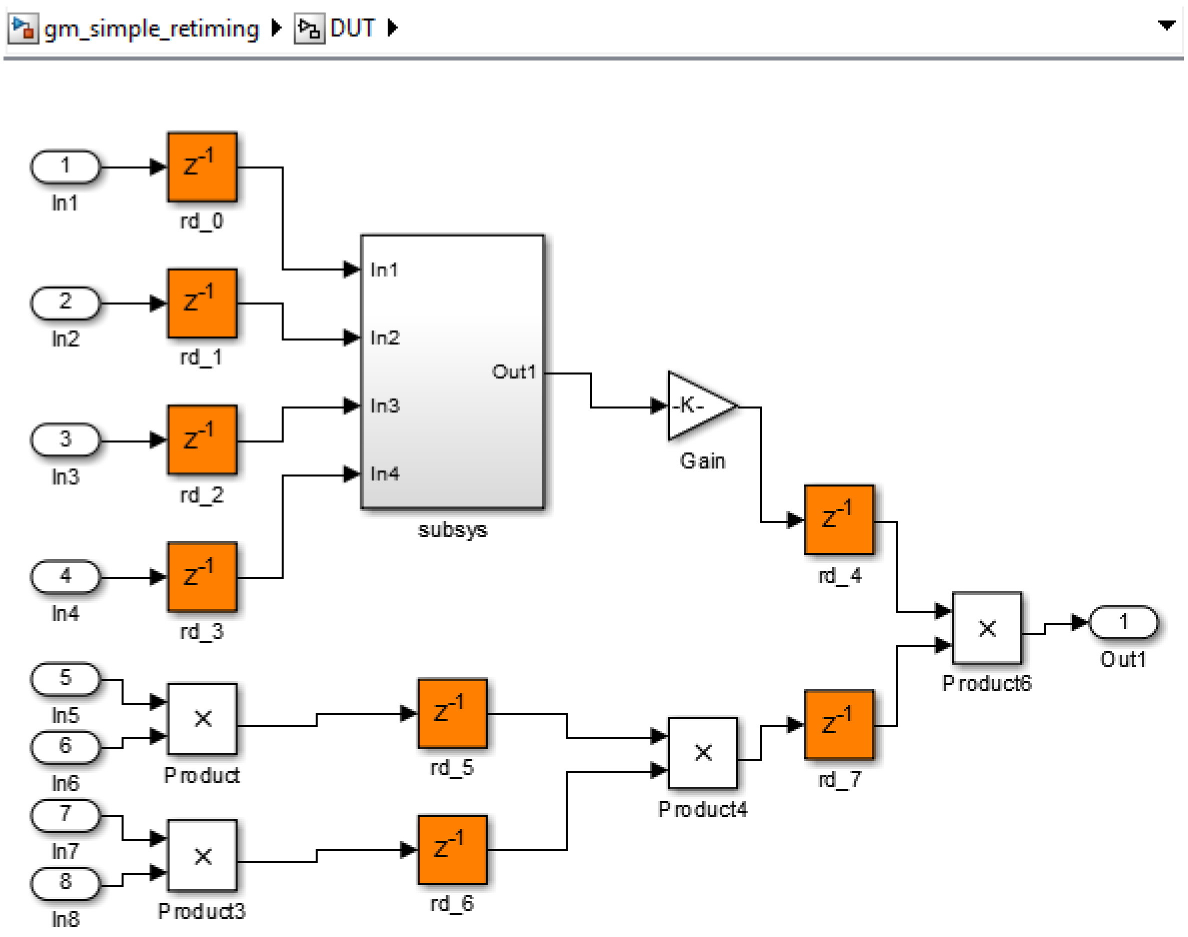
In the HDL Workflow Advisor, right-click **FPGA Synthesis and Analysis** > **Perform Synthesis and P/R** > **Perform Logic Synthesis** and select **Run to Selected Task**.

```
Minimum period: 22.025ns (Maximum Frequency: 45.403MHz)
```

The maximum clock frequency has increased.

In the Code Generation Report, click **Distributed Pipelining** to open the distributed pipelining report.

Under Generated Model, click the `gm_simple_retiming` link to open the generated model. You can see that HDL Coder redistributed the delay blocks.



In the `subsys` block, there are no delay blocks because hierarchical distributed pipelining is not enabled.

Next, use hierarchical distributed pipelining to further decrease the critical path.

In the HDL Workflow Advisor, in the **HDL Code Generation** > **Set Code Generation Options** > **Set Advanced Options** > **Optimization tab**, enable **Hierarchical distributed pipelining** and click **Apply**.

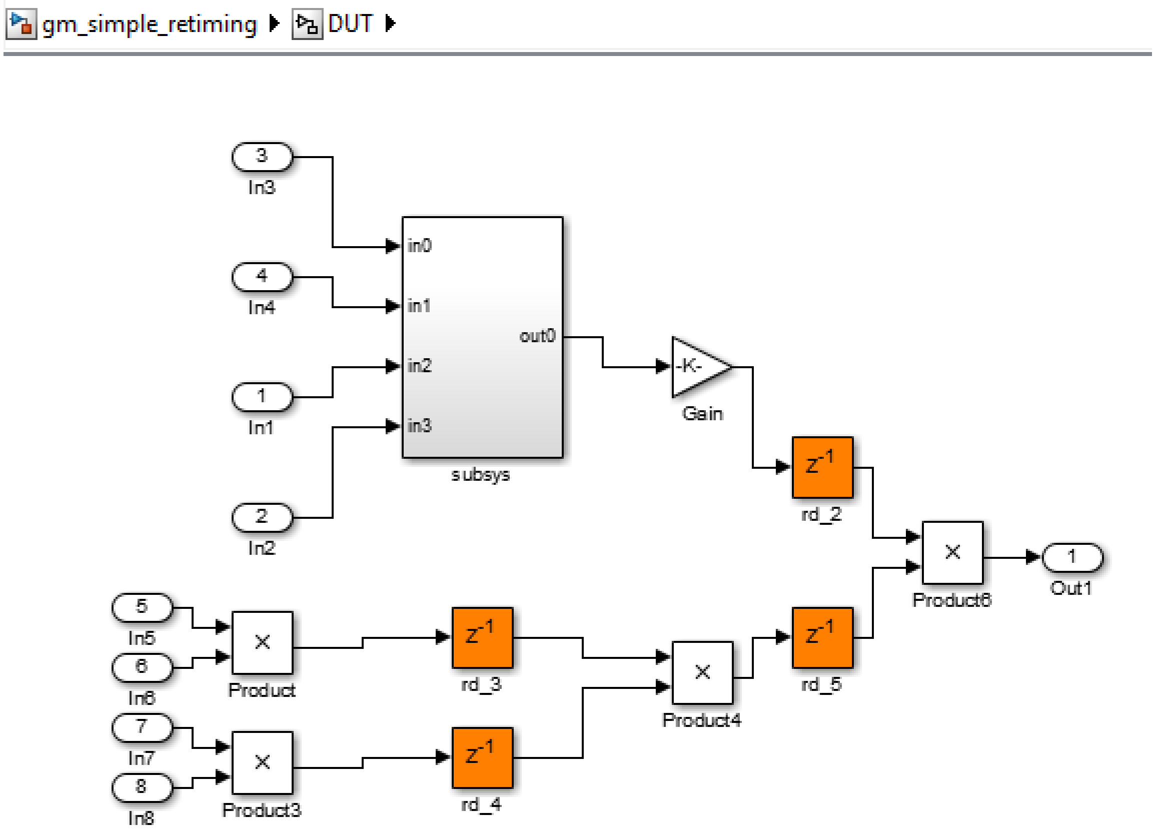
Right-click **FPGA Synthesis and Analysis** > **Perform Synthesis and P/R** > **Perform Logic Synthesis** and select **Run to Selected Task** to rerun synthesis.

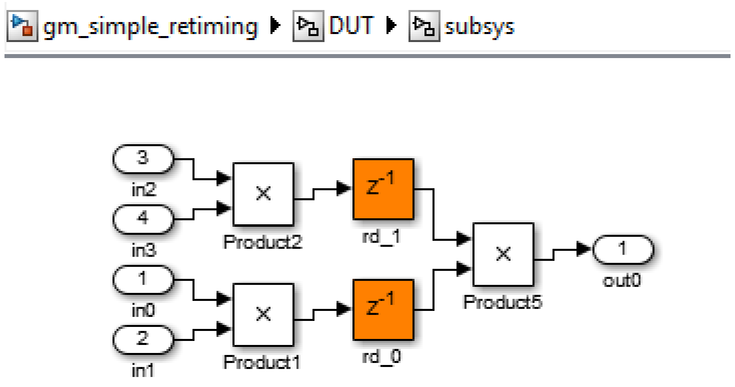
```
Minimum period: 17.263ns (Maximum Frequency: 57.928MHz)
```

The maximum clock frequency has increased.

In the Code Generation Report, click **Distributed Pipelining** to open the distributed pipelining report. Click the `gm_simple_retiming` link to open the generated model.

HDL Coder has distributed delays in the DUT and within the subsystem block.





Now, use constrained output pipelining to reduce the critical path further.

In the `simple_retiming` model, open the `subsys` block within the `DUT` subsystem.

Right-click the `Product5` block, and select **HDL Code** > **HDL Block Properties**.

For **OutputPipeline**, enter 1, and for **ConstrainedOutputPipeline**, enter 1. Click **OK**.

This adds a pipeline register and constrains it at the output of `Product5`.

In the HDL Workflow Advisor, right-click **Set Target** > **Set Target Device and Synthesis Tool** and select **Reset This Task**.

Right-click **FPGA Synthesis and Analysis** > **Perform Synthesis and P/R** > **Perform Logic Synthesis** and select **Run to Selected Task** to rerun synthesis.

```
Minimum period: 10.266ns (Maximum Frequency: 97.410MHz)
```

The maximum clock frequency is now higher.

Clock-Rate Pipelining

In this section...

For speed and area optimizations that insert pipeline registers, clock-rate pipelining identifies multi-cycle paths in your design. It inserts pipeline registers at the clock rate instead of the data rate. Clock-rate pipelining improves clock frequency and reduces area usage without introducing additional latency or adding minimal latency.

Rationale for Clock-Rate Pipelining

HDL Coder restructures your design architecture to implement your Simulink design in hardware. For some block implementations and optimizations, the coder must introduce pipelining.

For example, the following options introduce pipelining:

- Multi-cycle block implementations
- Input and output pipelining
- Distributed pipelining
- Floating-point library mapping
- Resource sharing
- Streaming

By default, in slow paths, these pipeline registers operate at the slow data rate. With clock-rate pipelining, they operate at the clock rate. Clock-rate pipelining optimizes slow data paths in your design. It is an alternative to using multicycle path constraints with your synthesis tool.

How Clock-Rate Pipelining Works

Clock-rate pipelining identifies slow paths or regions in the model by analyzing the block data rates. Blocks that operate at a slower rate than the DUT base sample time are part of the slow path and therefore are candidates for clock-rate pipelining. If `Oversampling` is greater than 1, the DUT sample time is also slower than the actual clock rate.

Clock-rate pipelining identifies the largest slow region within the model. It enables optimizations to introduce pipeline delays at the clock-rate within these regions. The coder computes the number of clock-rate pipeline registers available for optimizations by analyzing the block sample times.

The coder determines the maximum number of clock-rate pipeline registers it can insert based on the DUT-to-block sample time ratio and the oversampling factor:

Maximum number of clock-rate delays = (*block_rate* ÷ *DUT_base_rate*) × `Oversampling`

Clock-rate pipelining affects only pipeline registers introduced by the coder for optimizations and block implementations. It does not affect existing delays in the original model, which continue to operate at the rate at which you modeled them.

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 enable 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 help you adjust for the difference in timing, the coder generates messages that provide 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

- Enable hierarchy flattening to increase the opportunities for clock-rate pipelining. When sample-time delays are at the same level of hierarchy, clock-rate pipelining works optimally.
- 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.
- When you specify `Oversampling` for your design, specify the maximum oversampling factor. The maximum oversampling factor provides HDL Coder with more flexibility for optimizing your design.

Specify Clock-Rate Pipelining

You can set clock-rate pipelining for an entire model or for subsystems within the top-level DUT subsystem for finer control.

Set Clock-Rate Pipelining for a Model

Clock-rate pipelining is enabled by default. You can disable clock-rate pipelining in one of the following ways:

- In the HDL Workflow Advisor, in the **HDL Code Generation** > **Set Code Generation Options** > **Set Optimization Options** > **Pipelining** tab, select **Clock-rate pipelining**.
- In the Configuration Parameters dialog box, on the **HDL Code Generation** > **Target and Optimizations** > **Pipelining** tab, select **Clock-rate pipelining**
- At the command line, use the `makehdl` or `hdlset_param` function to set the `ClockRatePipelining` property to `off`.

Disable Clock-Rate Pipelining for a Subsystem

You can disable clock-rate pipelining for an entire model, or 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.

By default, clock-rate pipelining is enabled at the model level. To disable clock-rate pipelining for a subsystem within the top-level DUT subsystem, set **ClockRatePipelining** to `off` for that subsystem.

To learn how to set clock-rate pipelining for a subsystem, see “Set Clock-Rate Pipelining For a Subsystem” on page 12-7.

Diagnostics for Clock-Rate Pipelining

The coder generates a block highlighting script. Run that script to highlight blocks in your model that are obstacles to clock-rate pipelining.

Sometimes, if the coder 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.

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 Function, if it uses persistent variables

- MATLAB System, if it uses persistent variables
- Rate Transition
- Serializer1D
- Simple Dual Port RAM
- Single Port RAM
- Subsystem, if `FlattenHierarchy` is not enabled

The coder 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 System Toolbox blocks.
- DSP System Toolbox blocks, except for Delay and Discrete FIR Filter.
- Stateflow blocks.

See Also
`ClockRatePipelining` | `Oversampling`

Adaptive Pipelining

In this section...

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 number of pipeline registers, the optimization considers the target device, target frequency, multiplier word lengths, and the settings in the HDL Block Properties.

Best Practices

- For HDL Coder to insert adaptive pipelines, specify the target device. When your design has multipliers, specify the target device and the target frequency.
- 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.
- By using clock-rate pipelining with adaptive pipelining, HDL Coder inserts pipelines at a faster clock-rate instead of the slower data-rate. With clock-rate pipelining, the best practice is to design your DUT at one rate, and then specify the **Oversampling factor**.
- By using resource sharing with adaptive pipelining, HDL Coder shares resources and inserts pipeline registers, which saves area and timing.

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.

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)
- Downsample
- Rate Transition
- Product
- Gain
- Multiply-Add

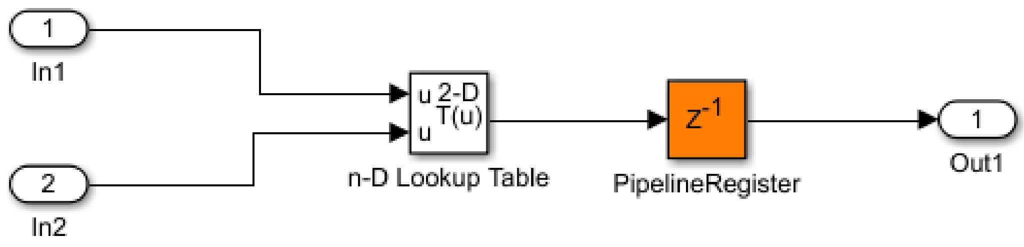
Lookup Tables

To insert adaptive pipelines for lookup table blocks:

1. Specify the target device.
2. Make sure that **Interpolation method** is set to `Flat`.

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 Virtex5 as the target FPGA device.



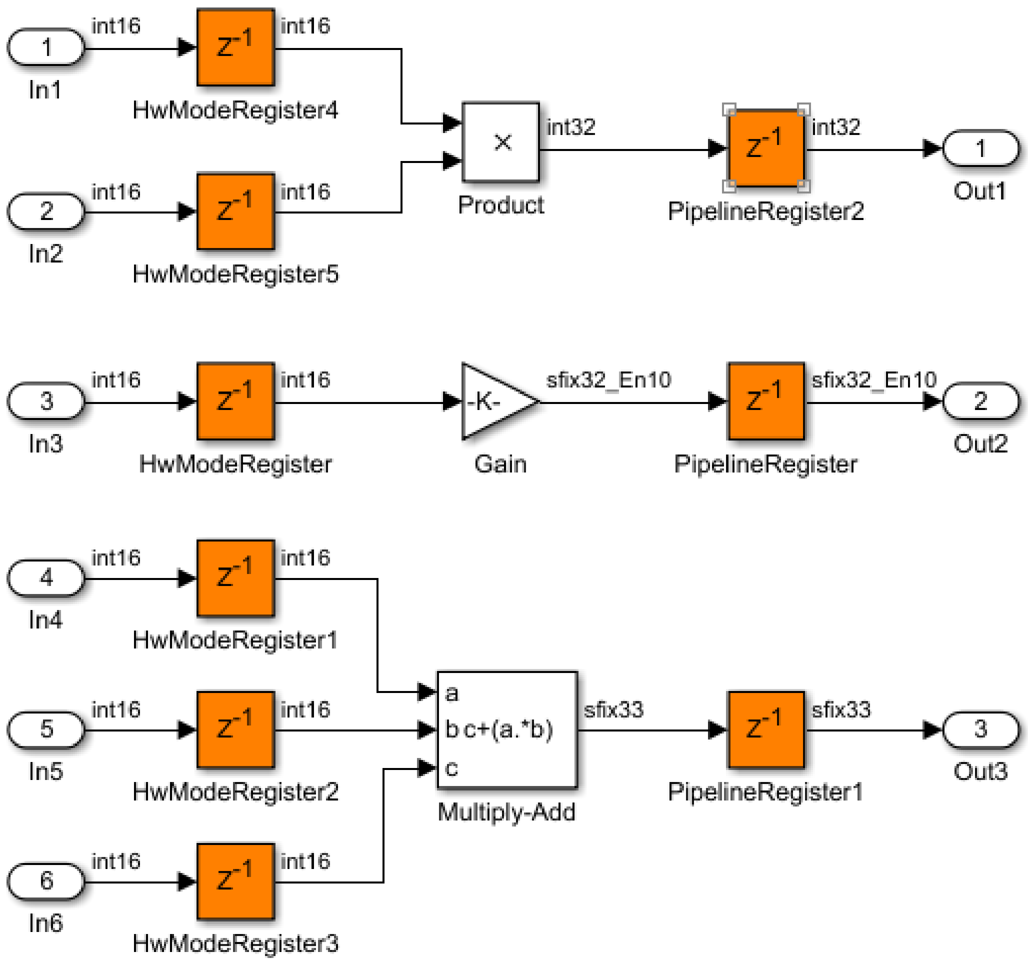
Product, Gain, and Multiply-Add

To insert adaptive pipelines for these blocks:

1. Specify the target device.
2. Specify a target frequency greater than zero.

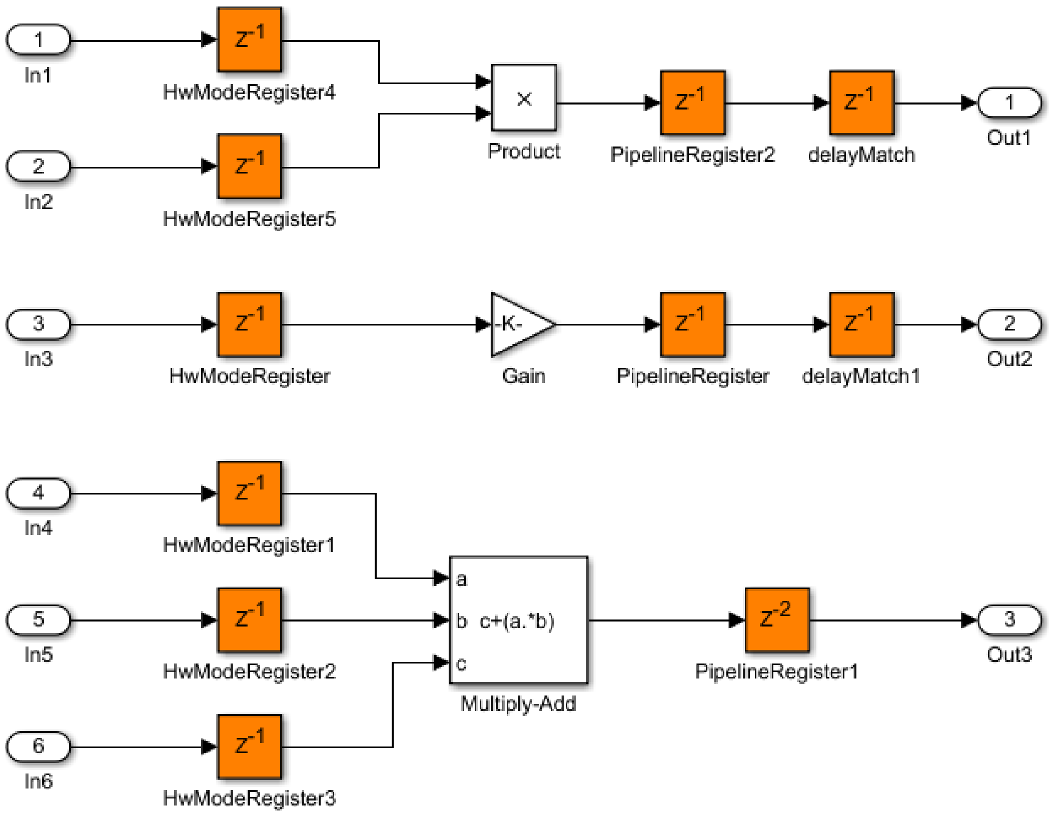
When generating code, HDL Coder inserts registers at the input and output ports of the blocks. The combination of multipliers with the registers can potentially map to DSP units on the target device.

This figure is the generated model for the Product, Gain, and Multiply-Add blocks with Altera 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 Virtex5 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.

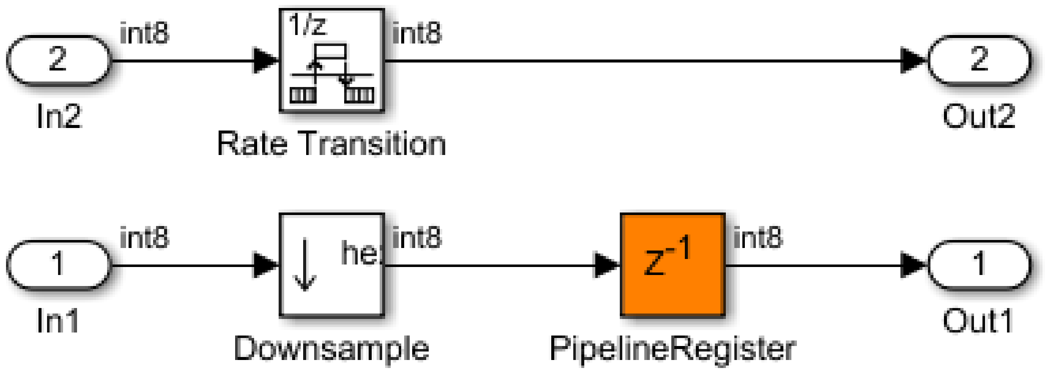
Rate Transition and Downsample

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 Virtex5 as the target FPGA device.



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. By default, adaptive pipelining is enabled at the model level.

Set Adaptive Pipelining for a Model

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** > **Target and Optimizations** > **Pipelining** tab, select **Adaptive pipelining**
- At the command line, use the `makehdl` or `hdlset_param` function to set the `AdaptivePipelining` property to `off`.

Set Adaptive Pipelining for a Subsystem

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 12-5.

Adaptive Pipelining Information in Reports

To see the adaptive pipelining information in the report, before you generate code for each subsystem or model reference, enable the Code Generation report. In the Code Generation report, select the **Adaptive Pipelining** section of the Optimization report.

To enable the Code Generation report, in the Configuration Parameters dialog box, on the **HDL Code Generation** pane, enable **Generate optimization report**.

The 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 lists the various blocks for which HDL Coder inserted adaptive pipelines, and the number of pipeline registers inserted.

See also “Optimization Report” on page 16-9.

See Also

`AdaptivePipelining`

More About

- “AdaptivePipelining” on page 12-5
- “Clock-Rate Pipelining” on page 15-66

Find Estimated Critical Paths Without Synthesis Tools

In this section...

When to Use Critical Path Estimation

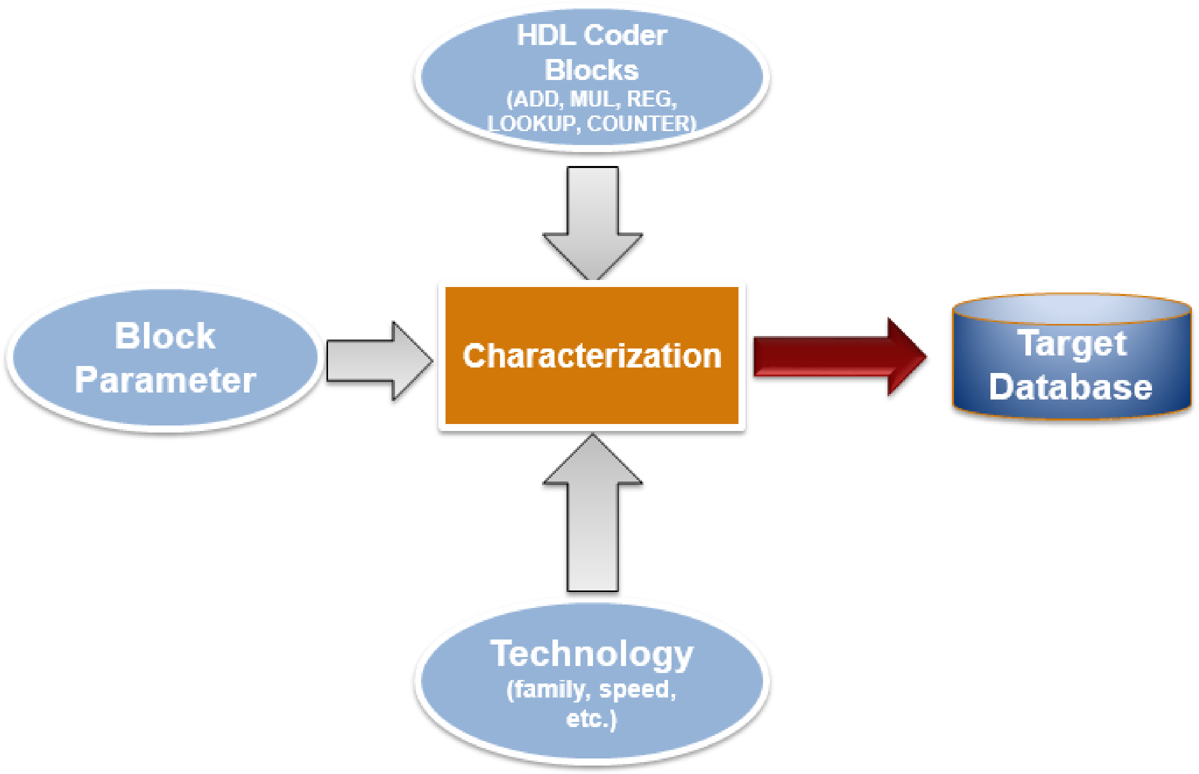
Use critical path estimation to find the most likely timing for critical paths in your design quickly. With critical path estimation, you can quickly iterate through finding and pipelining the critical path until your model design is likely to meet timing.

Estimating the critical path without running synthesis tools is a faster iteration, however it can lead to some inaccuracy. Critical path estimation is intended to speed up the design iteration process.

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 creates timing databases by characterizing basic design components, such as Simulink blocks, block architectures, and subcomponents of those blocks, for specific target devices.


HDL Coder Blocks (ADD, MUL, REG, LOOKUP, COUNTER)
Block Parameter
Characterization
Target Database
Technology (family, speed, etc.)

Supported Target Devices for Critical Path Estimation

HDL Coder has timing databases for the following target devices:

- Altera Cyclone V
- Xilinx Virtex®-7, speed grade -1
- Xilinx Zynq®, speed grade -1

Analyzing the Design

The coder analyzes your model design to decompose it into the blocks and subcomponents in the timing databases. If your design is composed of blocks or subcomponents in the timing databases, the coder 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, it is likely that you will still find the critical path, but it is possible that the timing numbers are not accurate.

How to Use Critical Path Estimation

1. Enable critical path estimation.

   - In the HDL Workflow Advisor, in the **HDL Code Generation** > **Set Code Generation Options** > **Set Basic Options**, select **Generate high-level timing critical path report**.
   - At the command line, set `CriticalPathEstimation` to `on`:

     ```
     hdlset_param('modelname','CriticalPathEstimation','on');
     ```
2. Generate HDL code.

   The critical path estimation message in the MATLAB Command Window or the HDL Workflow Advisor **Result** pane includes a link that you can click to run the highlighting script.
3. Run the critical path highlighting script to see which blocks are on the critical path.
4. Break the critical path by adding pipeline registers.
5. Regenerate code.

HDL Code Generation Behavior

When you enable critical path estimation, it is possible that the generated HDL code is different for a Delay block that has an external reset or an enable port. Following are other Simulink blocks for which the generated HDL code can be potentially different.

- Unit Delay Enabled
- Unit Delay Resettable

- Unit Delay Enabled Resettable
- Enabled Delay
- Resettable Delay
- Tapped Delay
- Discrete FIR Filter
- Biquad Filter
- MATLAB Function

Characterized Blocks

The following blocks have been characterized and are part of the timing database for each supported target device.

- Abs
- Add block with not more than two inputs.
- Bit Concat
- Bit Extract
- Bit Shift
- Bit Slice
- Bitwise Operator
- Bus Creator
- Bus Selector
- Buffer
- Complex to Real-Imag
- Compare To Constant
- Constant
- Convert 1-D to 2-D
- Counter Free-Running
- Counter Limited
- Data Type Conversion
- Delay
- Demux

- Deserializer1D
- Dual Port RAM
- Dual Rate Dual Port RAM
- Gain
- HDL Counter
- HDL Reciprocal
- Logical Operator
- Product
- Multiport Switch
- n-D Lookup Table
- Rate Transition
- Reciprocal Sqrt
- Relational Operator
- Reshape
- Selector
- Serializer1D
- Signal Conversion
- Signal Specification
- Simple Dual Port RAM
- Single Port RAM
- Switch
- Sqrt
- Unary Minus
- Unit Delay

Inaccuracy in Critical Path Estimation

Routing Delay

Critical path estimation tries to account for routing delay by using an estimation factor. Without running place and route, however, it is difficult to accurately account for routing delay.

Uncharacterized Blocks

HDL Coder treats blocks that are not in the list of characterized blocks, but are inferred to be combinational, as zero-delay combinational blocks. the code generator treats other blocks as registers.

Unsupported Target Devices

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.

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 15-20 and the “Area Reduction of Filter Subsystem” on page 15-85 example.

You can also use a **SharingFactor** with multichannel filters. See “Area Reduction of Multichannel Filter Subsystem” on page 15-89.

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 15-7 and the “Area Reduction of Multichannel Filter Subsystem” on page 15-89 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:

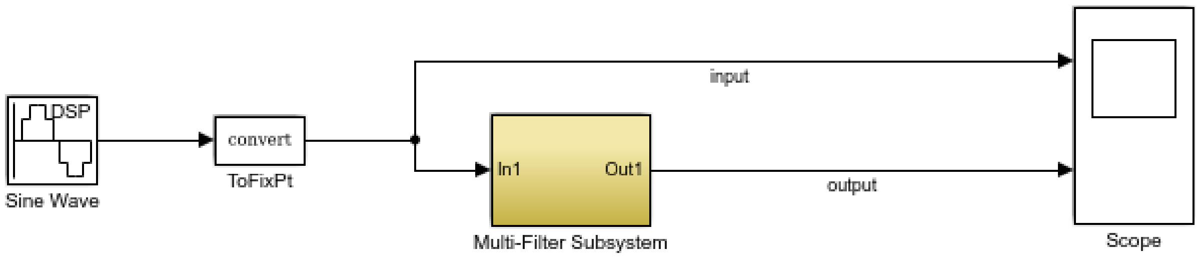
- **InputPipeline** and **OutputPipeline**
- **MultiplierInputPipeline** and **MultiplierOutputPipeline**
- **AddPipelineRegisters**

The optimization tools do not move design delays within the filter architecture. See “Distributed Pipelining” on page 8-14.

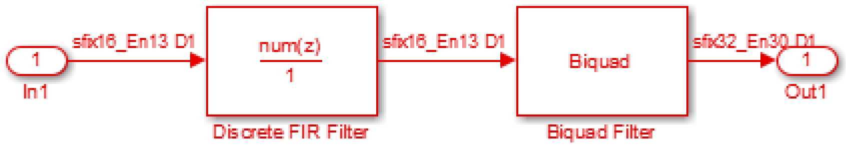
The filter block also participates in clock-rate pipelining, if enabled in **Configuration Parameters**. This feature is enabled by default. See “Clock-Rate Pipelining” on page 15-66.

Area Reduction of Filter Subsystem

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



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

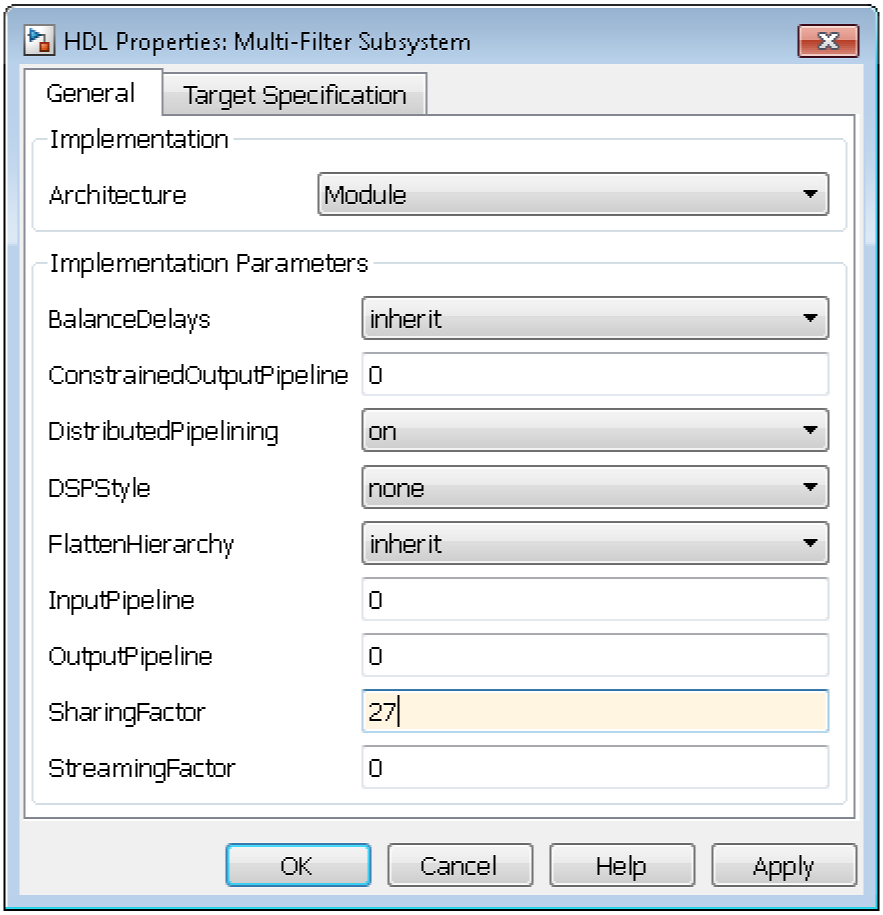


The subsystem contains a Discrete FIR Filter block and a Biquad Filter block. This design demonstrates how the optimization tools share resources between multiple filter blocks.

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**.


HDL Properties: Multi-Filter Subsystem

Implementation

Architecture	Module

Implementation Parameters

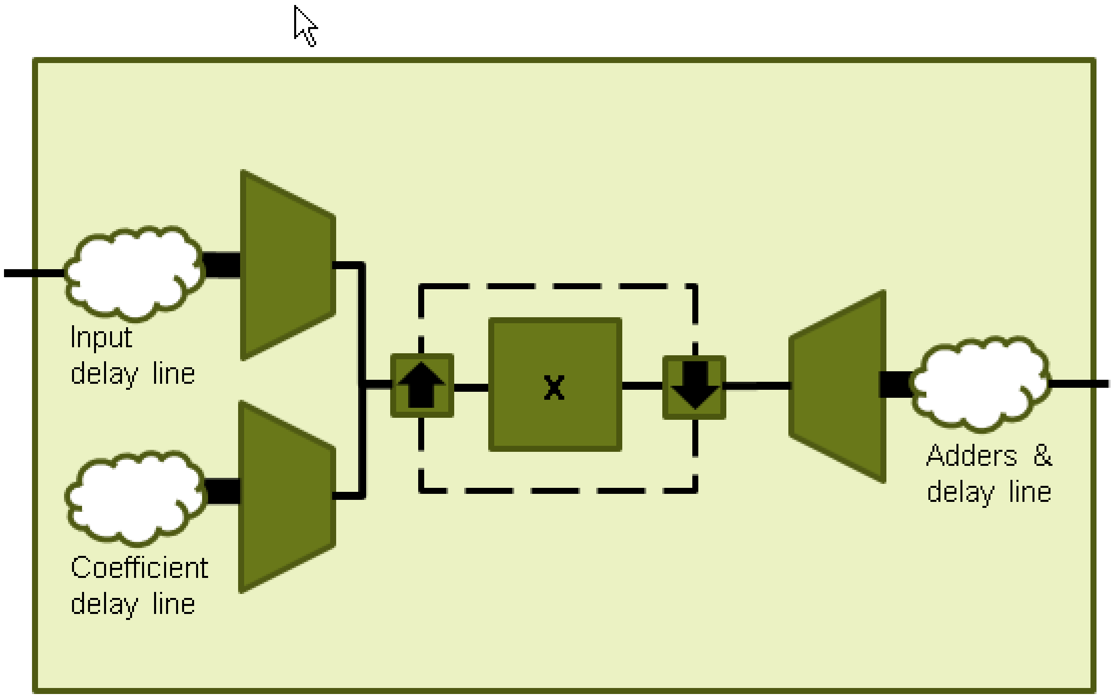
BalanceDelays	inherit
ConstrainedOutputPipeline	0
DistributedPipelining	on
DSPStyle	none
FlattenHierarchy	inherit
InputPipeline	0
OutputPipeline	0
SharingFactor	27
StreamingFactor	0

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.

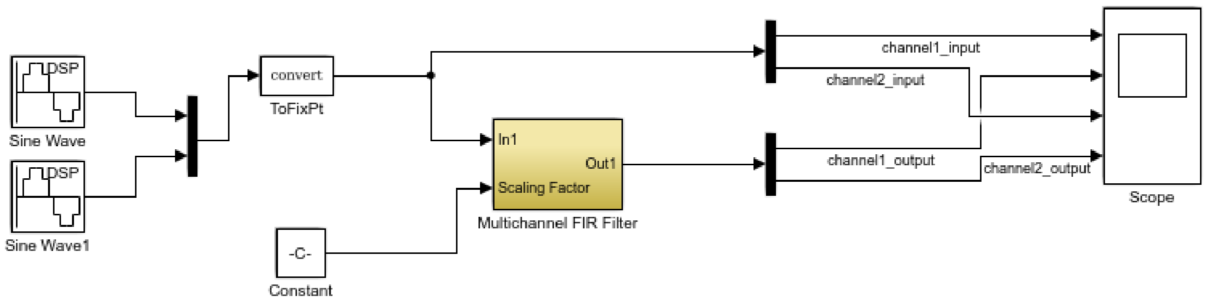


In the **Code Generation Report** window, click **High-level Resource Report**. The generated HDL code now uses one multiplier.

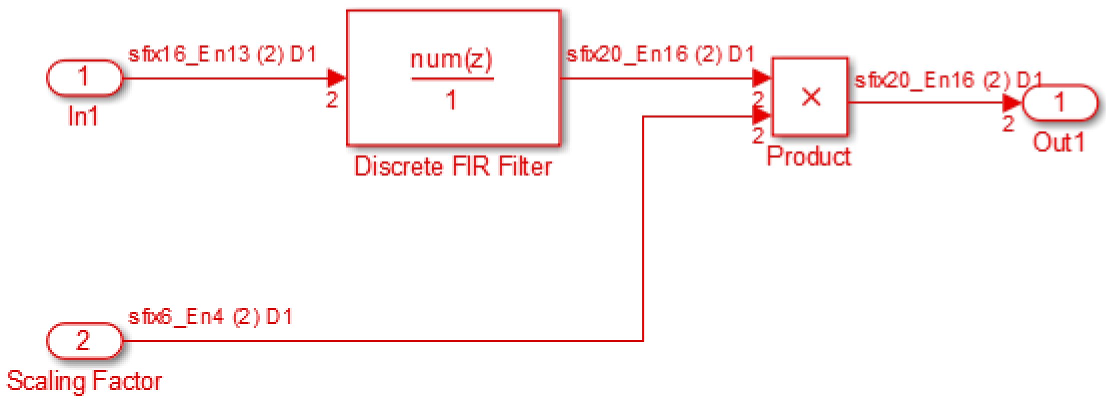
Multipliers	1
Adders/Subtractors	48
Registers	102
Total 1-Bit Registers	2457
RAMs	0
Multiplexers	7
I/O Bits	52

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.

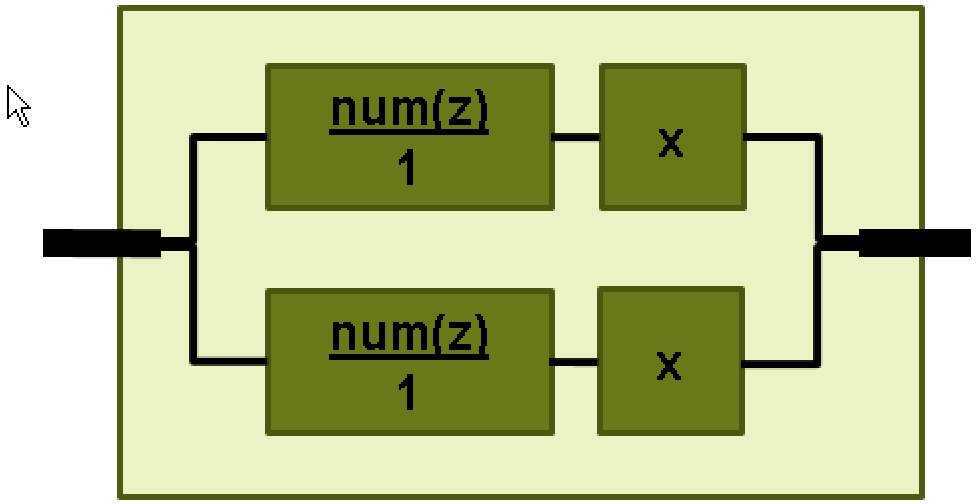


The model includes a two-channel sinusoidal signal source feeding a filter subsystem targeted for HDL code generation.

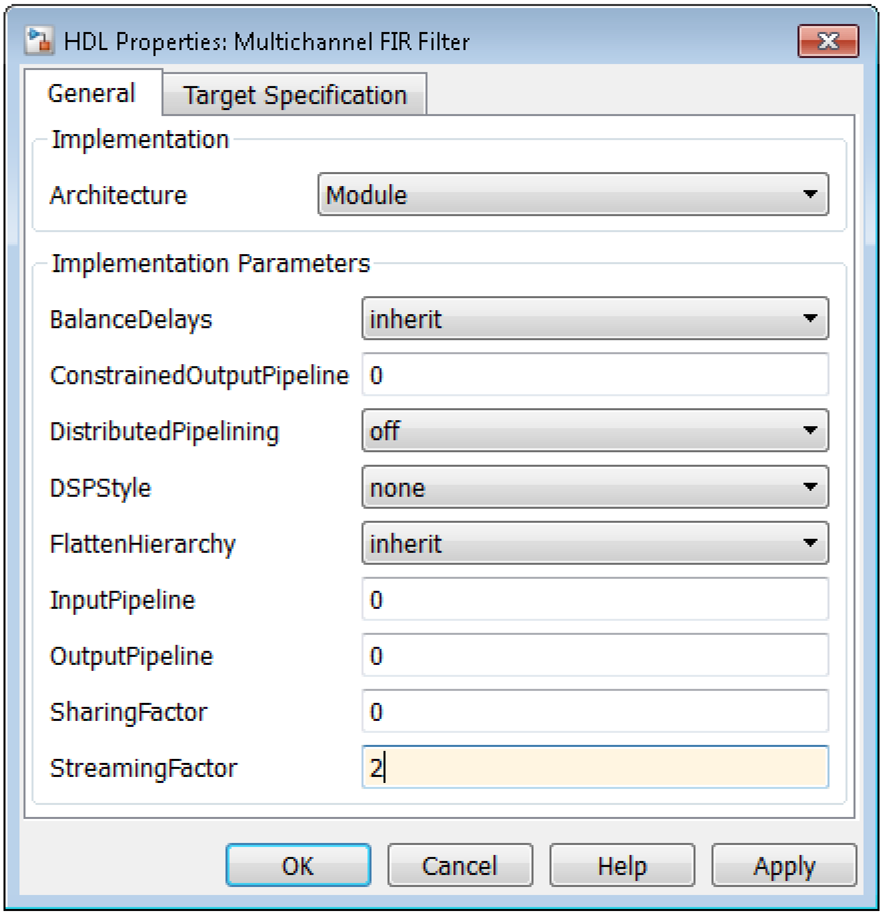


The subsystem contains a Discrete FIR Filter block and a constant multiplier. The multiplier is included to show the optimizations operating over all eligible logic in a subsystem.

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**.


HDL Properties: Multichannel FIR Filter

Implementation

Architecture	Module

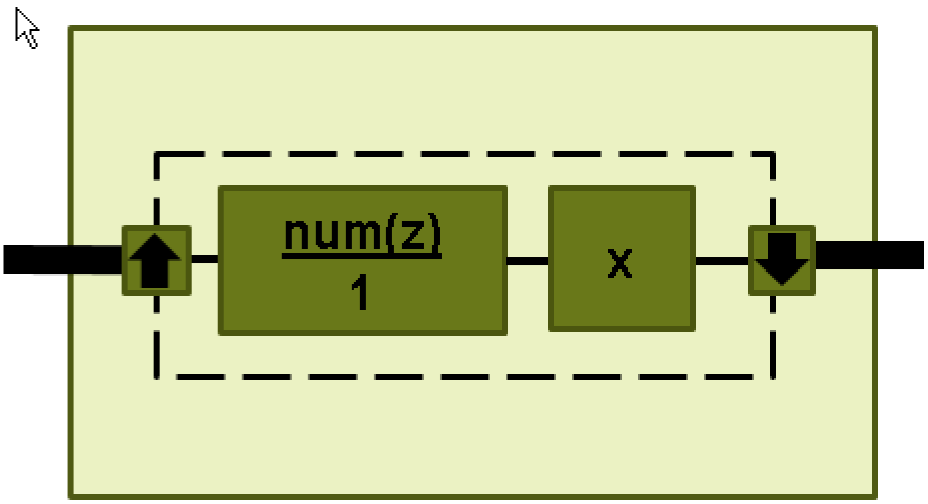
Implementation Parameters

BalanceDelays	inherit
ConstrainedOutputPipeline	0
DistributedPipelining	off
DSPStyle	none
FlattenHierarchy	inherit
InputPipeline	0
OutputPipeline	0
SharingFactor	0
StreamingFactor	2

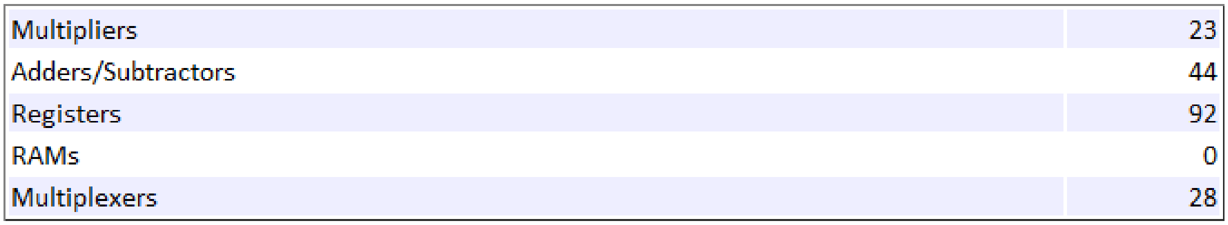
Set the **StreamingFactor** to 2, because this design is a two-channel system.

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.

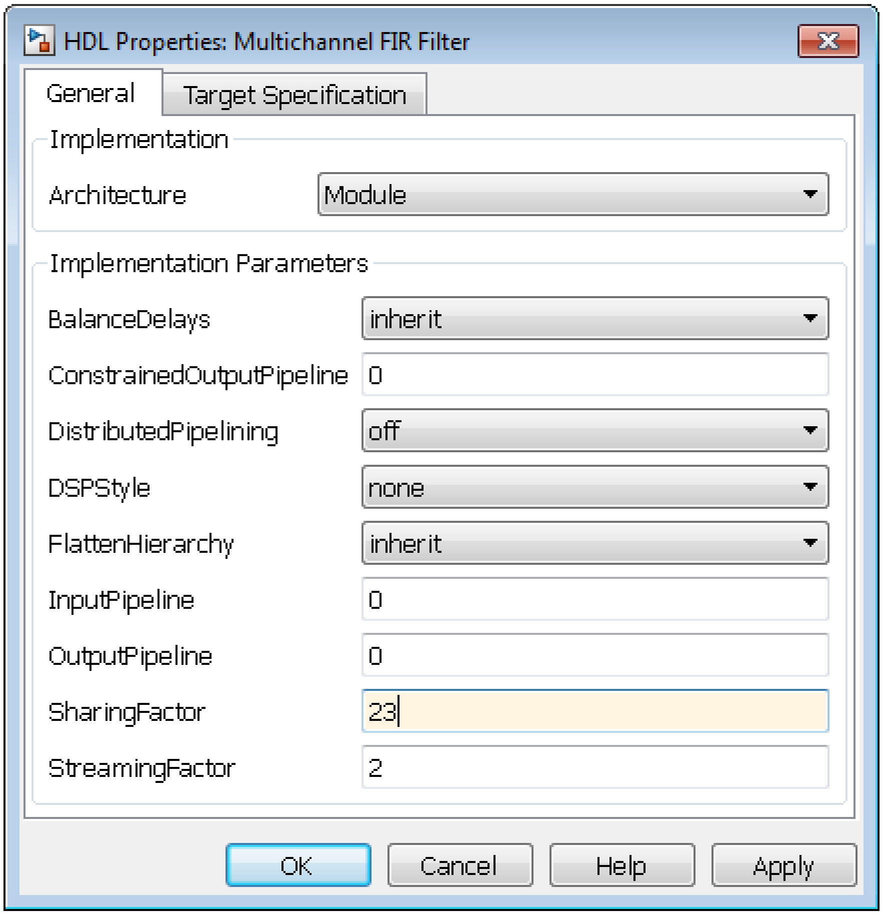


In the **Code Generation Report** window, click **High-level Resource Report**. The generated HDL code now uses 23 multipliers, compared to 46 in the nonoptimized code. The multipliers in the filter kernel and subsequent scaling are shared between the channels.



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

To apply **SharingFactor** to multichannel filters, set the **SharingFactor** to 23.


HDL Properties: Multichannel FIR Filter

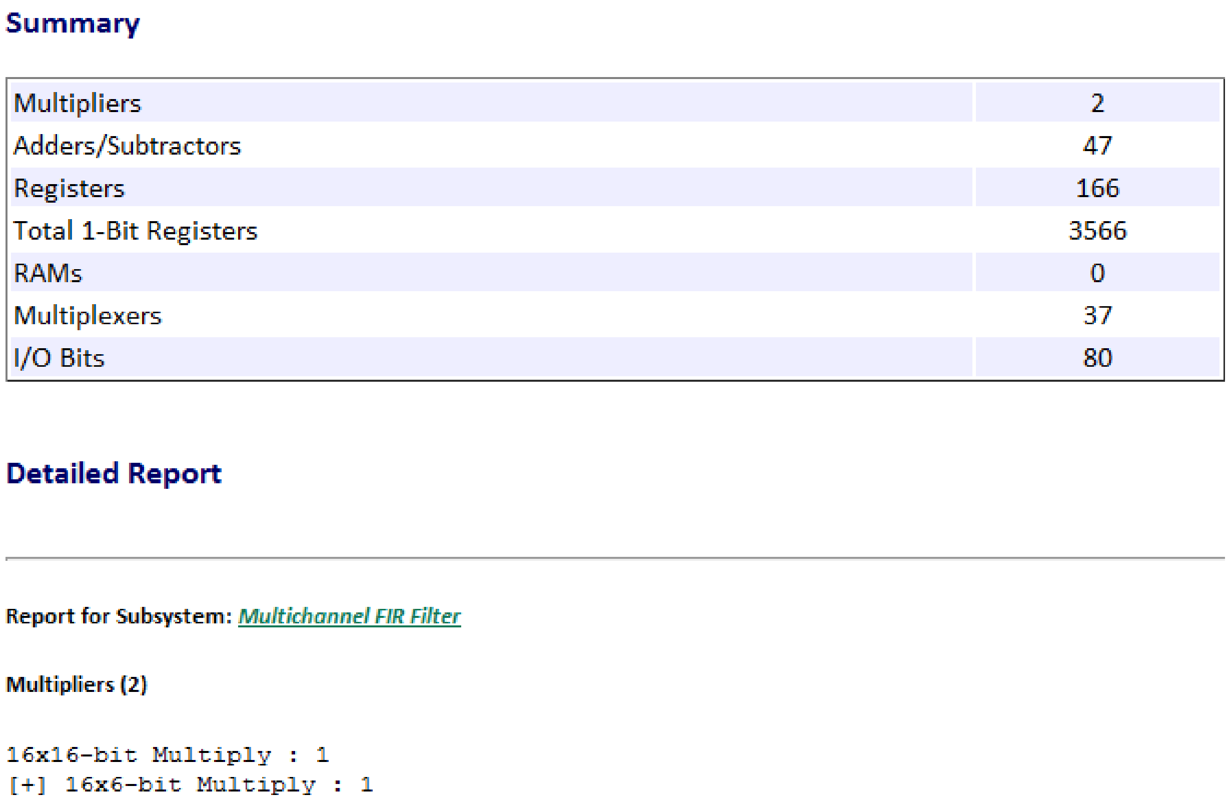
Implementation

Architecture	Module

Implementation Parameters

BalanceDelays	inherit
ConstrainedOutputPipeline	0
DistributedPipelining	off
DSPStyle	none
FlattenHierarchy	inherit
InputPipeline	0
OutputPipeline	0
SharingFactor	23
StreamingFactor	2

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
```

Code Generation Reports, HDL Compatibility Checker, Block Support Library, and Code Annotation

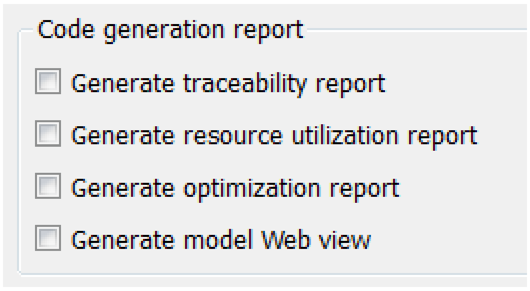
Create and Use Code Generation Reports

Information Included in Code Generation Reports

The HDL Coder software creates and displays an HDL Code Generation Report when you select one or more of the following options:

GUI option	makehdl Property
Generate traceability report	`Traceability`
Generate resource utilization report	`ResourceReport`
Generate optimization report	`OptimizationReport`
Generate model Web view	`HDLGenerateWebview`

These options appear in the **Code generation report** panel of the **HDL Code Generation** pane of the Configuration Parameters dialog box:


Code generation report

☐ Generate traceability report

☐ Generate resource utilization report

☐ Generate optimization report

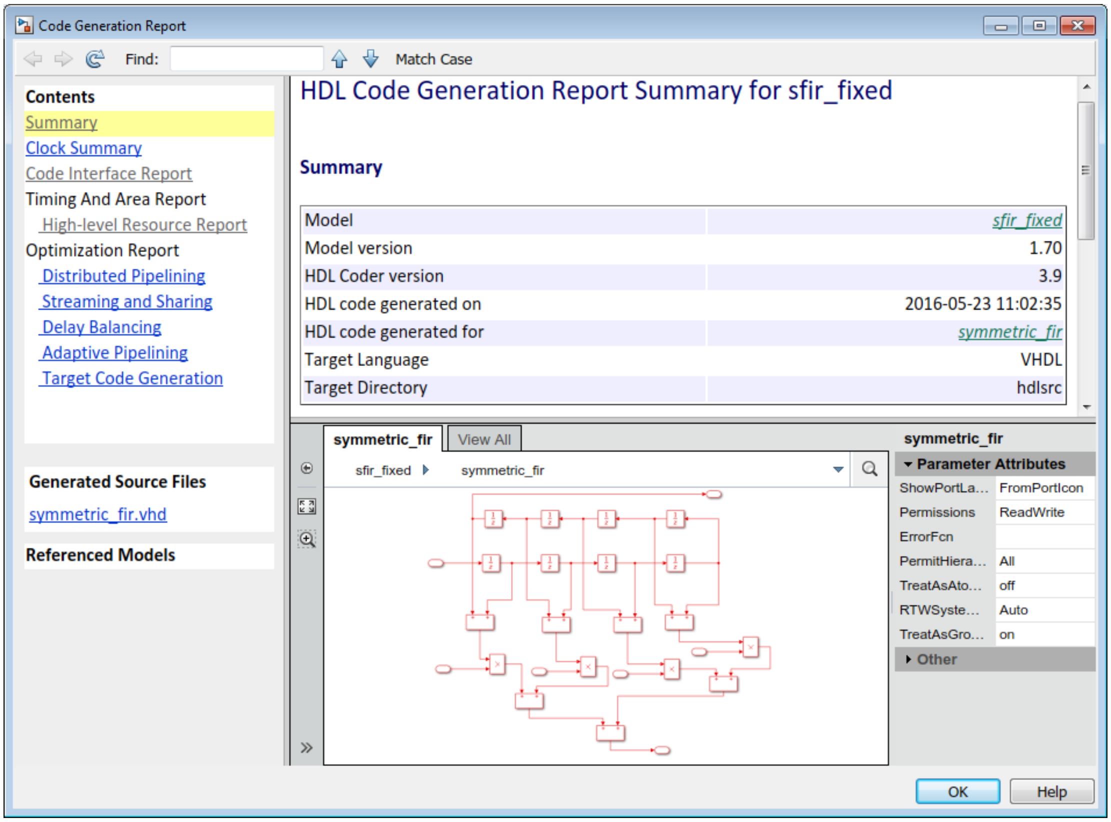
☐ Generate model Web view

The HDL Code Generation Report is an HTML report that includes a Summary, a "Code Interface Report" on page 16-4, and one or more of the following optional sections:

- Traceability Report
- Resource Utilization Report
- "Optimization Report" on page 16-9
- "Web View of Model in Code Generation Report" on page 16-35

HDL Code Generation Report Summary

All reports include a Summary section. 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.



HDL Code Generation Report Summary for sfir_fixed

Summary

Model	sfir_fixed
Model version	1.70
HDL Coder version	3.9
HDL code generated on	2016-05-23 11:02:35
HDL code generated for	symmetric_fir
Target Language	VHDL
Target Directory	hdlsrc

symmetric_fir — Parameter Attributes

ShowPortLa...	FromPortIcon
Permissions	ReadWrite
ErrorFcn	
PermitHiera...	All
TreatAsAto...	off
RTWSyste...	Auto
TreatAsGro...	on

Resource Utilization Report

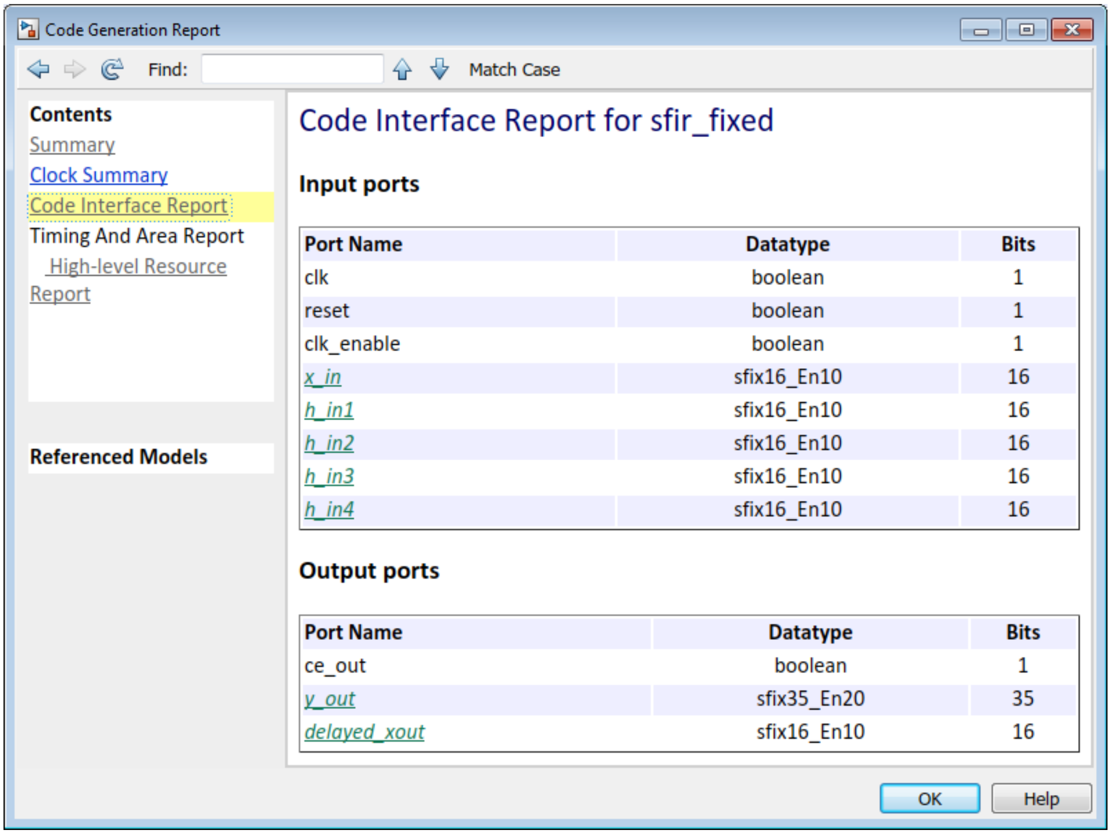
When you select **Generate resource utilization report**, HDL Coder adds a Code Generation Report that has a **Code Interface Report** and a **Timing and Area Report** section.

The **Code Interface Report** shows the DUT input and output port names, data types, and bit widths. The **Timing and Area Report** section has two subsections:

- **High-level Resource Report**: Summarizes multipliers, adders/subtractors, and registers consumed by the device under test (DUT). Includes a detailed report on the resources that each subsystem uses. Wherever possible, the detailed report links back to corresponding blocks in your model.
- **Target-Specific Report**: When you request target-specific code generation on the model, this subsection shows the resource utilization report.

Code Interface Report

The **Code Interface Report** shows the DUT input and output port names, data types, and bit widths. The report displays a link corresponding to each input and output port in your Simulink model.



Code Interface Report for sfir_fixed

Input ports

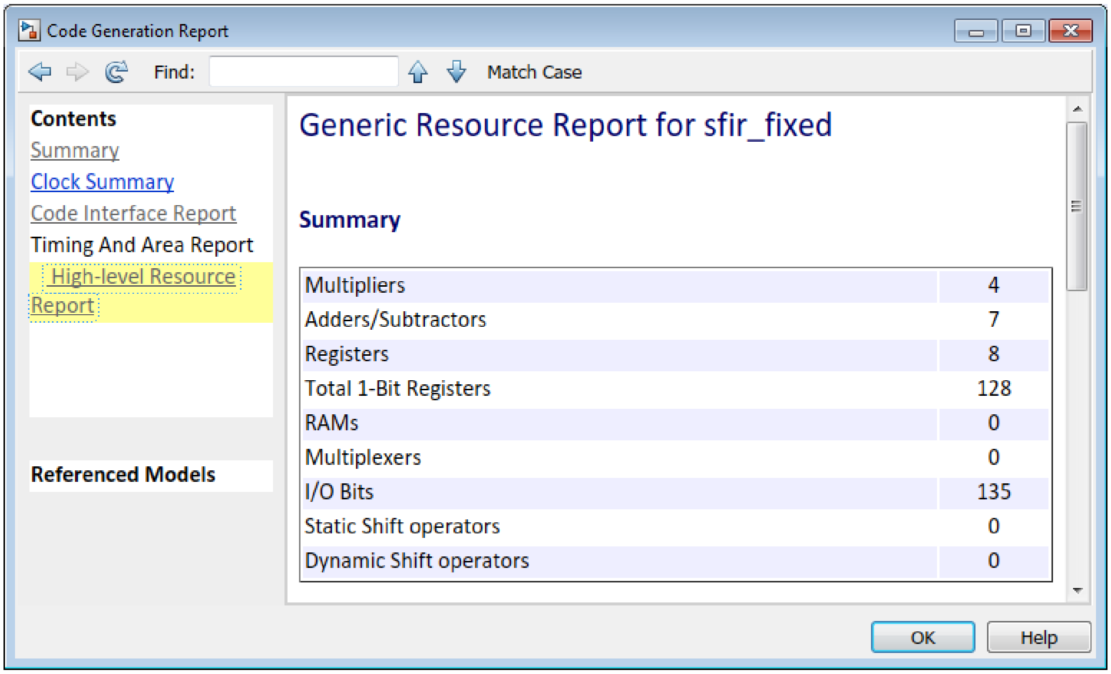
Port Name	Datatype	Bits
clk	boolean	1
reset	boolean	1
clk_enable	boolean	1
x_in	sfix16_En10	16
h_in1	sfix16_En10	16
h_in2	sfix16_En10	16
h_in3	sfix16_En10	16
h_in4	sfix16_En10	16

Output ports

Port Name	Datatype	Bits
ce_out	boolean	1
y_out	sfix35_En20	35
delayed_xout	sfix16_En10	16

High-Level Resource Report

The High-Level Resource report analyzes the effects of optimizations, such as resource sharing and streaming and displays a summary of the resources that your DUT uses. The report shows the number of 1–bit registers and I/O bits and includes resource usage for model references.



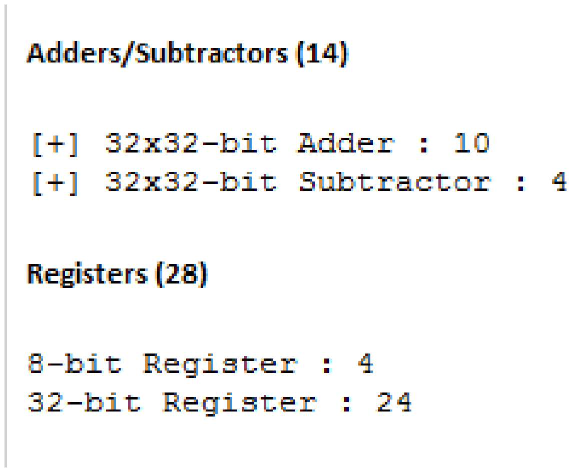
Generic Resource Report for sfir_fixed

Summary

Multipliers	4
Adders/Subtractors	7
Registers	8
Total 1-Bit Registers	128
RAMs	0
Multiplexers	0
I/O Bits	135
Static Shift operators	0
Dynamic Shift operators	0

The total number of 1–bit Registers is calculated as a sum of products over the bit widths of the registers and their frequency of occurrence. To calculate the total 1–bit registers, refer to the **Registers** section in the **Detailed Report**. For this example,

$$\text{Total 1-bit Registers} = (8x4) + (32x24) = 800 .$$



Adders/Subtractors (14)

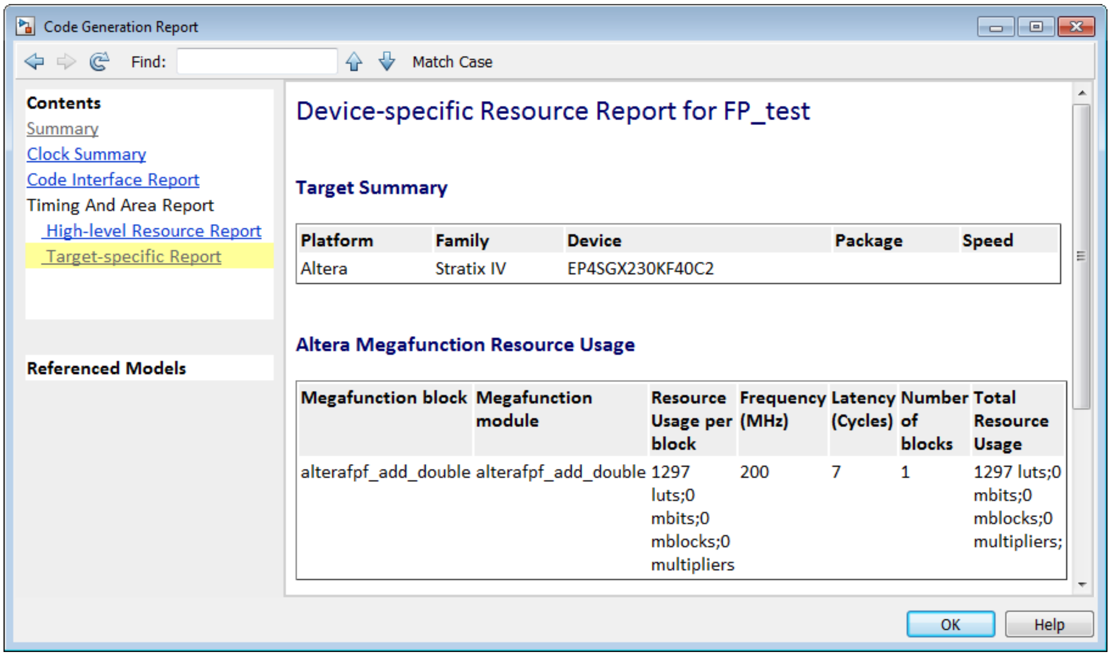
```
[+] 32x32-bit Adder : 10
[+] 32x32-bit Subtractor : 4
```

Registers (28)

```
8-bit Register : 4
32-bit Register : 24
```

Target-Specific Report

When you map the Simulink model to a floating-point target library, the Target-Specific report shows the resource utilization.



Device-specific Resource Report for FP_test

Target Summary

Platform	Family	Device	Package	Speed
Altera	Stratix IV	EP4SGX230KF40C2		

Altera Megafunction Resource Usage

Megafunction block	Megafunction module	Resource Usage per block	Frequency (MHz)	Latency (Cycles)	Number of blocks	Total Resource Usage
alterafpf_add_double	alterafpf_add_double	1297 luts;0 mbits;0 mblocks;0 multipliers	200	7	1	1297 luts;0 mbits;0 mblocks;0 multipliers;

See also “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 22-26.

Optimization Report

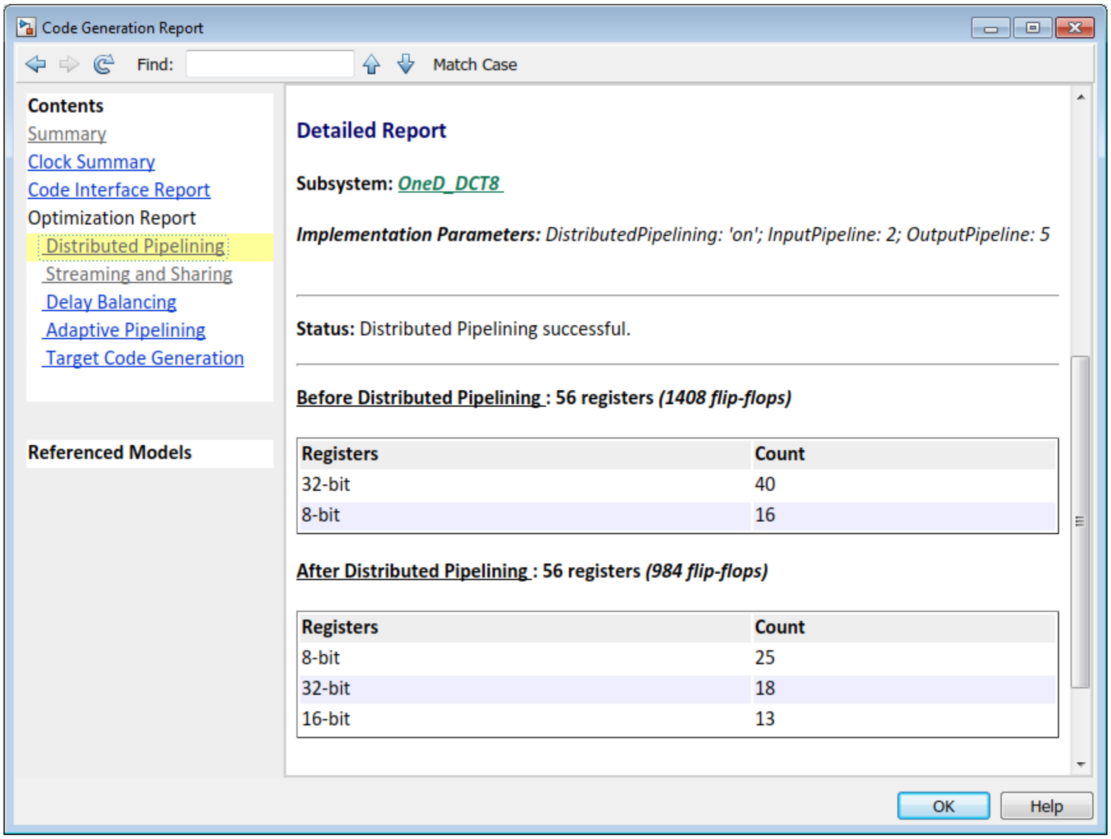
When you select **Generate optimization report**, HDL Coder adds an Optimization Report section, with three subsections:

- **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.

Following are graphical representations of these report subsections.

Distributed Pipelining

A typical Distributed Pipelining report looks like this:



Detailed Report

Subsystem: OneD_DCT8

Implementation Parameters: *DistributedPipelining: 'on'; InputPipeline: 2; OutputPipeline: 5*

Status: Distributed Pipelining successful.

Before Distributed Pipelining : 56 registers *(1408 flip-flops)*

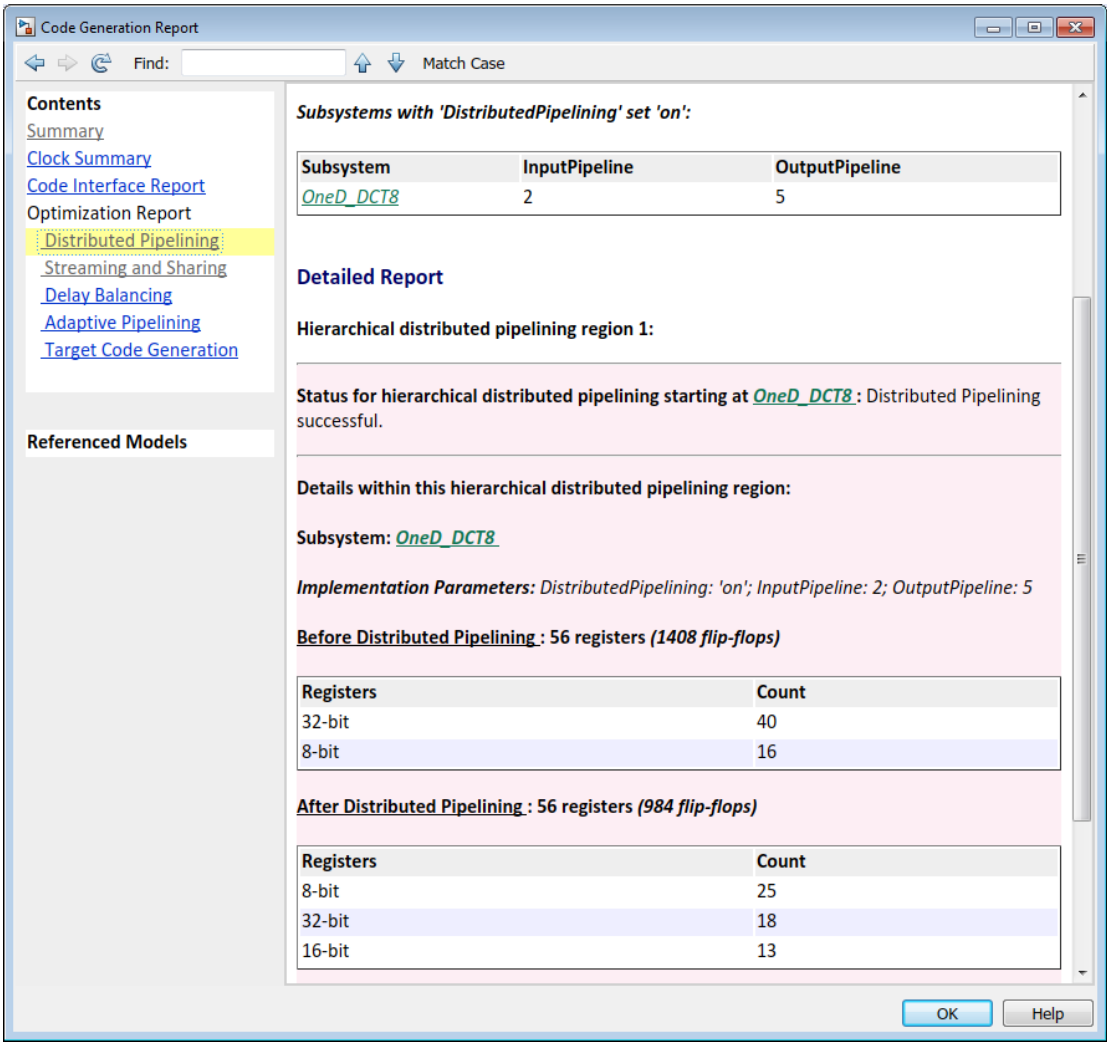
Registers	Count
32-bit	40
8-bit	16

After Distributed Pipelining : 56 registers *(984 flip-flops)*

Registers	Count
8-bit	25
32-bit	18
16-bit	13

Hierarchical Distributed Pipelining in the Optimization Report

If `HierarchicalDistPipelining` is `on`, the Optimization Report uses colored sections to distinguish between different regions where HDL Coder applies hierarchical distributed pipelining.



Code Generation Report

Find: Match Case

Contents
Summary
Clock Summary
Code Interface Report
Optimization Report
Distributed Pipelining
Streaming and Sharing
Delay Balancing
Adaptive Pipelining
Target Code Generation

Referenced Models

Subsystems with 'DistributedPipelining' set 'on':

Subsystem	InputPipeline	OutputPipeline
OneD_DCT8	2	5

Detailed Report

Hierarchical distributed pipelining region 1:

Status for hierarchical distributed pipelining starting at *OneD_DCT8* : Distributed Pipelining successful.

Details within this hierarchical distributed pipelining region:

Subsystem: *OneD_DCT8*

Implementation Parameters: *DistributedPipelining: 'on'; InputPipeline: 2; OutputPipeline: 5*

Before Distributed Pipelining : 56 registers *(1408 flip-flops)*

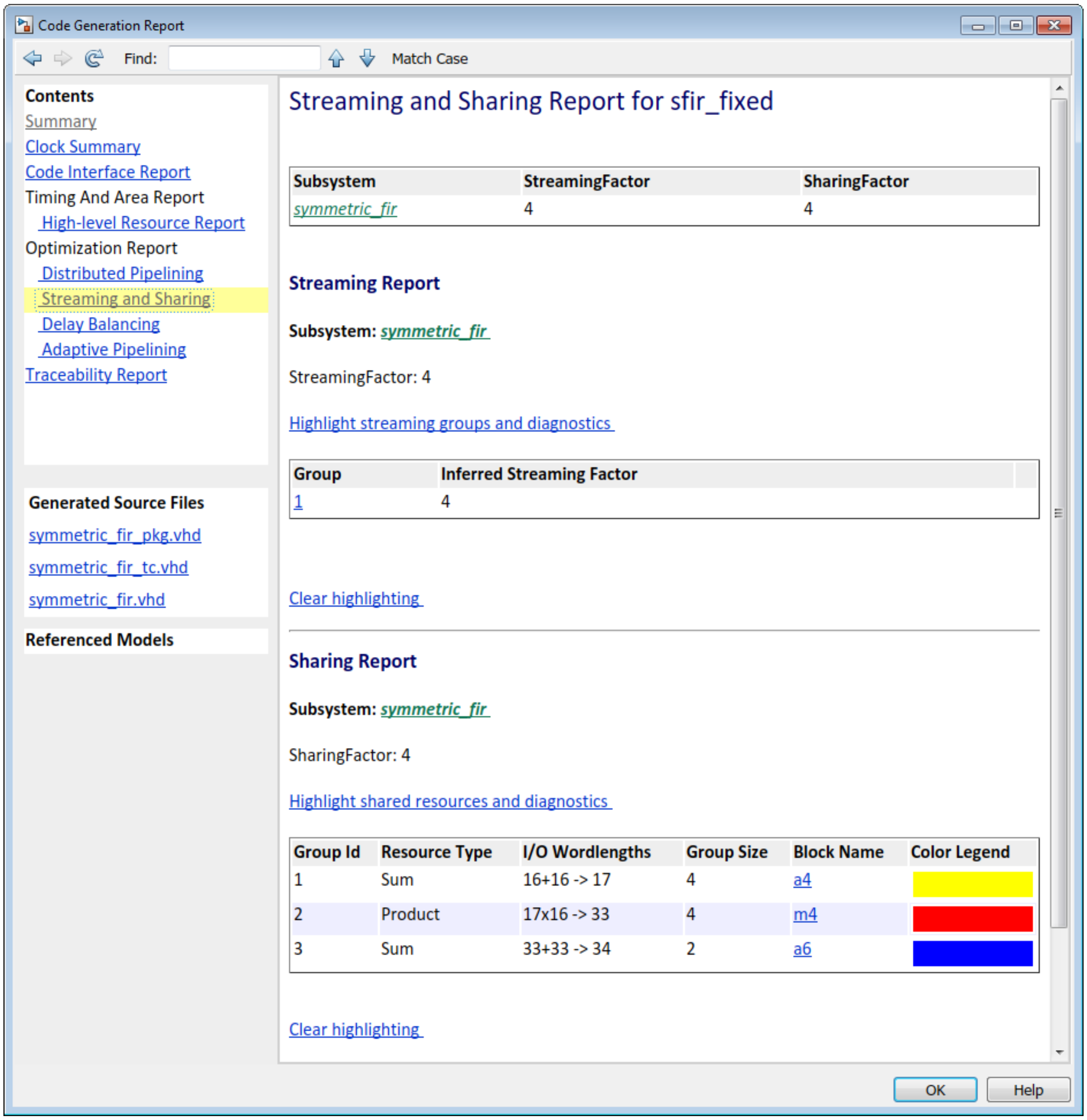
Registers	Count
32-bit	40
8-bit	16

After Distributed Pipelining : 56 registers *(984 flip-flops)*

Registers	Count
8-bit	25
32-bit	18
16-bit	13

OK Help

Streaming and Sharing Report


Code Generation Report

Find: Match Case

Contents
Summary
Clock Summary
Code Interface Report
Timing And Area Report
High-level Resource Report
Optimization Report
Distributed Pipelining
Streaming and Sharing
Delay Balancing
Adaptive Pipelining
Traceability Report

Generated Source Files
symmetric_fir_pkg.vhd
symmetric_fir_tc.vhd
symmetric_fir.vhd

Referenced Models

Streaming and Sharing Report for sfir_fixed

Subsystem	StreamingFactor	SharingFactor
symmetric_fir	4	4

Streaming Report

Subsystem: symmetric_fir

StreamingFactor: 4

Highlight streaming groups and diagnostics

Group	Inferred Streaming Factor
1	4

Clear highlighting

Sharing Report

Subsystem: symmetric_fir

SharingFactor: 4

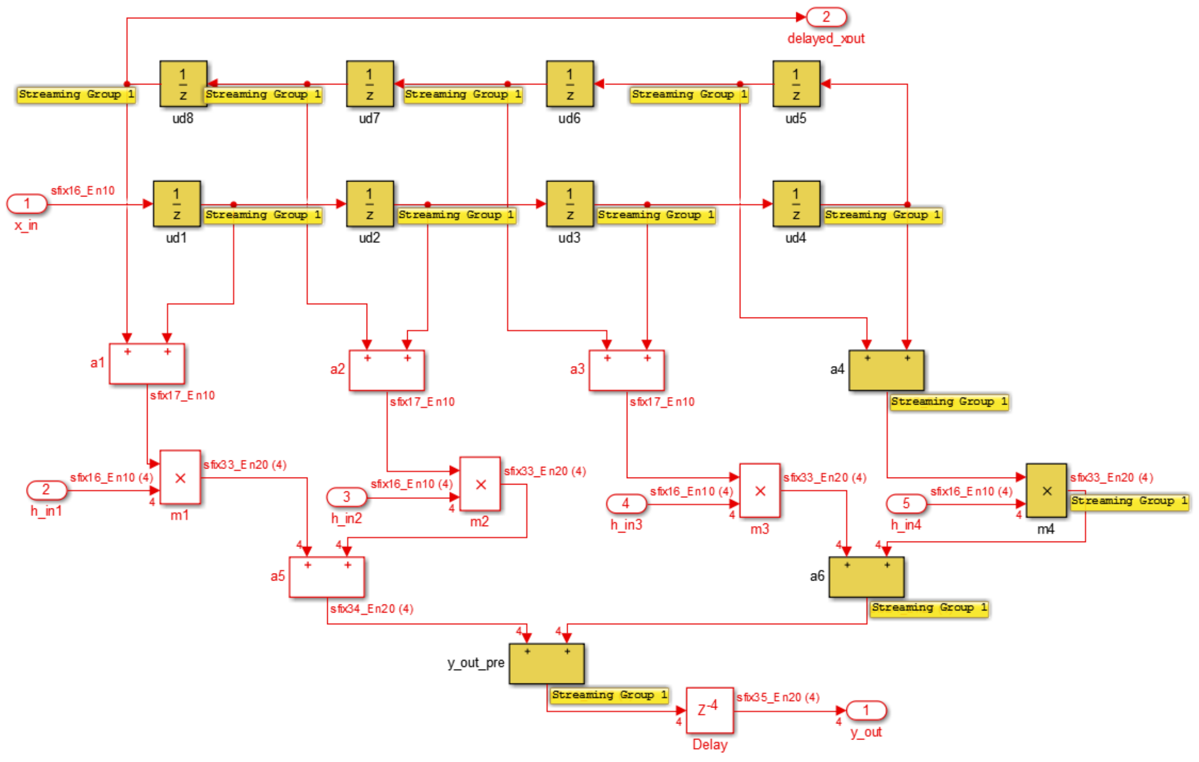
Highlight shared resources and diagnostics

Group Id	Resource Type	I/O Wordlengths	Group Size	Block Name	Color Legend
1	Sum	16+16 -> 17	4	a4	🟨
2	Product	17x16 -> 33	4	m4	🟥
3	Sum	33+33 -> 34	2	a6	🟦

Clear highlighting

Streaming Report

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.



The streaming report shows a table that specifies:

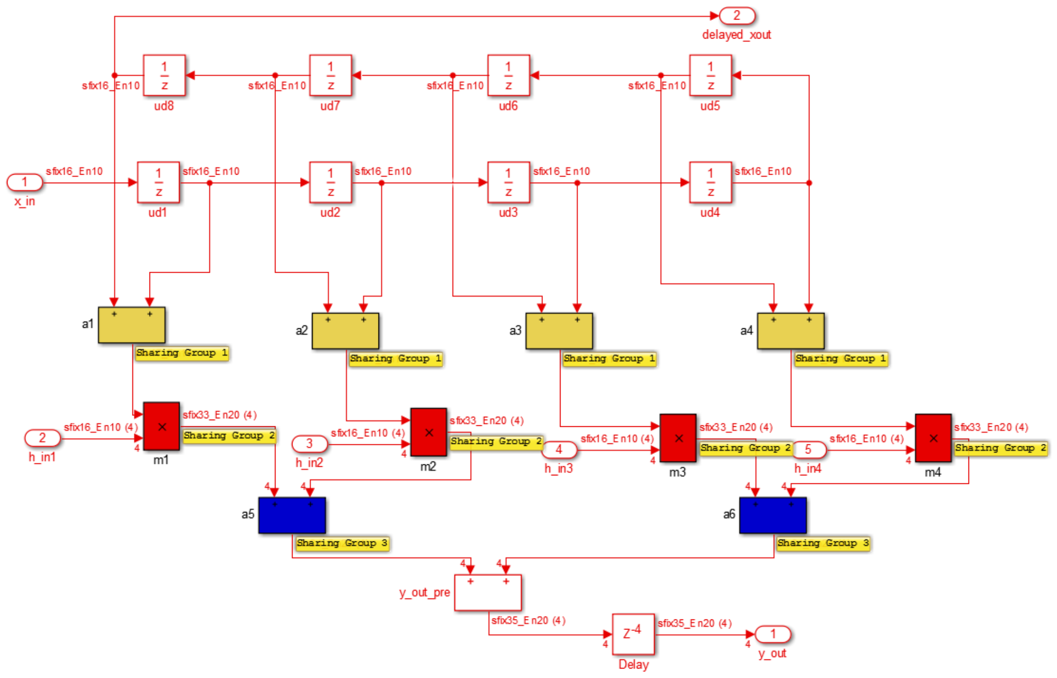
- **Group**: A unique group ID for a group of Simulink blocks that belong to a streaming group.
- **Inferred Streaming Factor**: Streaming factor inferred by HDL Coder with the **Streaming Factor** that you specify in the HDL Block Properties.

See also “Streaming” on page 15-7.

Sharing Report

To see groups of blocks that share resources in your Simulink model and in the generated model, click the **Highlight shared resources and diagnostics** link in the Streaming and Sharing report.

Note: If a MATLAB Function block in your Simulink model successfully shared resources, then the **Highlight shared resources and diagnostics** link cannot highlight the shared resources in the block.



The sharing report shows diagnostic messages and offending blocks that cause resource sharing to fail. The report also shows a table that specifies:

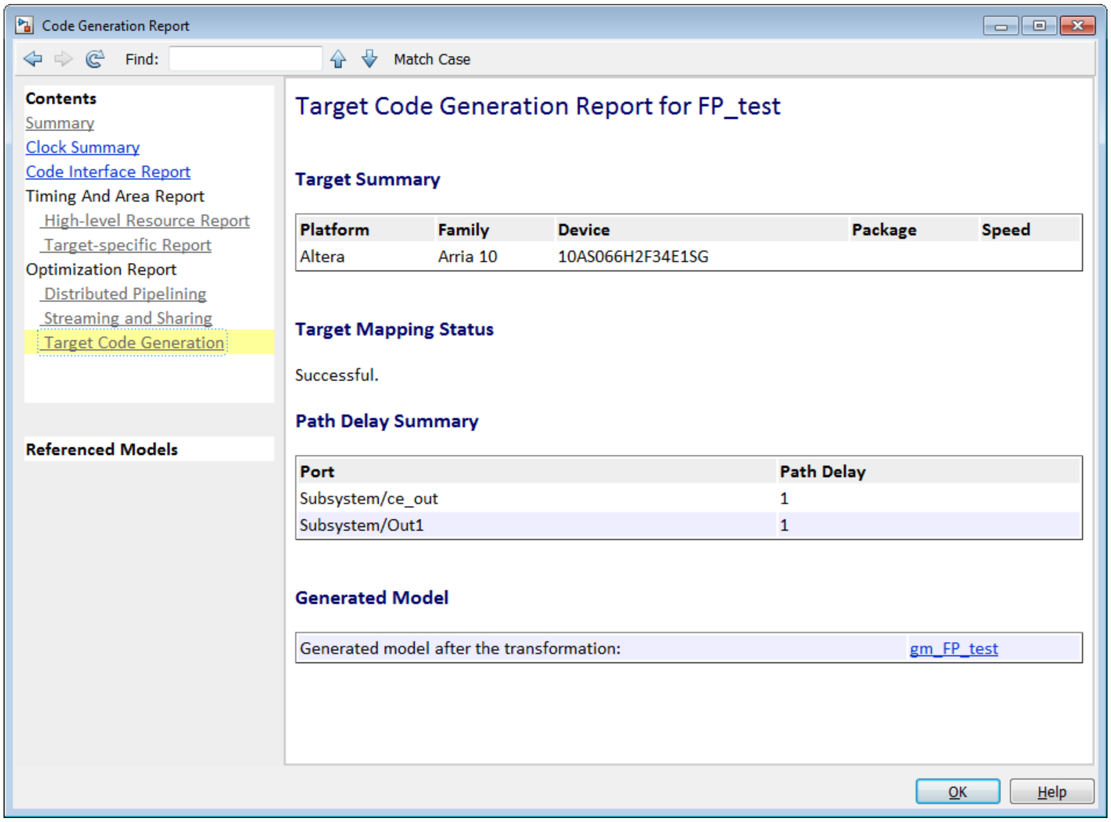
- **Group Id**: A unique ID for a group of similar Simulink blocks, such as add or product blocks, that share resources.

- **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.

See also “Resource Sharing” on page 15-20.

Target Code Generation

When you map the design to a floating-point target library, the Target Code Generation report shows the status of mapping.



Target Code Generation Report for FP_test

Target Summary

Platform	Family	Device	Package	Speed
Altera	Arria 10	10AS066H2F34E1SG		

Target Mapping Status

Successful.

Path Delay Summary

Port	Path Delay
Subsystem/ce_out	1
Subsystem/Out1	1

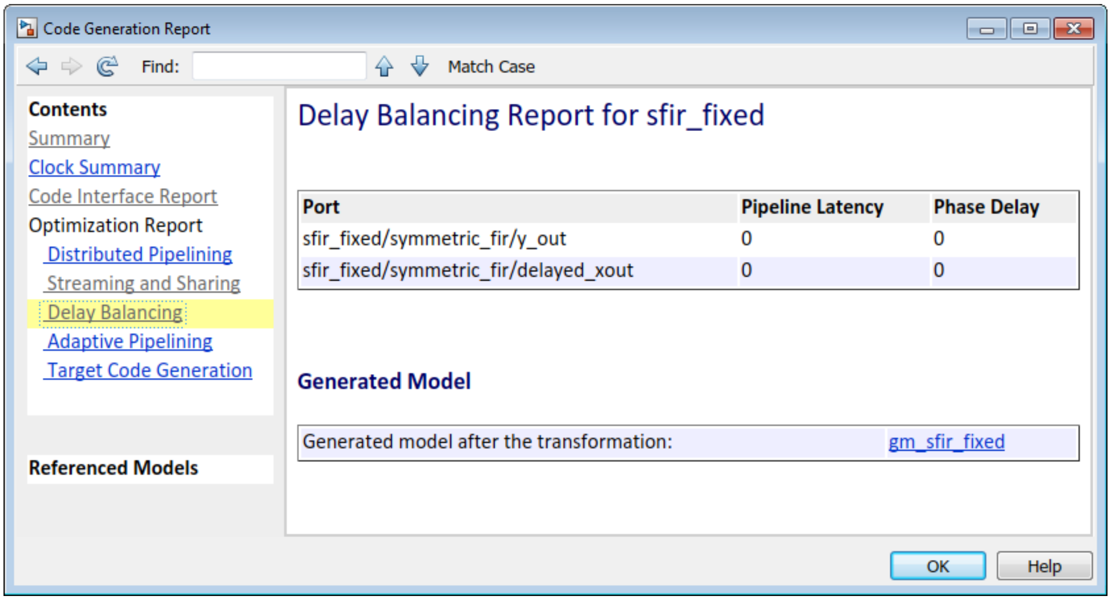
Generated Model

Generated model after the transformation:	gm_FP_test

See also “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 22-26.

Delay Balancing Report

The Delay Balancing Report shows the pipeline delay and phase delay at the output ports and the number of pipelines added at the output ports to match the delays.



Delay Balancing Report for sfir_fixed

Port	Pipeline Latency	Phase Delay
sfir_fixed/symmetric_fir/y_out	0	0
sfir_fixed/symmetric_fir/delayed_xout	0	0

Generated Model

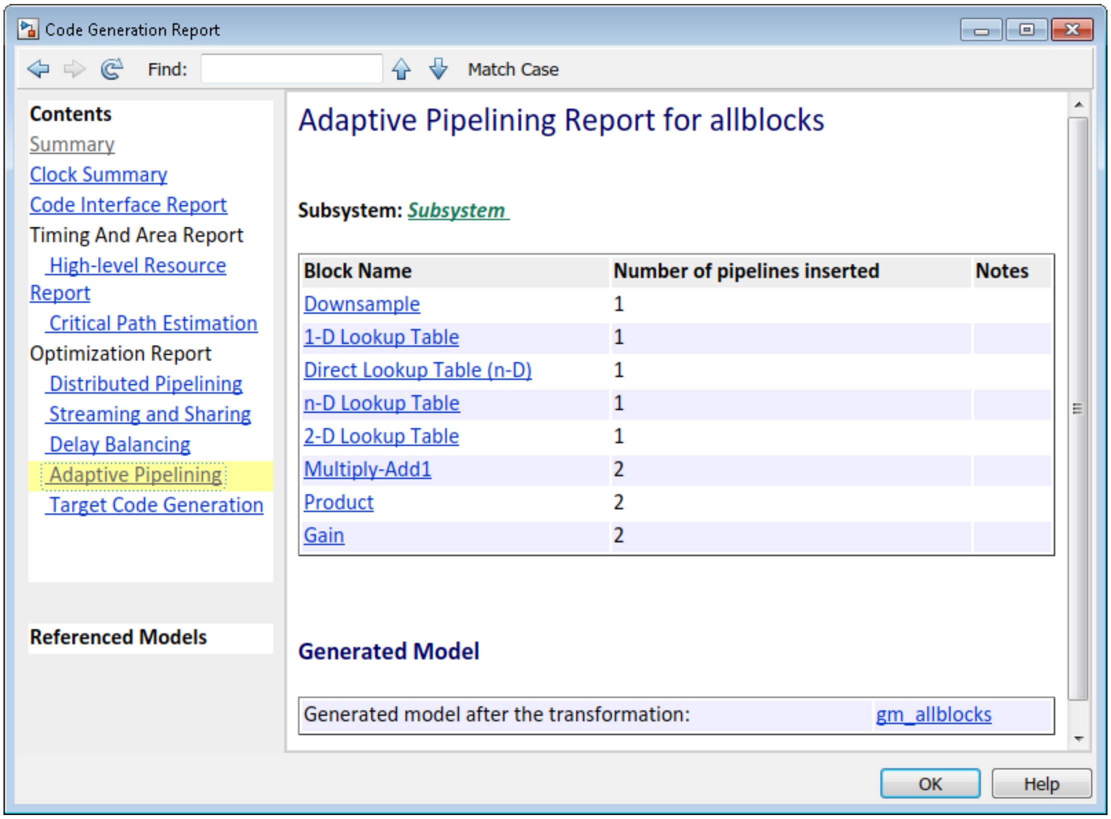
Generated model after the transformation:	gm_sfir_fixed

If delay balancing fails, the report mentions the criterion that was violated and displays the link to any block or subsystem that caused delay balancing to fail.

See also “Delay Balancing” on page 15-26.

Adaptive Pipelining Report

The adaptive pipelining 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.



Adaptive Pipelining Report for allblocks

Subsystem: *Subsystem*

Block Name	Number of pipelines inserted	Notes
Downsample	1	
1-D Lookup Table	1	
Direct Lookup Table (n-D)	1	
n-D Lookup Table	1	
2-D Lookup Table	1	
Multiply-Add1	2	
Product	2	
Gain	2	

Generated Model

Generated model after the transformation:	gm_allblocks

If adaptive pipelining fails, the report displays the criterion that caused adaptive pipelining to fail. For more information, see “Adaptive Pipelining” on page 15-71.

Related Examples

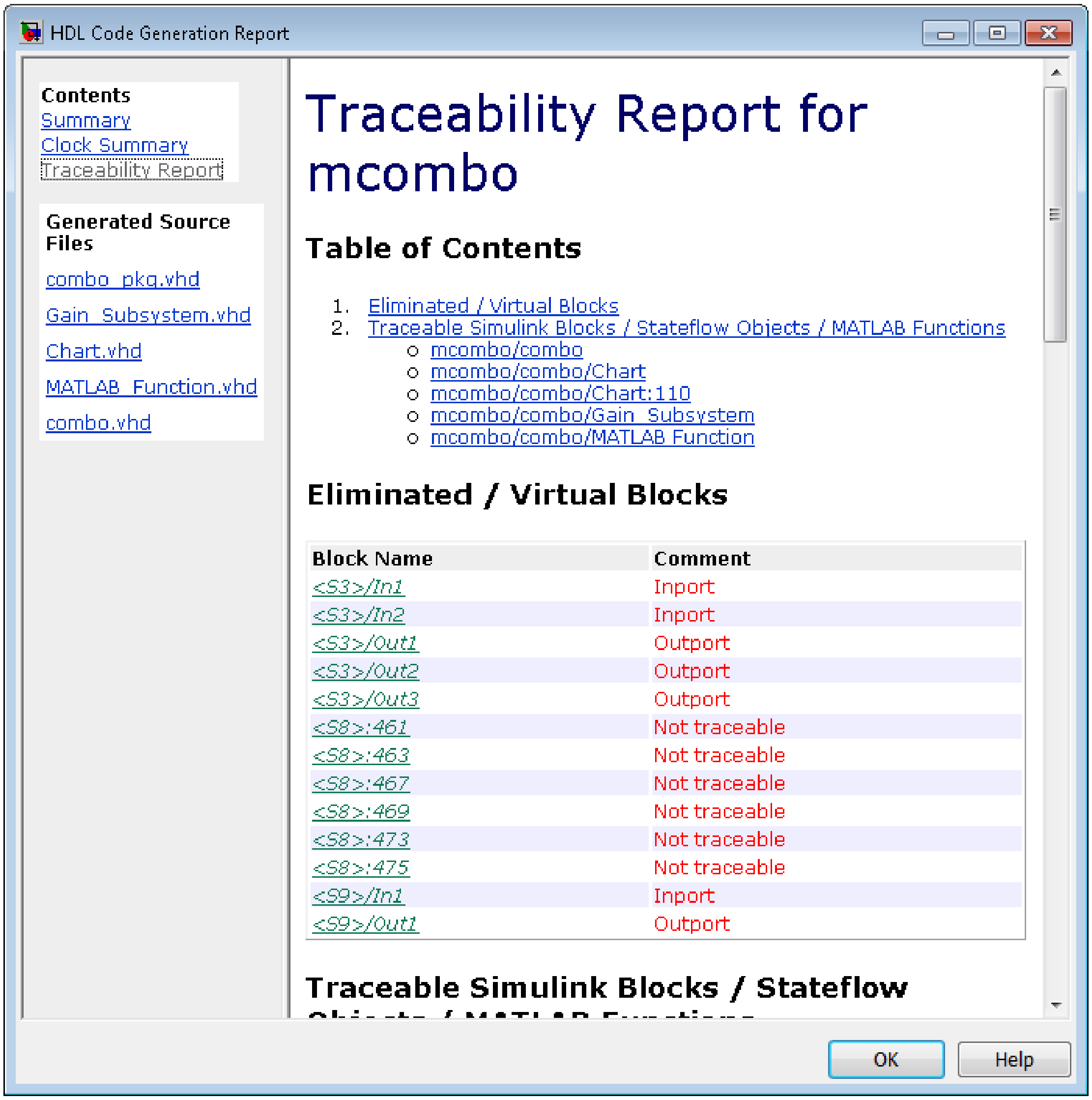
- “Find Feedback Loops” on page 15-33

Traceability Report

In this section...
"Traceability Report Overview" on page 16-18
"Generating a Traceability Report from Configuration Parameters" on page 16-22
"Generating a Traceability Report from the Command Line" on page 16-25
"Keeping the Report Current" on page 16-27
"Tracing from Code to Model" on page 16-27
"Tracing from Model to Code" on page 16-29
"Mapping Model Elements to Code Using the Traceability Report" on page 16-32
"Traceability Report Limitations" on page 16-33

Traceability Report Overview

Even a relatively small model can generate hundreds of lines of HDL code. The HDL Coder software provides the traceability report section to help you navigate more easily between the generated code and your source model. When you enable traceability, HDL Coder creates and displays an HTML code generation report. You can generate reports from the Configuration Parameters dialog box or the command line. A typical traceability report looks something like this:



Contents
Summary
Clock Summary
Traceability Report

Generated Source Files

combo_pkg.vhd
Gain_Subsystem.vhd
Chart.vhd
MATLAB_Function.vhd
combo.vhd

Traceability Report for mcombo

Table of Contents

1. Eliminated / Virtual Blocks
2. Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions
   - mcombo/combo
   - mcombo/combo/Chart
   - mcombo/combo/Chart:110
   - mcombo/combo/Gain_Subsystem
   - mcombo/combo/MATLAB Function

Eliminated / Virtual Blocks

Block Name	Comment
<S3>/In1	Inport
<S3>/In2	Inport
<S3>/Out1	Outport
<S3>/Out2	Outport
<S3>/Out3	Outport
<S8>:461	Not traceable
<S8>:463	Not traceable
<S8>:467	Not traceable
<S8>:469	Not traceable
<S8>:473	Not traceable
<S8>:475	Not traceable
<S9>/In1	Inport
<S9>/Out1	Outport

Traceable Simulink Blocks / Stateflow

The traceability report has several subsections:

- The **Traceability Report** lists **Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions**, providing a complete mapping between model elements and code. The **Eliminated / Virtual Blocks** section of the report accounts for blocks that are untraceable.

- The **Generated Source Files** table contains hyperlinks that let you view generated HDL code in a MATLAB web browser window. This view of the code includes hyperlinks that let you view the blocks or subsystems from which the code was generated. You can click the names of source code files generated from your model to view their contents in a MATLAB web browser window. The report supports two types of linkage between the model and generated code:

  - *Code-to-model* hyperlinks within the displayed source code let you view the blocks or subsystems from which the code was generated. Click the hyperlinks to view the relevant blocks or subsystems in a Simulink model window.
  - *Model-to-code* linkage lets you view the generated code for any block in the model. To highlight a block's generated code in the HTML report, right-click the block and select **HDL Code** > **Navigate to Code**.

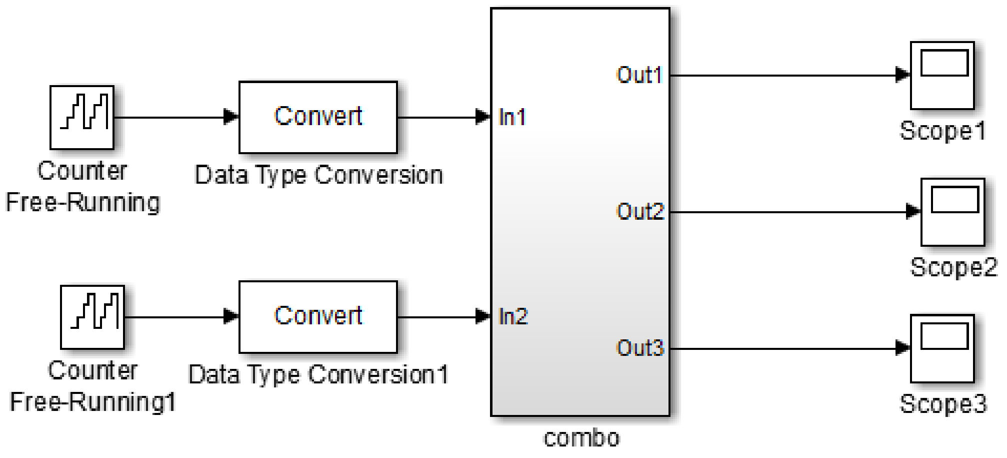
Note: If your model includes blocks that have requirements comments, you can also render the comments as hyperlinked comments within the HTML code generation report. See “Requirements Comments and Hyperlinks” on page 16-38 for more information.

In the following sections, the `mcombo` example model illustrates stages in the workflow for generating code generation reports from the Configuration Parameters dialog box and from the command line.

To open the model, enter:

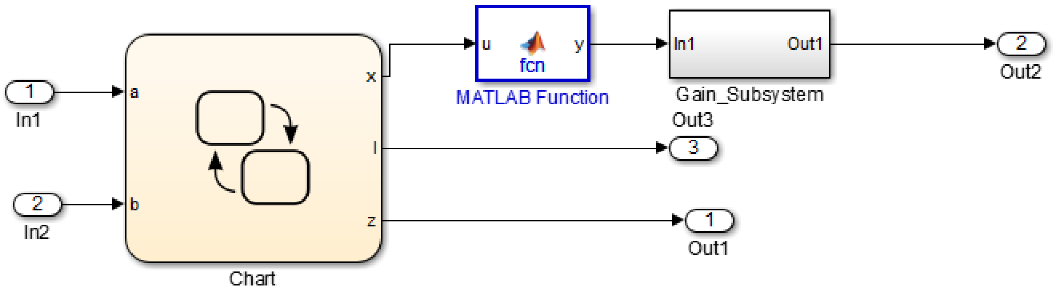
```
mcombo
```

The root-level `mcombo` model appears as follows:



Copyright 2008-2010 The MathWorks, Inc.

HDL Coder supports report generation for models, subsystems, blocks, Stateflow charts, and MATLAB Function blocks. This example uses the `combo` subsystem, shown in the following figure. The `combo` subsystem includes a Stateflow chart, a MATLAB Function block, and a subsystem.

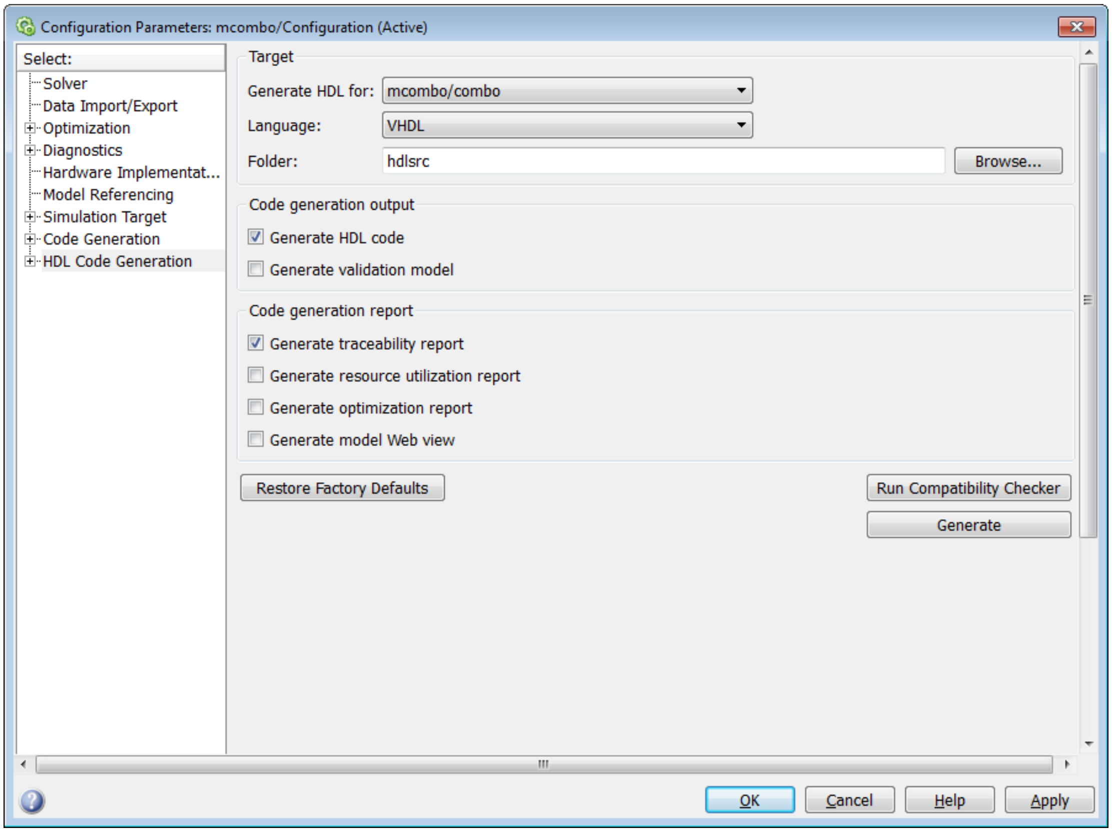


Generating a Traceability Report from Configuration Parameters

To generate a HDL Coder code generation report from the Configuration Parameters dialog box:

1. Verify that the model is open.
2. Open the Configuration Parameters dialog box and navigate to the **HDL Code Generation** pane.
3. To enable report generation, select **Generate traceability report**.

   If your model includes blocks that have requirements comments, you can also select **Include requirements in block comments** in the **HDL Code Generation** > **Global Settings** > **Coding style** pane to render the comments as hyperlinked comments in the HTML code generation report. See “Requirements Comments and Hyperlinks” on page 16-38 for more information.
4. Verify that **Generate HDL for** specifies the correct DUT for HDL code generation. You can generate reports for the root-level model or for subsystems, blocks, Stateflow charts, or MATLAB Function blocks.
5. Click **Apply**.



Target

Generate HDL for: mcombo/combo

Language: VHDL

Folder: hdlsrc

Code generation output

- [✓] Generate HDL code
- [ ] Generate validation model

Code generation report

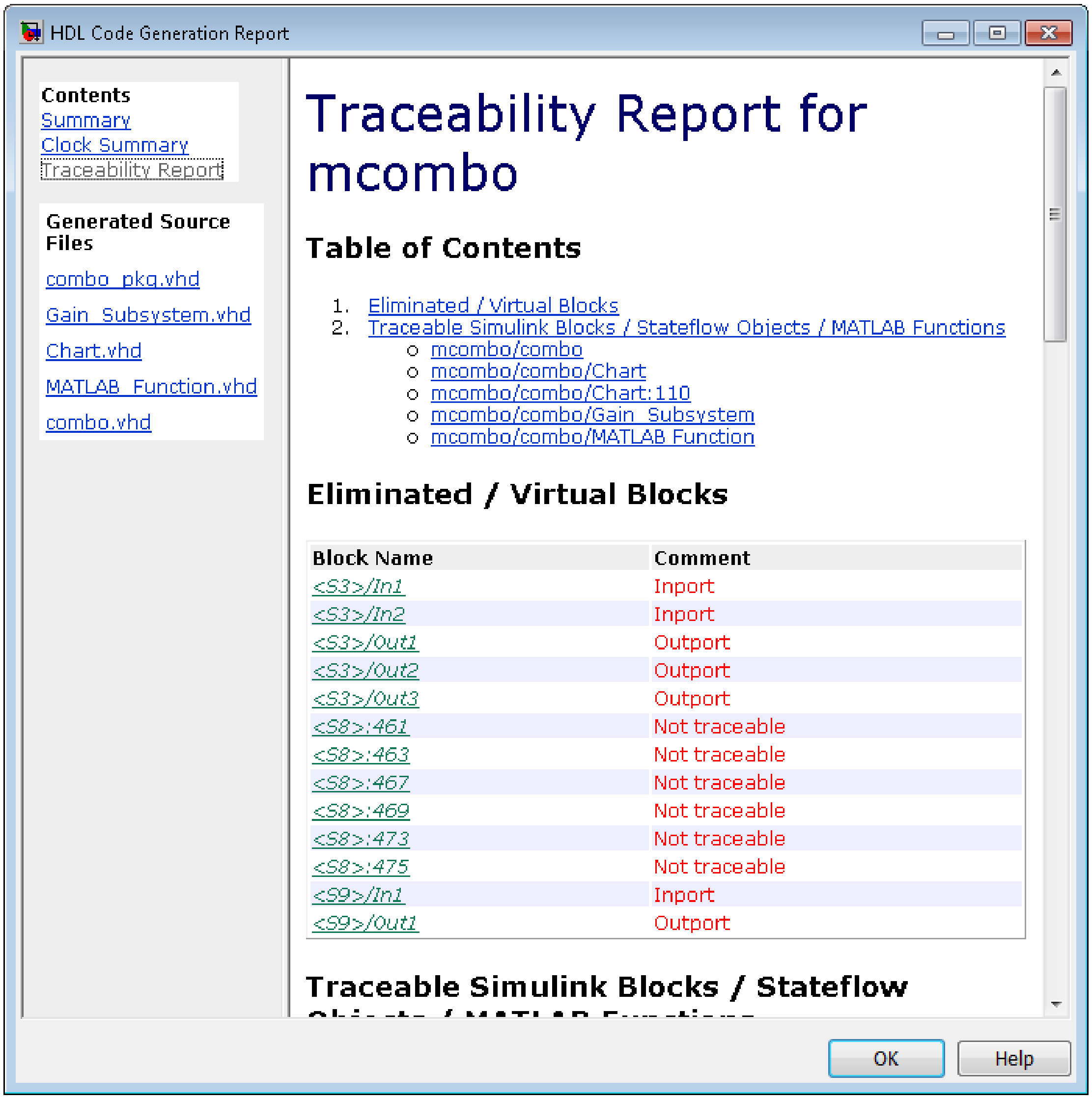
- [✓] Generate traceability report
- [ ] Generate resource utilization report
- [ ] Generate optimization report
- [ ] Generate model Web view

6 Click **Generate** to initiate code and report generation.

When you select **Generate traceability report**, HDL Coder generates HTML report files as part of the code generation process. Report file generation is the final phase of that process. As code generation proceeds, the coder displays progress messages. The process completes with messages similar to the following:

```
### Generating HTML files for traceability in slprj\hdl\mcombo\html directory ...

### HDL Code Generation Complete.
```

When code generation is complete, the HTML report appears in a new window:



Contents
Summary
Clock Summary
Traceability Report

Generated Source Files

combo_pkg.vhd
Gain_Subsystem.vhd
Chart.vhd
MATLAB_Function.vhd
combo.vhd

Traceability Report for mcombo

Table of Contents

1. Eliminated / Virtual Blocks
2. Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions
   - mcombo/combo
   - mcombo/combo/Chart
   - mcombo/combo/Chart:110
   - mcombo/combo/Gain_Subsystem
   - mcombo/combo/MATLAB Function

Eliminated / Virtual Blocks

Block Name	Comment
<S3>/In1	Inport
<S3>/In2	Inport
<S3>/Out1	Outport
<S3>/Out2	Outport
<S3>/Out3	Outport
<S8>:461	Not traceable
<S8>:463	Not traceable
<S8>:467	Not traceable
<S8>:469	Not traceable
<S8>:473	Not traceable
<S8>:475	Not traceable
<S9>/In1	Inport
<S9>/Out1	Outport

Traceable Simulink Blocks / Stateflow

7 To view the different report sections or view the generated code files, click the hyperlinks in the **Contents** pane of the report window.

Tip: HDL Coder writes the code generation report files to a folder in the `hdlsrc\html\` folder of the build folder. The top-level HTML report file is named `*system*_codegen_rpt.html`, where *`system`* is the name of the model, subsystem, or other component selected for code generation. However, because the coder automatically opens this file after report generation, you do not need to access the HTML files directly. Instead, navigate the report using the links in the top-level window.

For more information on using the report you generate for tracing, see:

- "Tracing from Code to Model" on page 16-27
- "Tracing from Model to Code" on page 16-29
- "Mapping Model Elements to Code Using the Traceability Report" on page 16-32

Generating a Traceability Report from the Command Line

To generate a HDL Coder code generation report from the command line:

1. Open your model by entering:

   ```
   open_system('model_name');
   ```
2. Specify the desired device under test (DUT) for code generation by entering:

   ```
   gcb = 'path_to_DUT';
   ```

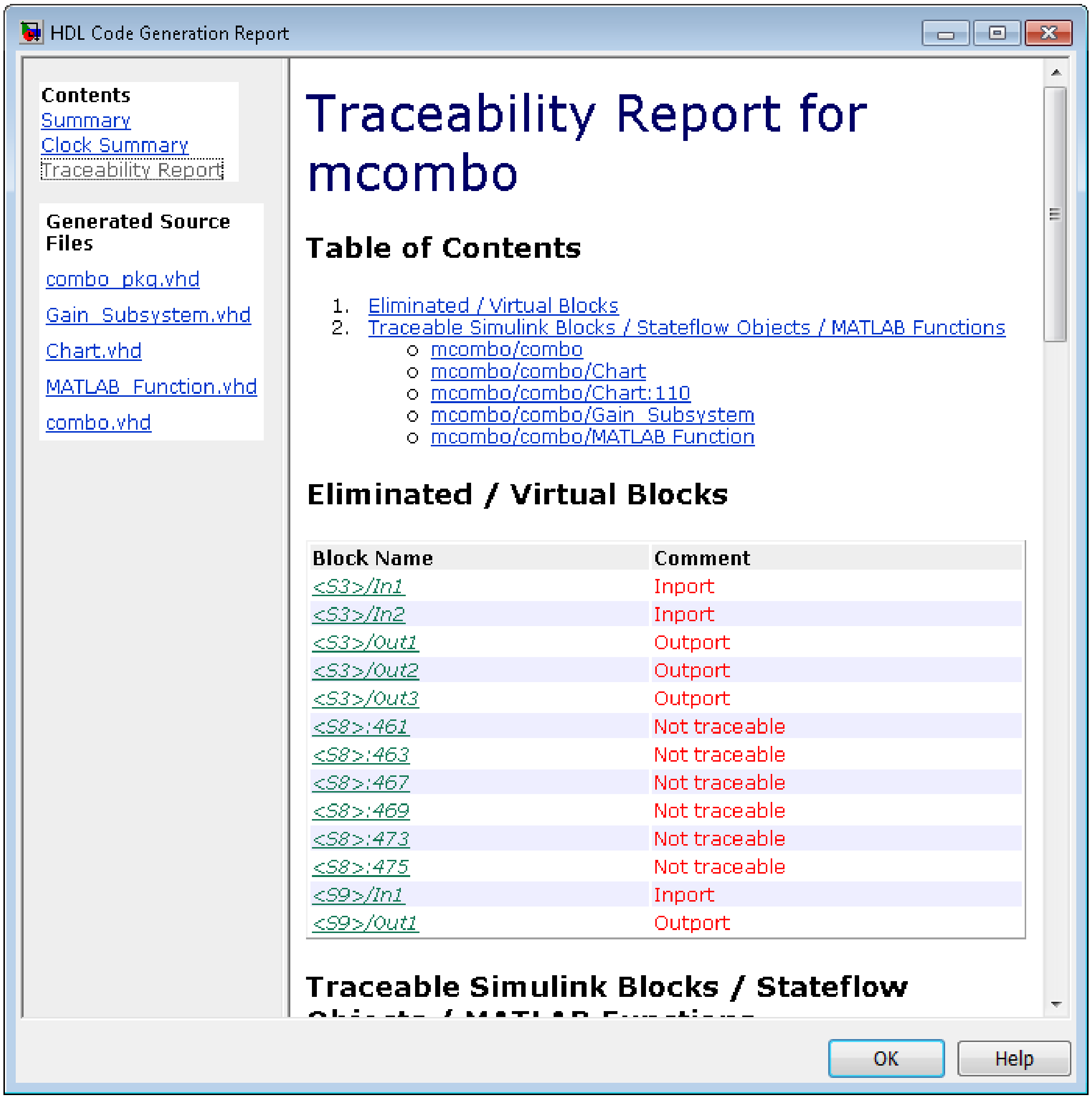
   You can generate reports for the root-level model or for subsystems, blocks, Stateflow charts, or MATLAB Function blocks. If you do not specify a subsystem, block, Stateflow chart, or MATLAB Function block, HDL Coder generates a report for the top-level model.
3. Enable the `makehdl` property `Traceability` by entering:

   ```
   makehdl(gcb,'Traceability','on');
   ```

   When you enable traceability, HDL Coder generates HTML report files as part of the code generation process. Report file generation is the final phase of that process. As code generation proceeds, the coder displays progress messages. The process completes with messages similar to the following:

   ```
   ### Generating HTML files for traceability in slprj\hdl\mcombo\html directory ...

   ### HDL Code Generation Complete.
   ```

When code generation is complete, the HTML report appears in a new window:



Contents
Summary
Clock Summary
Traceability Report

Generated Source Files
combo_pkg.vhd
Gain_Subsystem.vhd
Chart.vhd
MATLAB_Function.vhd
combo.vhd

Traceability Report for mcombo

Table of Contents

1. Eliminated / Virtual Blocks
2. Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions
   - mcombo/combo
   - mcombo/combo/Chart
   - mcombo/combo/Chart:110
   - mcombo/combo/Gain_Subsystem
   - mcombo/combo/MATLAB Function

Eliminated / Virtual Blocks

Block Name	Comment
<S3>/In1	Inport
<S3>/In2	Inport
<S3>/Out1	Outport
<S3>/Out2	Outport
<S3>/Out3	Outport
<S8>:461	Not traceable
<S8>:463	Not traceable
<S8>:467	Not traceable
<S8>:469	Not traceable
<S8>:473	Not traceable
<S8>:475	Not traceable
<S9>/In1	Inport
<S9>/Out1	Outport

Traceable Simulink Blocks / Stateflow

4 To view the different report sections or view the generated code files, click the hyperlinks in the **Contents** pane of the report window.

Tip: HDL Coder writes the code generation report files to a folder in the `hdlsrc\html\` folder of the build folder. The top-level HTML report file is named `*system*_codegen_rpt.html`, where `*system*` is the name of the model, subsystem, or other component selected for code generation. However, because the coder automatically opens this file after report generation, you do not need to access the HTML files directly. Instead, navigate the report using the links in the top-level window.

For more information on using the report you generate for tracing, see:

- "Tracing from Code to Model" on page 16-27
- "Tracing from Model to Code" on page 16-29
- "Mapping Model Elements to Code Using the Traceability Report" on page 16-32

Keeping the Report Current

If you generate a code generation report for a model, and subsequently make changes to the model, the report might become invalid.

To keep your code generation report current, you should regenerate HDL code and the report after modifying the source model.

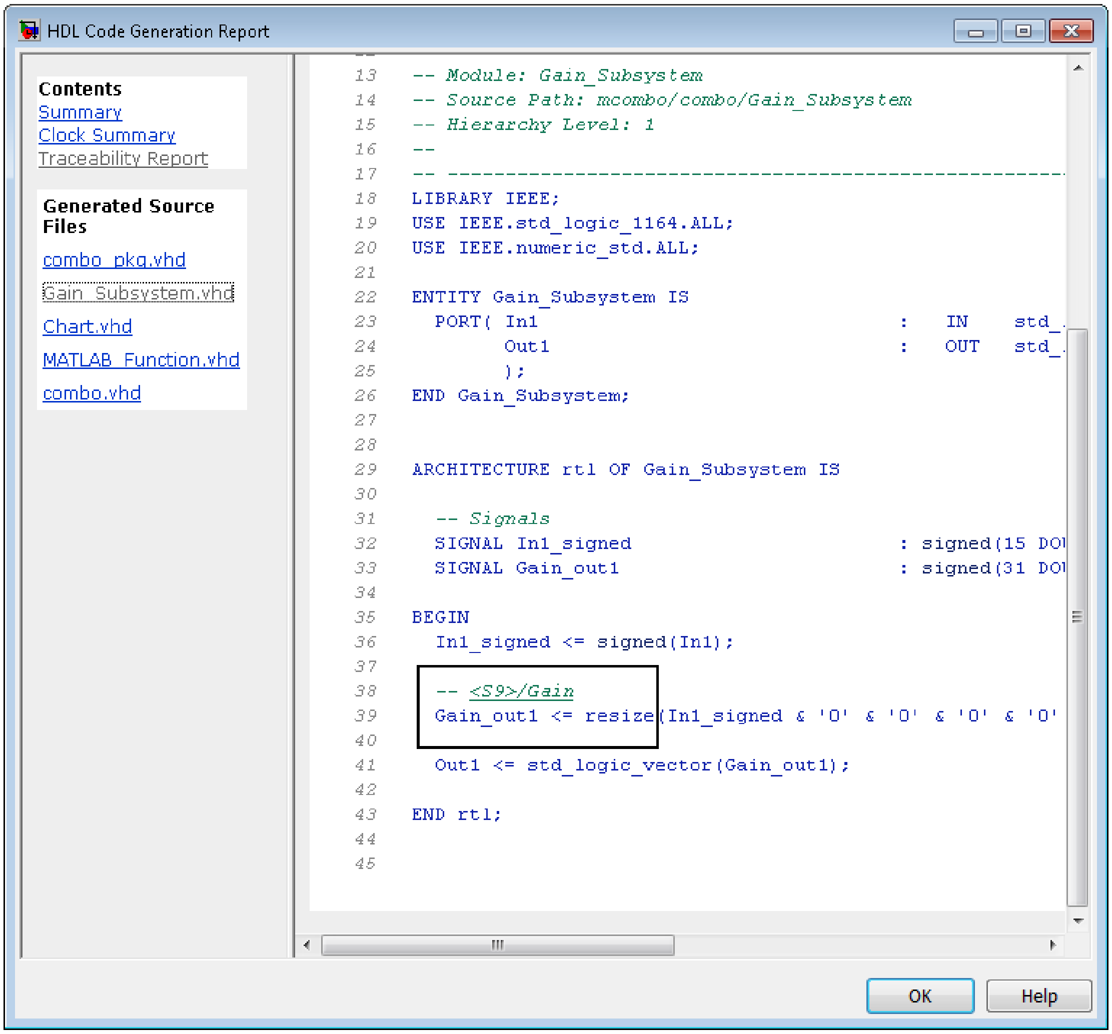
If you close and then reopen a model without making changes, the report remains valid.

Tracing from Code to Model

To trace from generated code to your model:

1. Generate code and open an HTML report for the desired DUT (see "Generating a Traceability Report from Configuration Parameters" on page 16-22 or "Generating a Traceability Report from the Command Line" on page 16-25).
2. In the left pane of the HTML report window, click the desired file name in the **Generated Source Files** table to view a source code file.

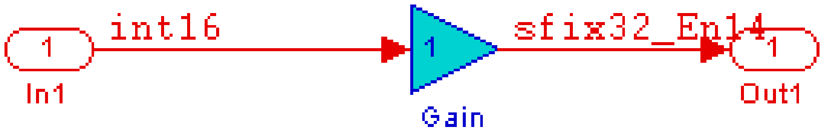
   The following figure shows a view of the source file `Gain_Subsystem.vhd`.



```
-- Module: Gain_Subsystem
-- Source Path: mcombo/combo/Gain_Subsystem
-- Hierarchy Level: 1
--
-- -------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY Gain_Subsystem IS
  PORT( In1                               :   IN    std_
        Out1                              :   OUT   std_
        );
END Gain_Subsystem;


ARCHITECTURE rtl OF Gain_Subsystem IS

  -- Signals
  SIGNAL In1_signed                       : signed(15 DO
  SIGNAL Gain_out1                        : signed(31 DO

BEGIN
  In1_signed <= signed(In1);

  -- <S9>/Gain
  Gain_out1 <= resize(In1_signed & '0' & '0' & '0' & '0'

  Out1 <= std_logic_vector(Gain_out1);

END rtl;
```

3 In the HTML report window, click a link to highlight the corresponding source block.

For example, in the HTML report shown in the previous figure, you could click the hyperlink for the Gain block (highlighted) to view that block in the model. Clicking the hyperlink locates and displays the corresponding block in the Simulink model window.



Tracing from Model to Code

Model-to-code traceability lets you select a component at any level of the model, and view the code references to that component in the HTML code generation report. You can select the following objects for tracing:

- Subsystem
- Simulink block
- MATLAB Function block
- Stateflow chart, or the following elements of a Stateflow chart:
  - State
  - Transition
  - Truth table
  - MATLAB function inside a chart

To trace a model component:

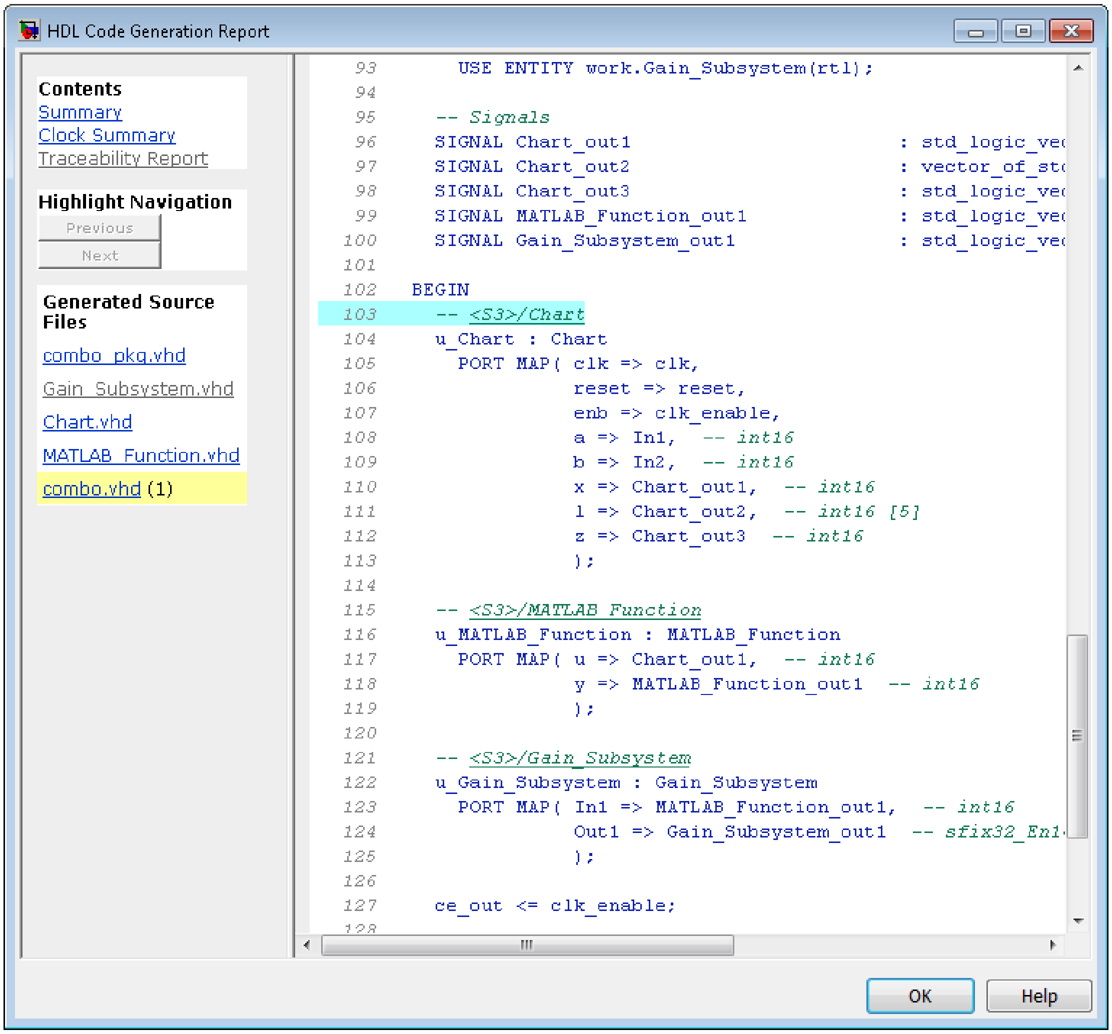
1 Generate code and open an HTML report for the desired DUT (see “Generating a Traceability Report from Configuration Parameters” on page 16-22 or “Generating a Traceability Report from the Command Line” on page 16-25).

Tip: If you have not generated code for the model, HDL Coder disables the **HDL Code** > **Navigate to Code** menu item.

2 In the model window, right-click the component and select **HDL Code** > **Navigate to Code**.

3 Selecting **Navigate to Code** activates the HTML code generation report.

   The following figure shows the result of tracing the Stateflow chart within the `combo` subsystem.



```
    USE ENTITY work.Gain_Subsystem(rtl);

  -- Signals
  SIGNAL Chart_out1                         : std_logic_ve
  SIGNAL Chart_out2                         : vector_of_st
  SIGNAL Chart_out3                         : std_logic_ve
  SIGNAL MATLAB_Function_out1               : std_logic_ve
  SIGNAL Gain_Subsystem_out1                : std_logic_ve

BEGIN
  -- <S3>/Chart
  u_Chart : Chart
    PORT MAP( clk => clk,
              reset => reset,
              enb => clk_enable,
              a => In1,  -- int16
              b => In2,  -- int16
              x => Chart_out1,  -- int16
              l => Chart_out2,  -- int16 [5]
              z => Chart_out3  -- int16
              );

  -- <S3>/MATLAB Function
  u_MATLAB_Function : MATLAB_Function
    PORT MAP( u => Chart_out1,  -- int16
              y => MATLAB_Function_out1  -- int16
              );

  -- <S3>/Gain_Subsystem
  u_Gain_Subsystem : Gain_Subsystem
    PORT MAP( In1 => MATLAB_Function_out1,  -- int16
              Out1 => Gain_Subsystem_out1  -- sfix32_En1
              );

  ce_out <= clk_enable;
```

In the right pane of the report, the highlighted tag `<S3>/Chart` indicates the beginning of the code generated code for the chart.

In the left pane of the report, the total number of highlighted lines of code (in this case, 1) appears next to the source file name `combo.vhd`.

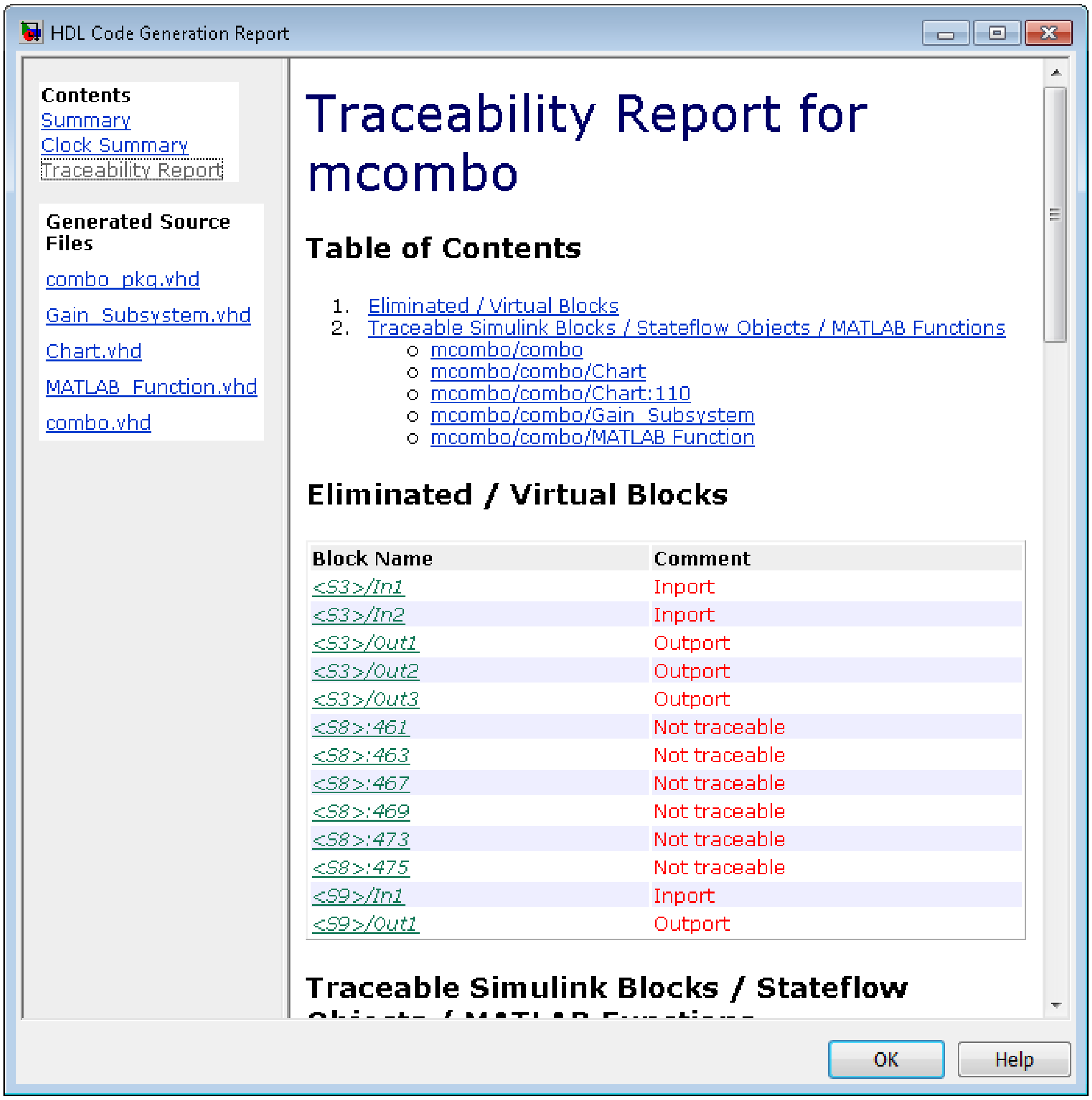
The left pane of the report also contains **Previous** and **Next** buttons. These buttons help you navigate through multiple instances of code generated for a selected component. In this example, there is only one instance, so the buttons are disabled.

Mapping Model Elements to Code Using the Traceability Report

The **Traceability Report** section of the report provides a complete mapping between model elements and code. The **Traceability Report** summarizes:

- **Eliminated / virtual blocks**: accounts for blocks that are untraceable because they are not included in generated code
- Traceable model elements, including:
  - **Traceable Simulink blocks**
  - **Traceable Stateflow objects**
  - **Traceable MATLAB functions**

The following figure shows the beginning of a traceability report generated for the `combo` subsystem of the `mcombo` model.



Contents
Summary
Clock Summary
Traceability Report

Generated Source Files

combo_pkg.vhd
Gain_Subsystem.vhd
Chart.vhd
MATLAB_Function.vhd
combo.vhd

Traceability Report for mcombo

Table of Contents

1. Eliminated / Virtual Blocks
2. Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions
   - mcombo/combo
   - mcombo/combo/Chart
   - mcombo/combo/Chart:110
   - mcombo/combo/Gain_Subsystem
   - mcombo/combo/MATLAB Function

Eliminated / Virtual Blocks

Block Name	Comment
<S3>/In1	Inport
<S3>/In2	Inport
<S3>/Out1	Outport
<S3>/Out2	Outport
<S3>/Out3	Outport
<S8>:461	Not traceable
<S8>:463	Not traceable
<S8>:467	Not traceable
<S8>:469	Not traceable
<S8>:473	Not traceable
<S8>:475	Not traceable
<S9>/In1	Inport
<S9>/Out1	Outport

Traceable Simulink Blocks / Stateflow

Traceability Report Limitations

The following limitations apply to HDL Coder HTML code generation reports:

- 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 ÿ (`char(255)`), code-to-model highlighting and model-to-code highlighting are disabled for that block.
- Some types of subsystems are not traceable from model to code at the subsystem block level:

  - Virtual subsystems
  - Masked subsystems
  - Nonvirtual subsystems for which code has been optimized away

  If you cannot trace a subsystem at the subsystem level, you might be able to trace individual blocks within the subsystem.

Web View of Model in Code Generation Report

In this section...
"About Model Web View" on page 16-35
"Generate HTML Code Generation Report with Model Web View" on page 16-35
"Model Web View Limitations" on page 16-37

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.

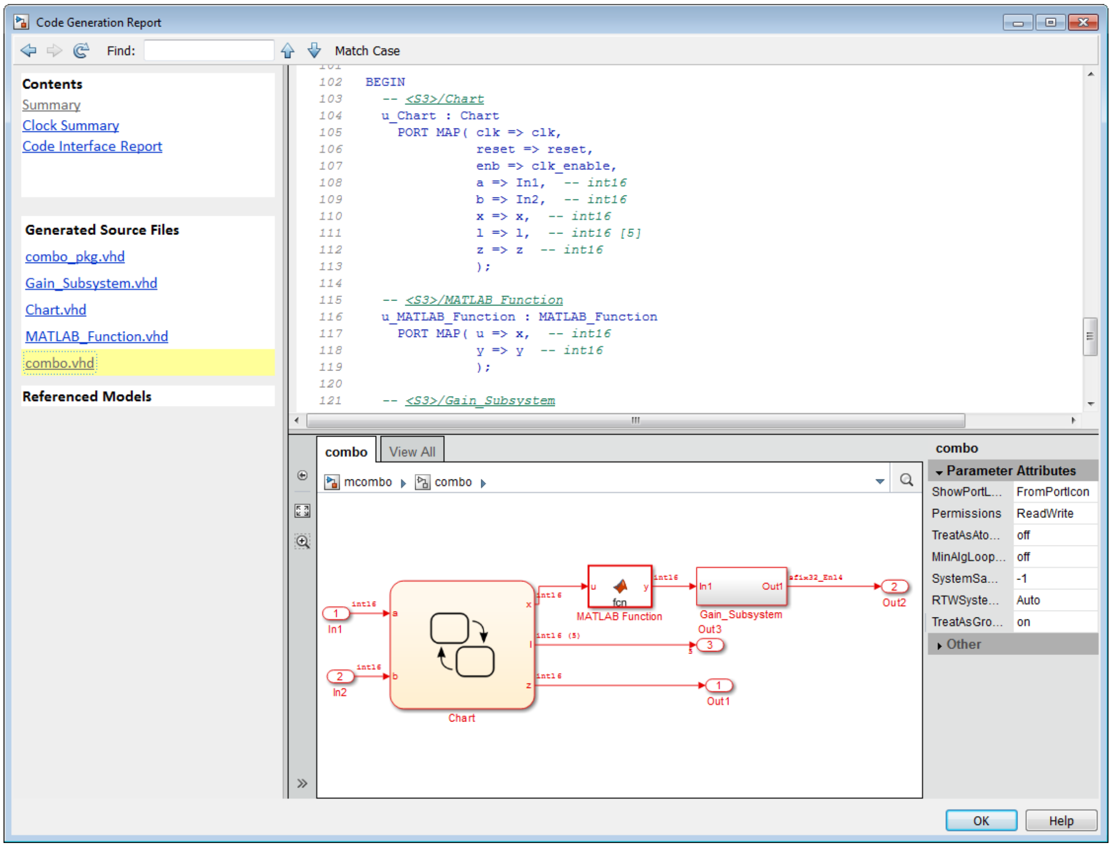
2 Open the **Configuration Parameters** dialog box or **Model Explorer** and navigate to the **HDL Code Generation** pane.

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.



```
BEGIN
  -- <S3>/Chart
  u_Chart : Chart
    PORT MAP( clk => clk,
              reset => reset,
              enb => clk_enable,
              a => In1,  -- int16
              b => In2,  -- int16
              x => x,  -- int16
              l => l,  -- int16 [5]
              z => z  -- int16
              );

  -- <S3>/MATLAB_Function
  u_MATLAB_Function : MATLAB_Function
    PORT MAP( u => x,  -- int16
              y => y  -- int16
              );

  -- <S3>/Gain_Subsystem
```

combo

Parameter Attributes	
ShowPortL...	FromPortIcon
Permissions	ReadWrite
TreatAsAto...	off
MinAlgLoop...	off
SystemSa...	-1
RTWSyste...	Auto
TreatAsGro...	on

Other

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" (Simulink Coder).
- For a subsystem build, the traceability hyperlinks of the root level inport 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...

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.

See "Describe Models Using Annotations" (Simulink) in the Simulink documentation for general information on annotations.

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 (Simulink) in the Simulink documentation for general information on the DocBlock.

Requirements Comments and Hyperlinks

You can assign requirement comments to blocks.

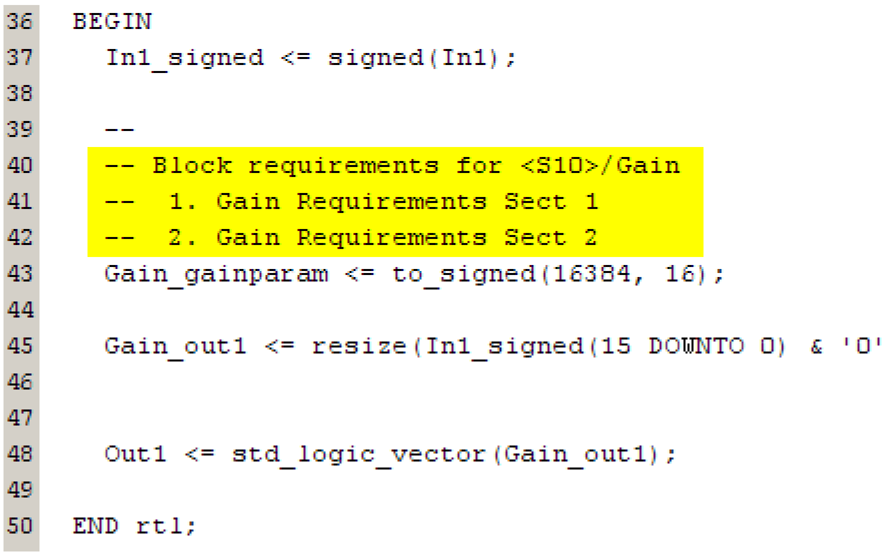
If your model includes requirements comments, you can choose to render the comments in one of the following formats:

- *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.

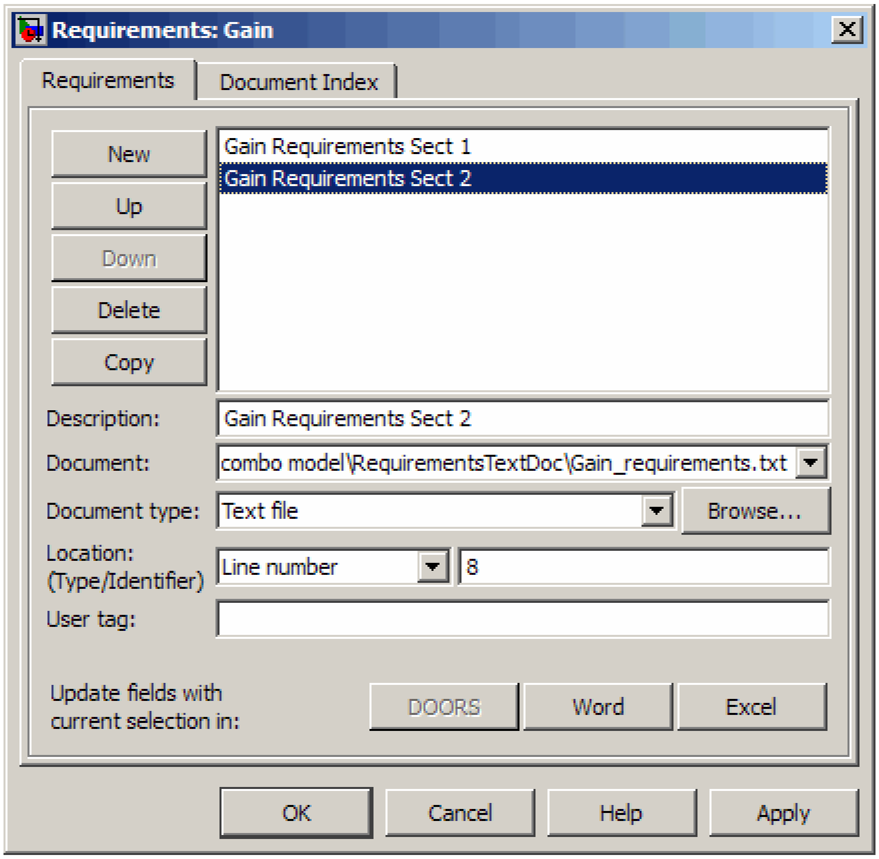
  ```
  BEGIN
    In1_signed <= signed(In1);

    --
    -- Block requirements for <S10>/Gain
    --   1. Gain Requirements Sect 1
    --   2. Gain Requirements Sect 2
    Gain_gainparam <= to_signed(16384, 16);

    Gain_out1 <= resize(In1_signed(15 DOWNTO 0) & '0'


    Out1 <= std_logic_vector(Gain_out1);

  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');
  ```

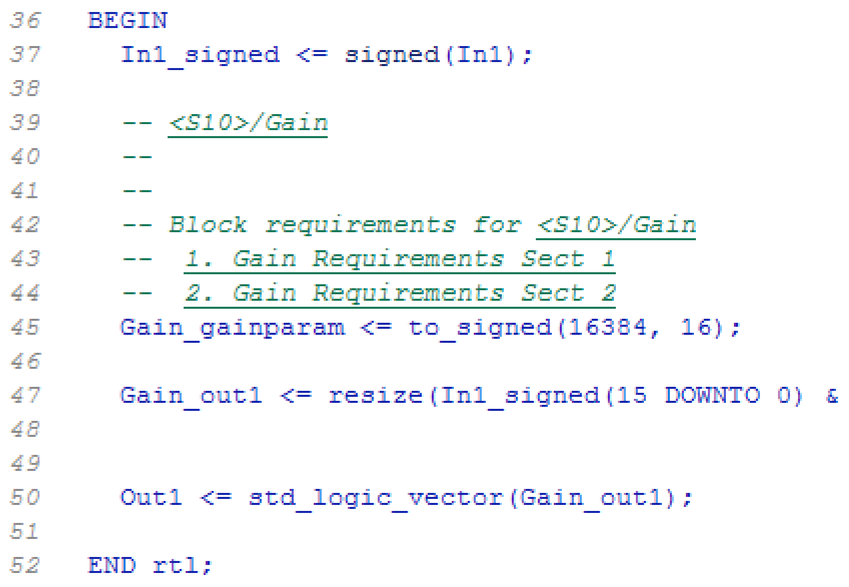
The comments include links back to a requirements document associated with the block and to the block within the original model. For example, the following figure shows two requirements links assigned to a Gain block. The links point to sections of a text requirements file.



Gain Requirements Sect 1
Gain Requirements Sect 2

Description: Gain Requirements Sect 2
Document: combo model\RequirementsTextDoc\Gain_requirements.txt
Document type: Text file
Location: (Type/Identifier) Line number 8
User tag:

The following figure shows hyperlinked requirements comments generated for the Gain block.



```
BEGIN
  In1_signed <= signed(In1);

  -- <S10>/Gain
  --
  --
  -- Block requirements for <S10>/Gain
  --  1. Gain Requirements Sect 1
  --  2. Gain Requirements Sect 2
  Gain_gainparam <= to_signed(16384, 16);

  Gain_out1 <= resize(In1_signed(15 DOWNTO 0) &


  Out1 <= std_logic_vector(Gain_out1);

END rtl;
```

Check Your Model for HDL Compatibility

The HDL compatibility checker lets you check whether a subsystem or model is compatible with HDL code generation. You can run the compatibility checker from the command line or from the GUI.

To run the compatibility checker from the command line, use the `checkhdl` function. The syntax of the function is

```
checkhdl('system')
```

where *`system`* is the device under test (DUT), typically a subsystem within the current model.

To run the compatibility checker from the GUI:

1 Open the Configuration Parameters dialog box or the Model Explorer. Select the **HDL Code Generation** pane.

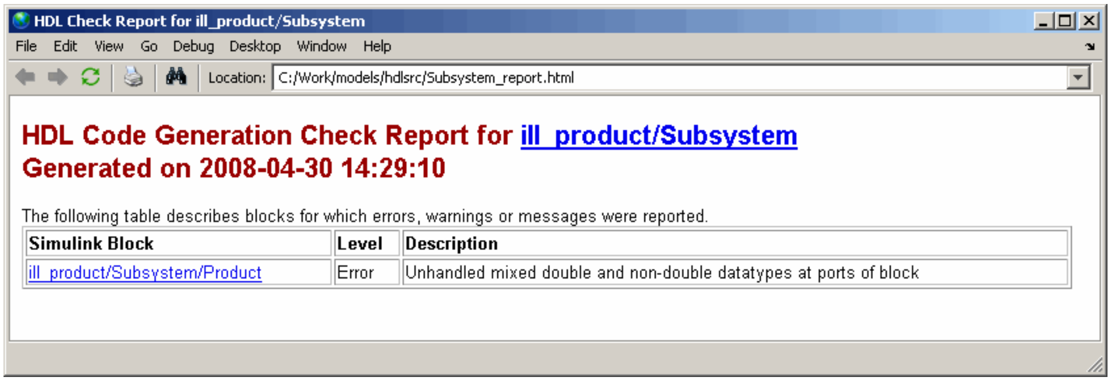
2 Select the subsystem you want to check from the **Generate HDL for** list.

3 Click the **Run Compatibility Checker** button.

The HDL compatibility checker examines the specified system for compatibility problems, such as use of unsupported blocks, illegal data type usage, etc. The HDL compatibility checker generates an HDL Code Generation Check Report, which is stored in the target folder. The report file naming convention is *`system`*`_report.html`, where *`system`* is the name of the subsystem or model 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 is hyperlinked to the block or subsystem that caused the problem. When you click the hyperlink, the block of interest highlights and displays (provided that the model referenced by the report is open).

The following figure shows an HDL Code Generation Check Report that was generated for a subsystem with a Product block that was configured with a mixture of double and integer port data types. This configuration is legal in a model, but incompatible with HDL code generation.

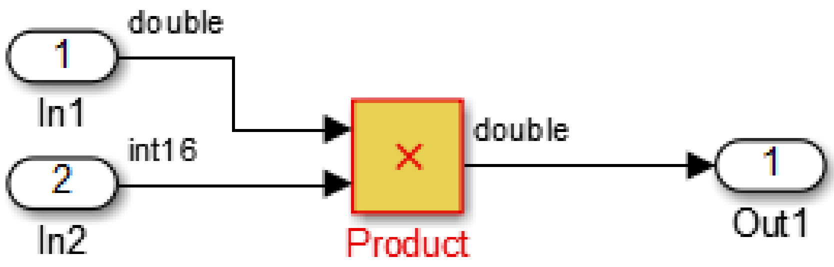

HDL Check Report for ill_product/Subsystem
File Edit View Go Debug Desktop Window Help
Location: C:/Work/models/hdlsrc/Subsystem_report.html

HDL Code Generation Check Report for ill_product/Subsystem
Generated on 2008-04-30 14:29:10

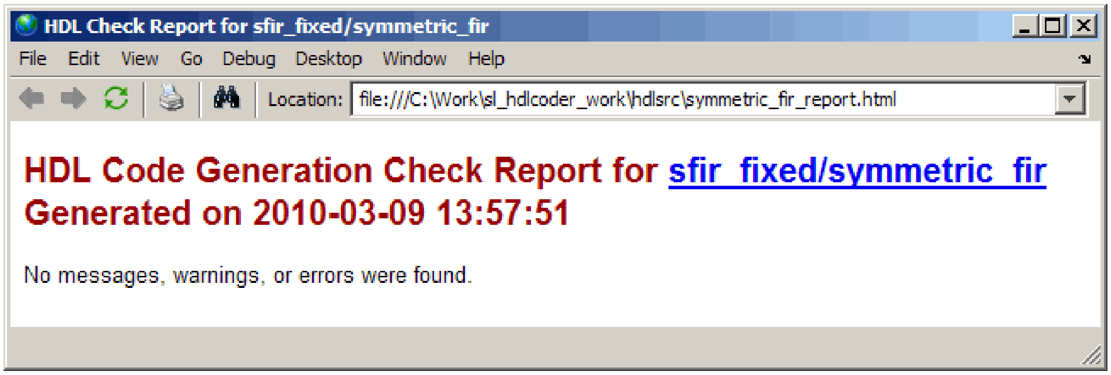
The following table describes blocks for which errors, warnings or messages were reported.

Simulink Block	Level	Description
ill_product/Subsystem/Product	Error	Unhandled mixed double and non-double datatypes at ports of block

When you click the hyperlink in the left column, the subsystem containing the offending block opens. The block of interest is highlighted, as shown in the following figure.



The following figure shows an HDL Code Generation Check Report that was generated for a subsystem that passed its compatibility checks. In this case, the report contains only a hyperlink to the subsystem that was checked.


HDL Check Report for sfir_fixed/symmetric_fir
File Edit View Go Debug Desktop Window Help
Location: file:///C:\Work\sl_hdlcoder_work\hdlsrc\symmetric_fir_report.html

HDL Code Generation Check Report for sfir_fixed/symmetric_fir
Generated on 2010-03-09 13:57:51

No messages, warnings, or errors were found.

Show Blocks Supported for HDL Code Generation

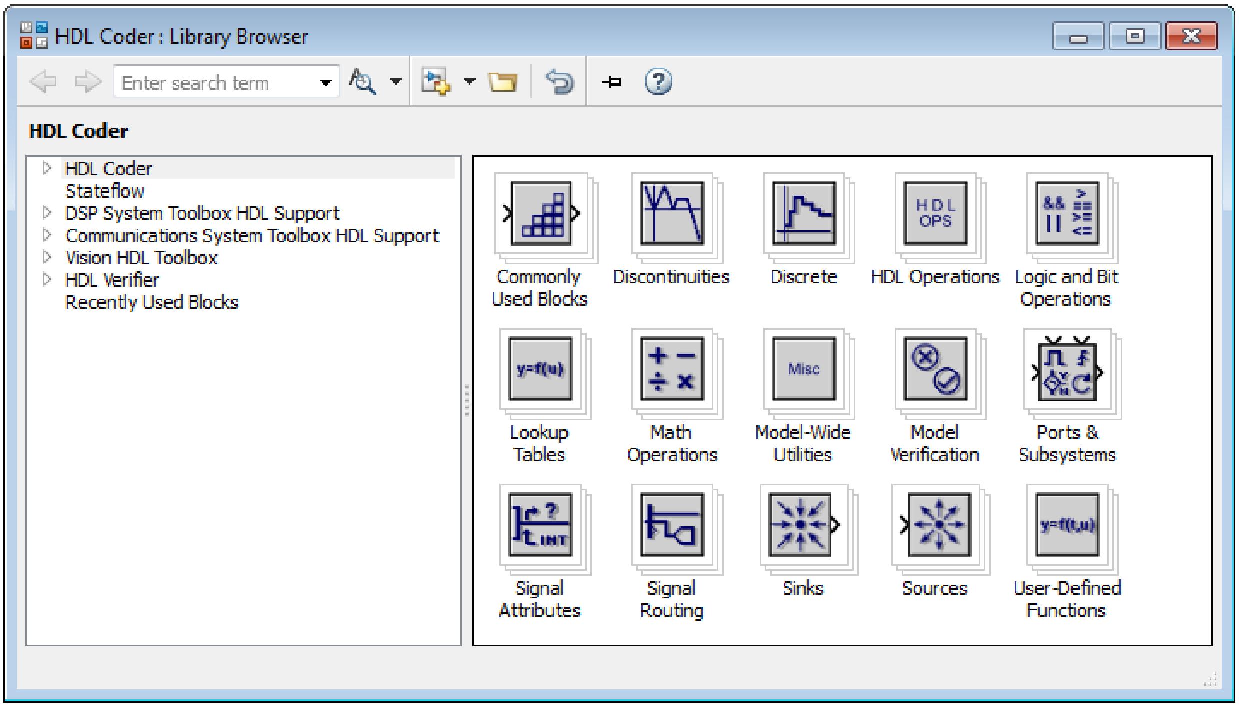
The `hdllib` function displays the blocks that are compatible with for HDL code generation in the Library Browser. It only displays those blocks for which you have a license. If you construct models using blocks from this Library Browser view, your models are compatible with HDL code generation.

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, at the command prompt, enter:

```
hdllib
```



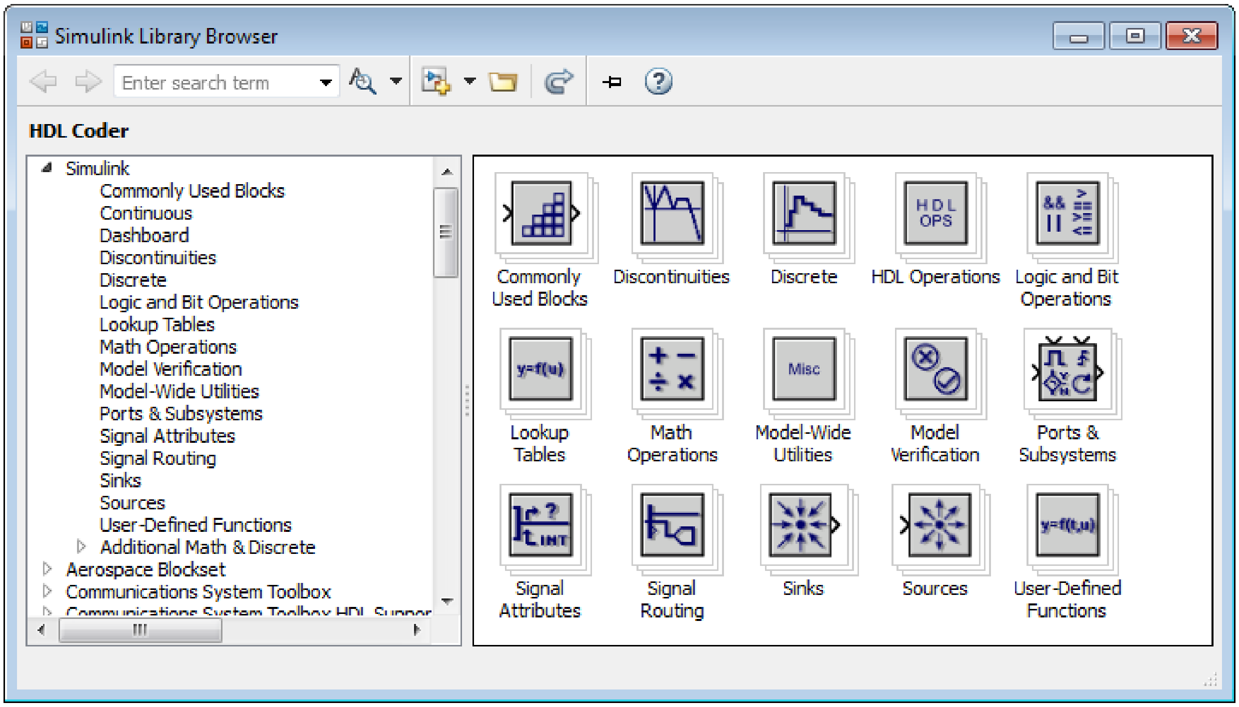
The Library Browser opens. You can drag and drop the blocks from the Library Browser into your model.

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.

See Also

Functions
`hdllib`

Related Examples

- “Prepare Simulink Model For HDL Code Generation”
- “View HDL-Specific Block Documentation” on page 12-3

More About

- “Supported Blocks”

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.

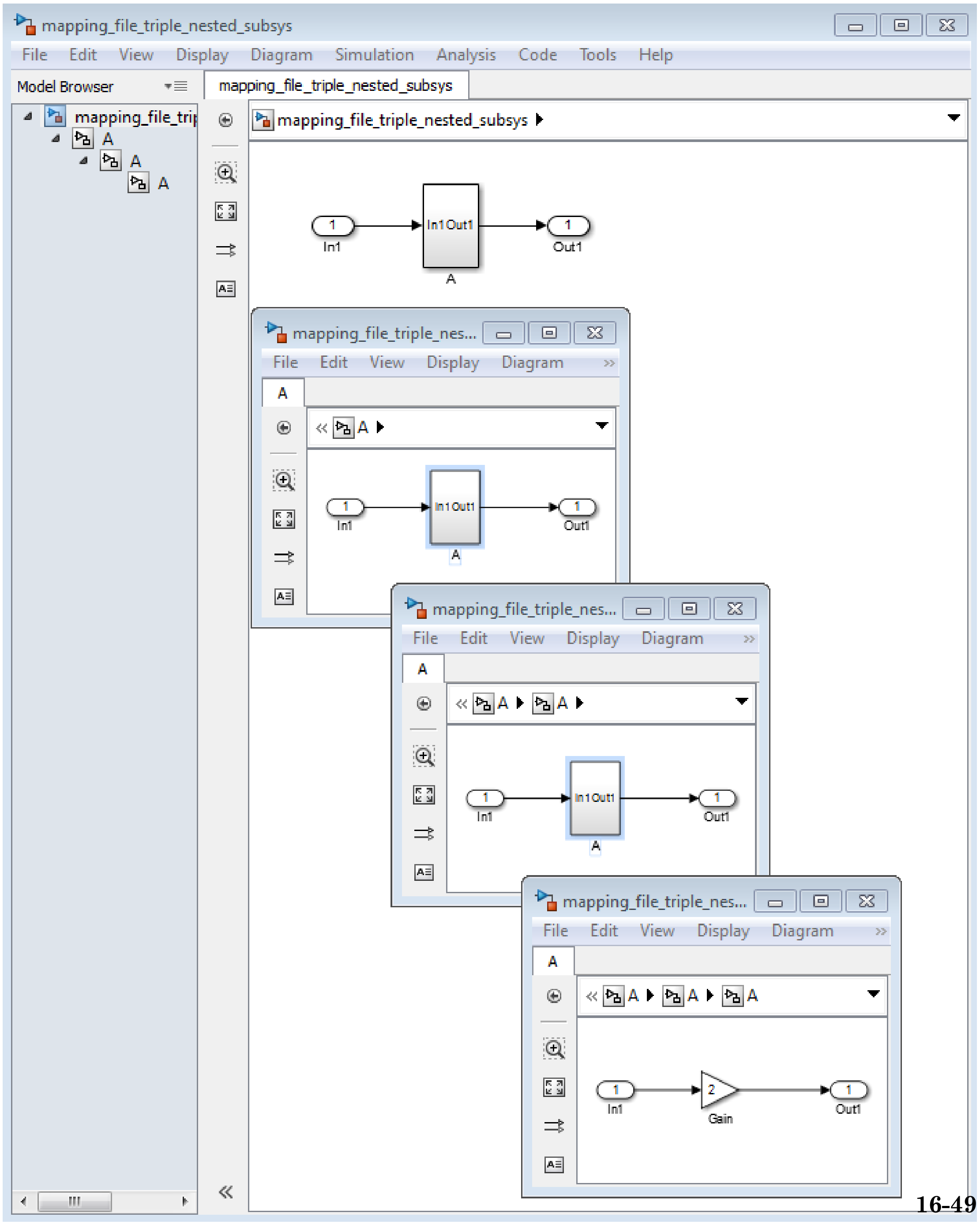
In simple cases, the mapping file may contain only one entry. For example, the `symmetric_fir` subsystem of the `sfir_fixed` model generates the following mapping file:

```
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,...`*`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.


mapping_file_triple_nested_subsys
File Edit View Display Diagram Simulation Analysis Code Tools Help
Model Browser
mapping_file_trip
A
A
A
mapping_file_triple_nested_subsys
In1
A
Out1
mapping_file_triple_nes...
File Edit View Display Diagram
A
In1
A
Out1
mapping_file_triple_nes...
File Edit View Display Diagram
A
In1
A
Out1
mapping_file_triple_nes...
File Edit View Display Diagram
A
In1
Gain
Out1

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?

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:

1. In the Simulink Editor, select **Code** > **HDL Code**.
2. Select **Add HDL Coder Configuration to Model**.
3. Save the model.

Removing the HDL Coder Configuration Component From a Model

To remove the HDL Coder configuration component from a model:

1. In the Simulink Editor, select **Code** > **HDL Code**, and select **Remove HDL Coder Configuration from Model**.

   HDL Coder displays a confirmation message.
2. Click **Yes** to confirm that you want to remove the HDL Coder configuration component.
3. Save the model.

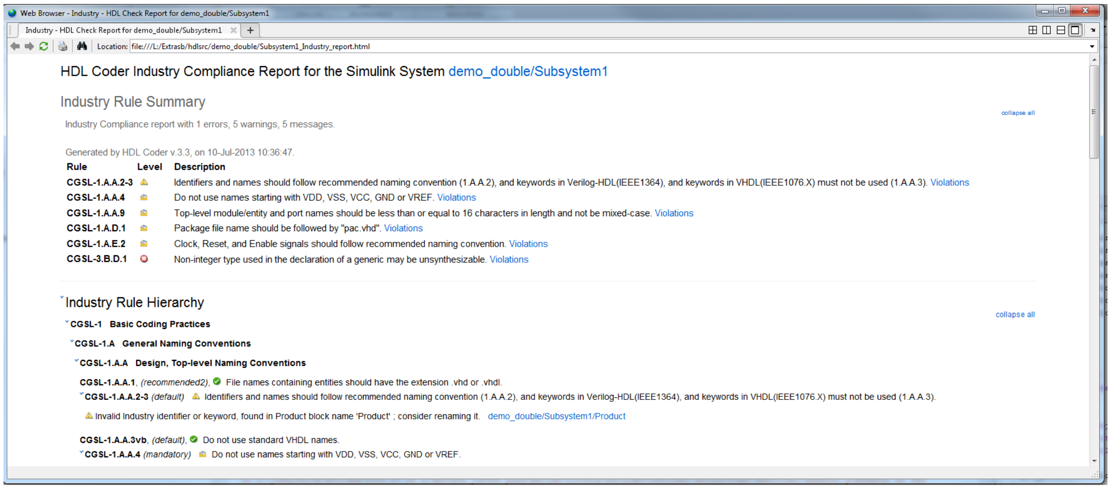
HDL Coding Standards

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.



Web Browser - Industry - HDL Check Report for demo_double/Subsystem1
Industry - HDL Check Report for demo_double/Subsystem1
Location: file:///L:/Extrasb/hdlsrc/demo_double/Subsystem1_Industry_report.html
collapse all
collapse all

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, *(recommended2)*, ✓ 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 17-4.

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 Properties.

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 17-9
- "RTL Description Techniques" on page 17-20
- "RTL Design Methodology Guidelines" on page 17-48

Related Examples

- "Generate an HDL Coding Standard Report from MATLAB" on page 5-44
- "Generate an HDL Coding Standard Report from Simulink" on page 17-5

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 17-2.

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 17-9.
- Section 2: “RTL Description Techniques” on page 17-20.
- Section 3: “RTL Design Methodology Guidelines” on page 17-48.

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 Properties.

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

Related Examples

- “Generate an HDL Coding Standard Report from MATLAB” on page 5-44
- “Generate an HDL Coding Standard Report from Simulink” on page 17-5

Generate an HDL Coding Standard Report from Simulink

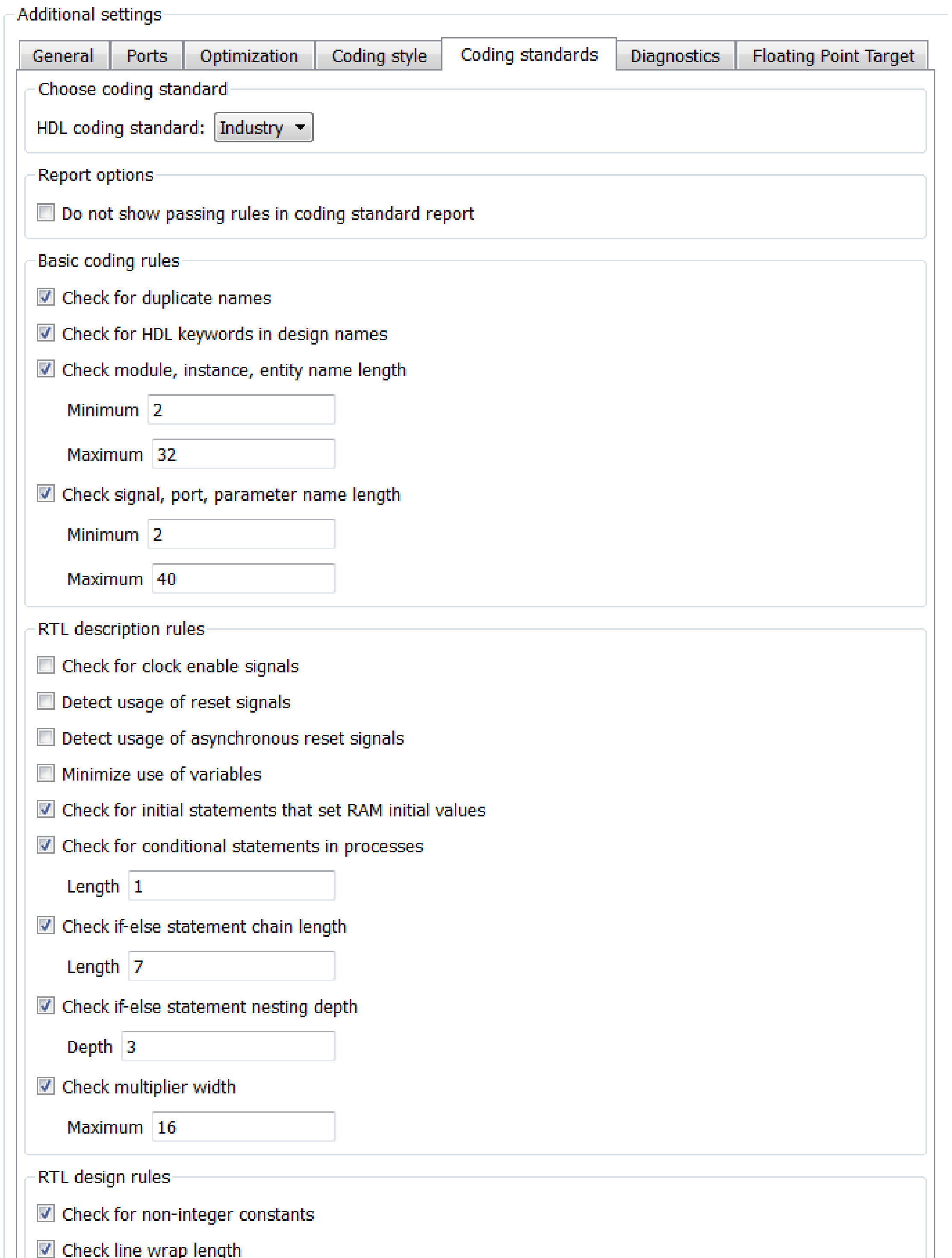
In this section...
"Using the HDL Workflow Advisor" on page 17-5
"Using the Command Line" on page 17-7

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 hdlsrc\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 5-44
- "Generate an HDL Coding Standard Report from Simulink" on page 17-5

More About

- "HDL Coding Standard Report" on page 17-2
- "Basic Coding Practices" on page 17-9
- "RTL Description Techniques" on page 17-20
- "RTL Design Methodology Guidelines" on page 17-48

Basic Coding Practices

In this section...

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.

Rule / Severity	Message	Problem	Recommendations
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(v3.1 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 (`_`).
1.A.A.5 Error	`Verilog/VHDL: Do not use case variants of`	Two or more names in your design, within the same scope, are	Update the names in your design so that no two names within the

Rule / Severity	Message	Problem	Recommendations
	`name in the same scope.`	identical except for case. For example, the names `foo` and `Foo` cannot be in the same scope.	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.`
	
	`VHDL: Component name should be same as its corresponding entity name.`
		HDL Coder generates code that complies with this rule for Verilog and VHDL.	No action required.

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 `ModuleInstanceEntityNameLeng` property of the HDL

Rule / Severity	Message	Problem	Recommendations
			coding standard customization object.

1.A.B Module Naming Conventions

Rule / Severity	Message	Problem	Recommendations
1.A.B.1–1b Error	`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.`	A module, instance, or entity name in the generated code is fewer than 2 characters or more than 32 characters in length.	Update the indicated name in your design so that it is from 2 through 32 characters in length. You can customize this rule by using the `ModuleInstanceEntityNameLeng` property of the HDL coding standard customization object.
	`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.`		

1.A.C Signal Naming Conventions

Rule / Severity	Message	Problem	Recommendations
1.A.C.3 Error	`Verilog: Signal names, port names, parameter names, define names and function names should be between 2 and 40 characters in length.`	A signal, port, parameter, define, or function name in the generated code is fewer than 2 characters, or more than 40 characters in length.	Update function names or subsystem names in your design to be from 2 through 40 characters in length. You can customize this rule by using the `SignalPortParamNameLength` property of the HDL coding standard customization object.
	`VHDL: Signal names, variable names, type names, label names, and function names should be between 2 and 40 characters in length.`		

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**

Rule / Severity	Message	Problem	Recommendations
			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 `MaskParameterAsGen` 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**

Rule / Severity	Message	Problem	Recommendations
			port options, update the names for the clock, reset, and enable signals respectively.

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.

Rule / Severity	Message	Problem	Recommendations
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`	Your Simulink model uses multiple clock signals.	Update your design to use a single clock signal. In the **HDL**

Rule / Severity	Message	Problem	Recommendations
	`a single clock only.`		**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 5-44
- “Generate an HDL Coding Standard Report from Simulink” on page 17-5

More About

- “HDL Coding Standard Report” on page 17-2
- “RTL Description Techniques” on page 17-20
- “RTL Design Methodology Guidelines” on page 17-48

RTL Description Techniques

In this section...

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_116 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`	HDL Coder does not generate functions for DUT.	No action required.

Rule / Severity	Message	Problem	Recommendations
	`possible states of the function.`		
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.

Rule / Severity	Message	Problem	Recommendations
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.

Rule / Severity	Message	Problem	Recommendations
	`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`	In the generated Verilog code, HDL Coder complies with	No action required.

Rule / Severity	Message	Problem	Recommendations
	`logical operators in multi-bit operations.`	this rule for multibit operators.	
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`	In the generated code, HDL Coder does not	No action required.

Rule / Severity	Message	Problem	Recommendations
	`z for an array index.`	use x or z for an array index.	

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`	HDL Coder complies with this rule.	No action required.

Rule / Severity	Message	Problem	Recommendations
	`not read Global signals.`		
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,`	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.

Rule / Severity	Message	Problem	Recommendations
	procedures, or selected signal assignments.		
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 senstivity list of a process or always block, do not define`	HDL Coder generates code that complies with the use of these constructs inside a `process` block	No action required.

Rule / Severity	Message	Problem	Recommendations
	`constants, use wait statements, or include a signal that is not read inside that block.`	(VHDL) or an `always` block (Verilog).	
	`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`	HDL Coder does not generate multiple asynchronous resets. The generated code can contain multiple synchronous resets.	No action required.

Rule / Severity	Message	Problem	Recommendations
	`reset description.`		
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.`	By default, HDL Coder uses the event syntax for clock events. By using the `UseRisingEdge` property, you can specify whether to use the `rising_edge` or `falling_edge` to detect clock transitions.	To fix this error, you can control the `UseRisingEdge` 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.	To minimize clock enables in the generated HDL code, in the HDL coding standard customization properties, enable the **MinimizeClockEnableCheck** property. To remove reset signals on the networks, in the HDL coding standard customization properties, enable the **RemoveResetCheck** setting.
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 initial construct.`	The generated HDL code for your design contains an unsynthesizable `initial` statement.	Disable the **Initialize block RAM** or **Initialize all RAM blocks** option in the HDL Workflow Advisor. You can disable this rule checking by using the `InitialStatements` property of the HDL coding standard customization object.

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`	HDL Coder does not create latches in the generated code.	No action required.

Rule / Severity	Message	Problem	Recommendations
	`asynchronous set or asynchronous reset, or be mixed with other descriptions in the same module.`		
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`	HDL Coder does not create latches or tristate buffers in the generated code.	No action required.

Rule / Severity	Message	Problem	Recommendations
	`of drivers. A net that is not tristated or a signal without a resolution function must not have multiple drivers.`		
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.`	By default, HDL Coder does not connect input or output ports directly to bidirectional ports. 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.	In your Simulink model, on the HDL block properties for the input or output port, set **BidirectionalPort** to `off`.

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 `ConditionalRegionCheck` 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.	Modify `if-elseif` statements in your MATLAB code so there are N or fewer levels of nesting.
			
			For example, the following `if-elseif` pseudocode contains three levels of nesting:
			
			`if ...`
			`   if ...`
			`      if ...`
			`      else`
			`   else`
			`else`
			
			You can customize this rule by using the `IfElseNesting` property of the HDL coding standard customization object.

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.	Modify `if-elseif` statements in your MATLAB code so that the number of branches is seven or fewer.
			
			For example, the following `if-elseif` pseudocode contains three branches:
			
			`if ...`
			`elseif ...`
			`elseif ...`
			`else`

Rule / Severity	Message	Problem	Recommendations
			You can customize this rule by using the `IfElseChain` property of the HDL coding standard customization object.

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`	HDL Coder describes all possible cases in a	No action required.

Rule / Severity	Message	Problem	Recommendations
	`full_case pragma when all conditions are not described as case clause or the default clause is missing.`	case statement so that the synthesis tool does not infer a latch.	
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 *don't care* 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 *don't care* 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.

Rule / Severity	Message	Problem	Recommendations
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' or constants that contain the values 'X','Z','U''-','W'`		
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`	HDL Coder ensures that the data types of the operands match in	No action required.

Rule / Severity	Message	Problem	Recommendations
	`relational or logical operator must match.`	a relational or logical expression.	
	`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`	HDL Coder does not use the function	No action required.

Rule / Severity	Message	Problem	Recommendations
	`should not be used in the design.`	`To_stdlogicvector` in the code.	

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.	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. You can customize this rule by using the `MultiplierBitWidth` property of the HDL coding standard customization object.

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 5-44

- “Generate an HDL Coding Standard Report from Simulink” on page 17-5

More About

- “HDL Coding Standard Report” on page 17-2
- “Basic Coding Practices” on page 17-9
- “RTL Design Methodology Guidelines” on page 17-48

RTL Design Methodology Guidelines

In this section...

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 decalarations 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`	The generated HDL code contains multiple assignments in one	Shorten names in your design that are longer than N characters.

Rule / Severity	Message	Problem	Recommendations
	`not be made in one line.`	line or lines greater than N characters. You have a name or identifier in your original design that contains more than N characters.	You can also customize N by using the `LineLength` 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.
	`Verilog/VHDL: The maximum number of characters in one line should not be more than N.`		

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.

Rule / Severity	Message	Problem	Recommendations
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`	The generated HDL code contains a noninteger data type.	If you have floating-point data types in your design, you can map

Rule / Severity	Message	Problem	Recommendations
	`declaration of a generic may be unsynthesizable.`		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 `NonIntegerTypes` property of the HDL coding standard customization object.
3.B.D.3 Error	`Verilog: Do not use defparam 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 aynschronous 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 `TriggerAsClock`, then the trigger signal becomes a clock signal that you can control from external pins.	To avoid this rule violation, disable the `TriggerAsClock`.

Rule / Severity	Message	Problem	Recommendations
		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.	

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 5-44
- “Generate an HDL Coding Standard Report from Simulink” on page 17-5

More About

- “HDL Coding Standard Report” on page 17-2
- “Basic Coding Practices” on page 17-9
- “RTL Description Techniques” on page 17-20

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

In this section...

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

Using the UI

To generate HDL code for referenced model 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.

Tip: If you encounter typing or naming conflicts between vector ports when interfacing two or more generated VHDL code modules, consider using the ScalarizePorts property to generate nonconflicting port definitions.

Using the Command Line

To generate HDL code for a referenced model using the command line:

1. Set the `Architecture` property of the Model block to `ModelReference`.
2. Generate HDL code for your DUT subsystem.

For example, to generate HDL code for a DUT subsystem, `mydut`, that includes a model reference, `referenced_model`, at the command line, enter:

```
hdlset_param ('mydut/referenced_model', 'Architecture', 'ModelReference');
makehdl ('mydut');
```

Tip: If you encounter typing or naming conflicts between vector ports when interfacing two or more generated VHDL code modules, consider using the ScalarizePorts property to generate nonconflicting port definitions.

Limitations for Model Reference Code Generation

For model reference code generation restrictions, see Model.

Generate Black Box Interface for Subsystem

In this section...

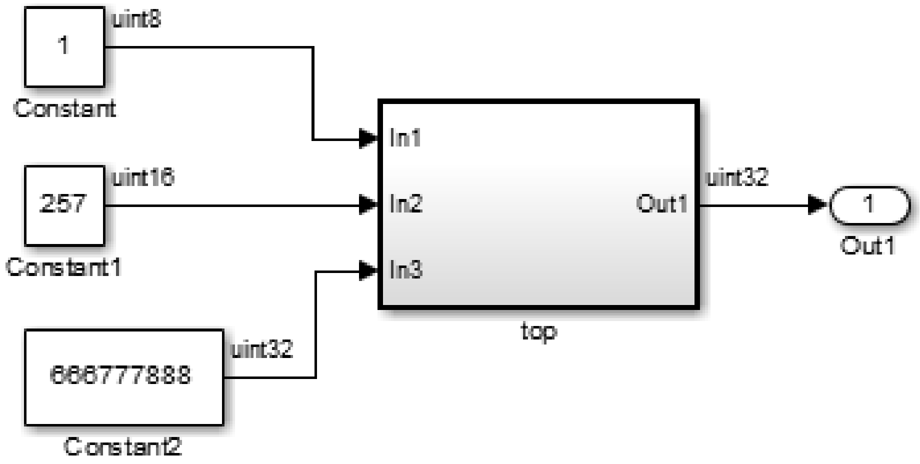
What Is a Black Box Interface?

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.

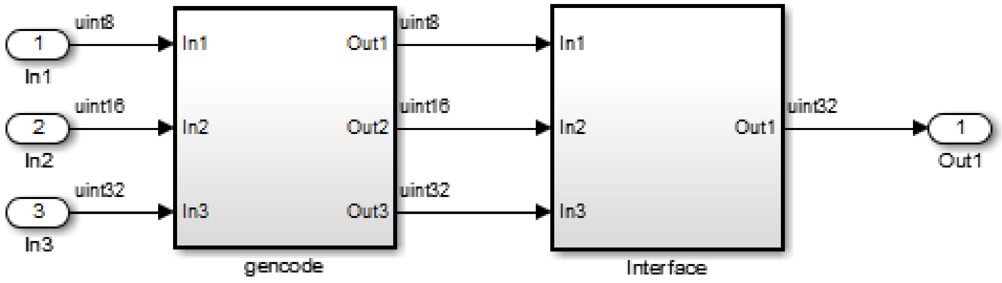
The black box implementation is available only for subsystem blocks below the level of the DUT. Virtual and atomic subsystem blocks of custom libraries that are below the level of the DUT also work with black box implementations.

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:



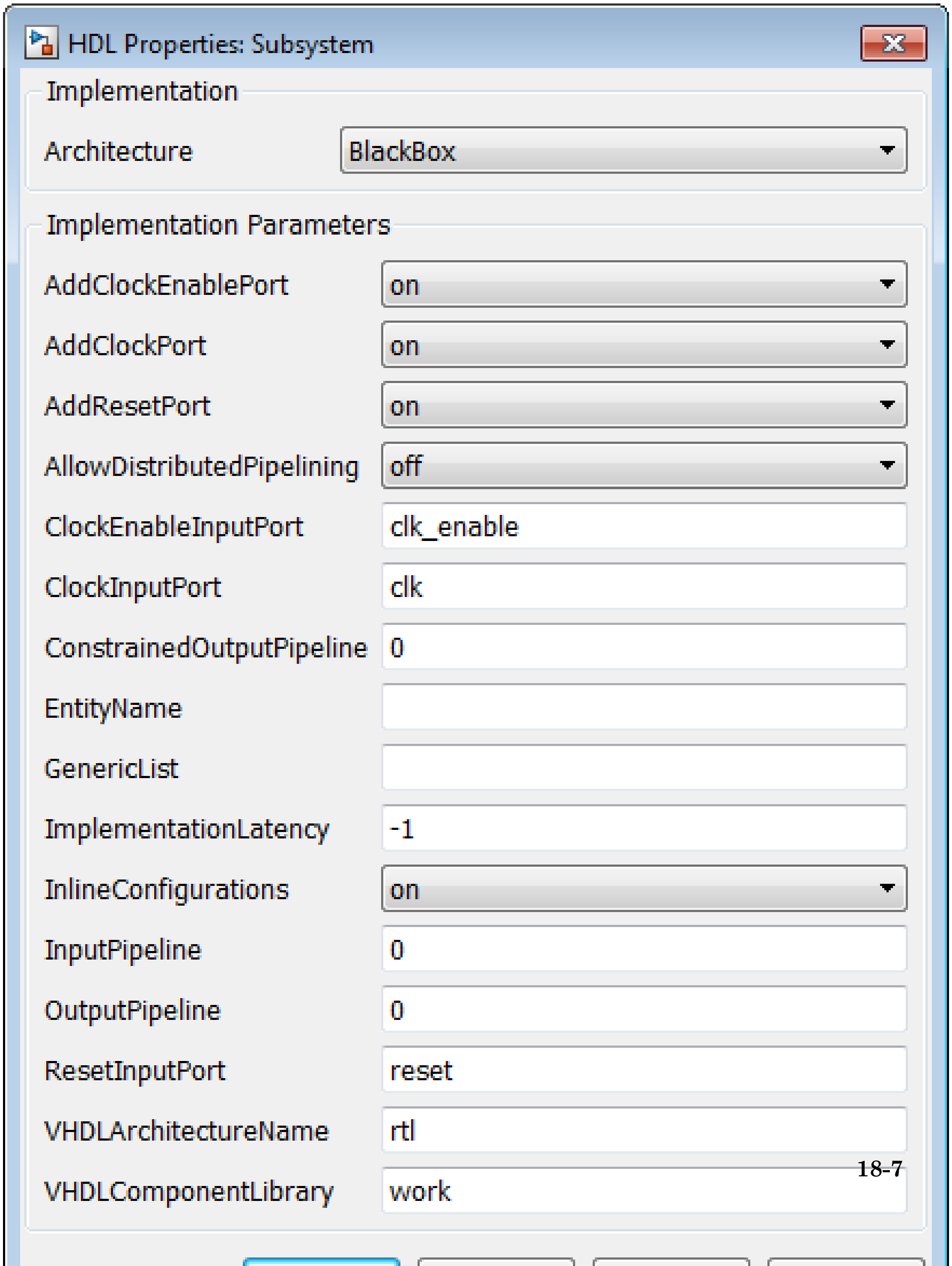
Suppose that you want to generate HDL code from `top`, with a black box interface from the `Interface` subsystem. To specify a black box interface:

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:



HDL Properties: Subsystem

Implementation

Architecture	BlackBox

Implementation Parameters

AddClockEnablePort	on
AddClockPort	on
AddResetPort	on
AllowDistributedPipelining	off
ClockEnableInputPort	clk_enable
ClockInputPort	clk
ConstrainedOutputPipeline	0
EntityName	
GenericList	
ImplementationLatency	-1
InlineConfigurations	on
InputPipeline	0
OutputPipeline	0
ResetInputPort	reset
VHDLArchitectureName	rtl
VHDLComponentLibrary	work

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 18-14 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 18-14 describes how you can rename or suppress generation of these signals, and customize other aspects of the generated interface.

Restriction for Multirate DUTs

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.

More About

- “Customize Black Box or HDL Cosimulation Interface” on page 18-14
- “Generate Black Box Interface for Referenced Model” on page 18-10
- “Integrate Custom HDL Code Using DocBlock” on page 18-12

Generate Black Box Interface for Referenced Model

In this section...
"When to Generate a Black Box Interface" on page 18-10
"How to Generate a Black Box Interface" on page 18-10

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 18-2.

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 18-14.
2. Generate HDL code for your DUT subsystem.

Note: The `checkhdl` function does not check port data types within the referenced model.

Tip: If you encounter typing or naming conflicts between vector ports when interfacing two or more generated VHDL code modules, consider using the ScalarizePorts property to generate nonconflicting port definitions.

More About

- "Customize Black Box or HDL Cosimulation Interface" on page 18-14
- "Generate Black Box Interface for Subsystem" on page 18-4
- "Integrate Custom HDL Code Using DocBlock" on page 18-12

Integrate Custom HDL Code Using DocBlock

In this section...

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 18-4 or black box model reference on page 18-10.

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 18-14.
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 `VHDL`˙.

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.

More About

- "Customize Black Box or HDL Cosimulation Interface" on page 18-14
- "Generate Black Box Interface for Subsystem" on page 18-4
- "Generate Black Box Interface for Referenced Model" on page 18-10

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.

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.

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 `entity` or Verilog `module` name generated for the block.

GenericList	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 `integer`.
	
	Default: none
		Specifies a list of VHDL `generic` or Verilog `parameter` name-value pairs, each with an optional data type specification, to pass to a subsystem with a `BlackBox` implementation.
		
		For example, in the HDL Block Properties dialog box, enter `{'`*`name`*`','`*`value`*`','`*`type`*`'}`, or, if the data type is `integer`, enter `{'`*`name`*`','`*`value`*`'}`.
		
		To set `GenericList` using `hdlset_param`, at the command line, enter:
		
		`hdlset_param (`*`blockname`*`,'GenericList','{''`*`name`*`'','''`
		
		If the data type is `integer`, at the command line, enter:
		
		`hdlset_param (`*`blockname`*`,'GenericList','{''`*`name`*`'',''`*`value`*`''}');`

Parameter Name	Values	Description
ImplementationLatency	-1 | 0 | positive integer Default: -1	Specifies the additional latency of the external component in time steps, relative to the Simulink block. If 0 or greater, this value is used for delay balancing. Your inputs and outputs must operate at the same rate. If -1, latency is unknown. This disables delay balancing.
InlineConfigurations (VHDL only)	`on` | `off` Default: If this parameter is unspecified, defaults to the value of the global `InlineConfigurations` property.	If `off`, suppress generation of a configuration for the block, and require a user-supplied external configuration.
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: `reset`	Specifies HDL name for block's reset input.
VHDLArchitectureName (VHDL only)	Default: `rtl`	Specifies RTL architecture name generated for the block. The architecture name is generated only if **InlineConfigurations** is `on`.
VHDLComponentLibrary (VHDL only)	Default: `work`	Specifies the library from which to load the VHDL component.

More About

- “Generate Black Box Interface for Subsystem” on page 18-4

- “Generate Black Box Interface for Referenced Model” on page 18-10
- “Integrate Custom HDL Code Using DocBlock” on page 18-12
- “Specify Bidirectional Ports” on page 18-18

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 18-18
"How To Specify a Bidirectional Port" on page 18-18
"Limitations" on page 18-19

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 Inport 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 Inport, *`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.
- Simulink does not support bidirectional ports, so you cannot simulate the bidirectional behavior in Simulink.

More About

- "Generate Black Box Interface for Subsystem" on page 18-4
- "Generate Black Box Interface for Referenced Model" on page 18-10
- "Integrate Custom HDL Code Using DocBlock" on page 18-12
- "Customize Black Box or HDL Cosimulation Interface" on page 18-14

Generate Reusable Code for Atomic Subsystems

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.

In this section...

Requirements for Generating Reusable Code for Atomic Subsystems

To generate reusable HDL code for atomic subsystems:

- `HandleAtomicSubsystem` must be `on`.
- The `DefaultParameterBehavior` Simulink configuration parameter must be `Inlined`.
- The atomic subsystems must be identical, or identical except for their mask parameter values.

  If the atomic subsystems are identical except for their mask parameter values:

  - `MaskParameterAsGeneric` must be `on`.
  - Mask parameters must be 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.

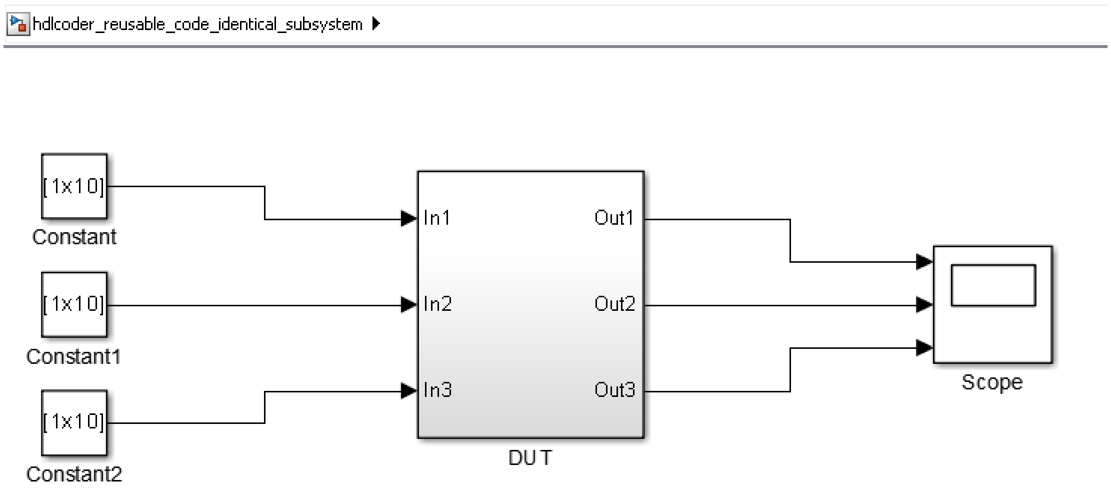
If you do not want to generate reusable code for identical atomic subsystems, or for atomic subsystems that are identical except for their tunable mask parameter values, disable the `HandleAtomicSubsystem` property. By default, `HandleAtomicSubsystem` is enabled.

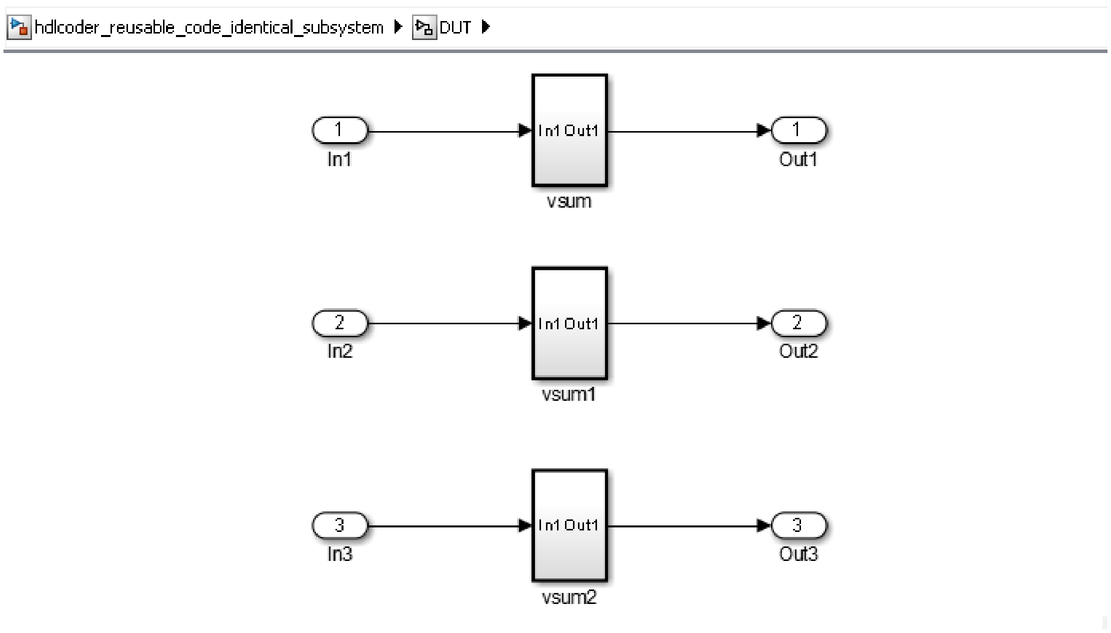
For example, to disable `HandleAtomicSubsystem` for the `hdlcoder_reusable_code_identical_subsystem` model, enter:

```
hdlset_param('hdlcoder_reusable_code_identical_subsystem','HandleAtomicSubsystem','off')
```

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\vsum.vhd.
### Working on hdlcoder_reusable_code_identical_subsystem/DUT as
    hdl_prj\hdlsrc\hdlcoder_reusable_code_identical_subsystem\DUT.vhd.
### Generating package file hdl_prj\hdlsrc\hdlcoder_reusable_code_identical_subsystem\DUT_pkg.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.

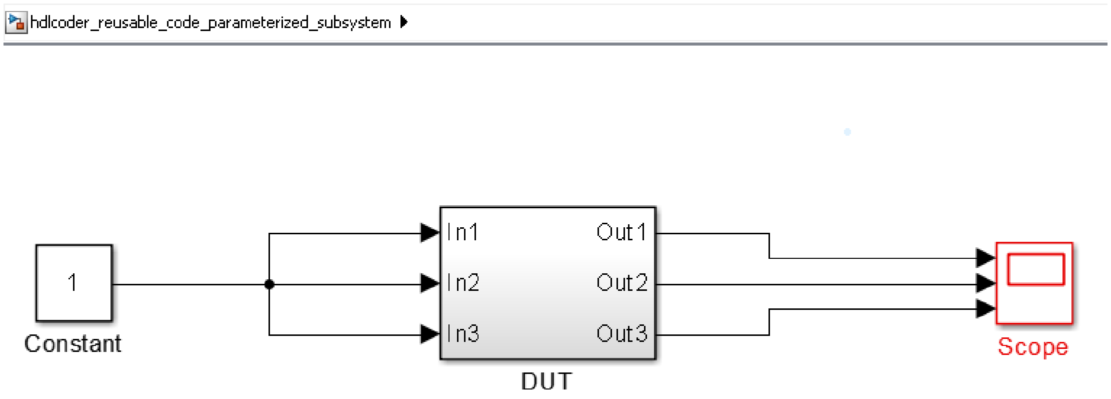
For example, to enable the generation of reusable code for the atomic subsystems with tunable parameters in the `hdlcoder_reusable_code_parameterized_subsystem` model, enter:

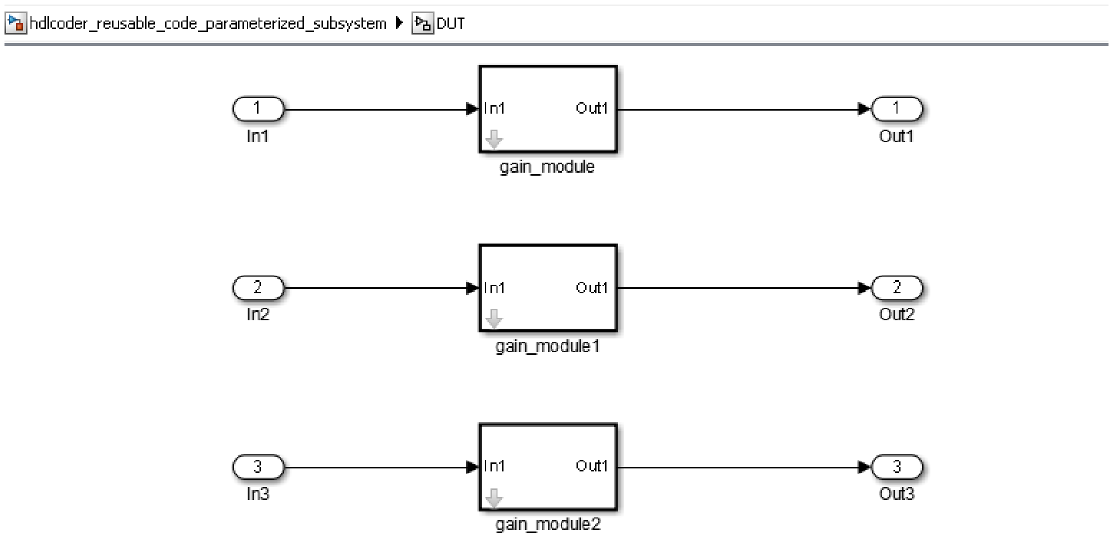
```
hdlset_param('hdlcoder_reusable_code_parameterized_subsystem','MaskParameterAsGeneric','on')
```

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

`HandleAtomicSubsystem` | `MaskParameterAsGeneric`

More About

- “Generate parameterized HDL code from masked subsystem” on page 11-71
- “Generate Parameterized Code for Referenced Models” on page 10-22
- “Create and Add Tunable Parameter That Maps to DUT Ports” on page 10-20

Create a Xilinx System Generator Subsystem

In this section...

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, you must use Vivado or ISE Design Suite 13.4 or later.

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 18-29.

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.

For an example of HDL code generation from a Xilinx System Generator subsystem, see “Using Xilinx System Generator for DSP with HDL Coder” on page 18-33.

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.

Create an Altera DSP Builder Subsystem

In this section...

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 18-31 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 Xilinx System Generator for DSP with HDL Coder

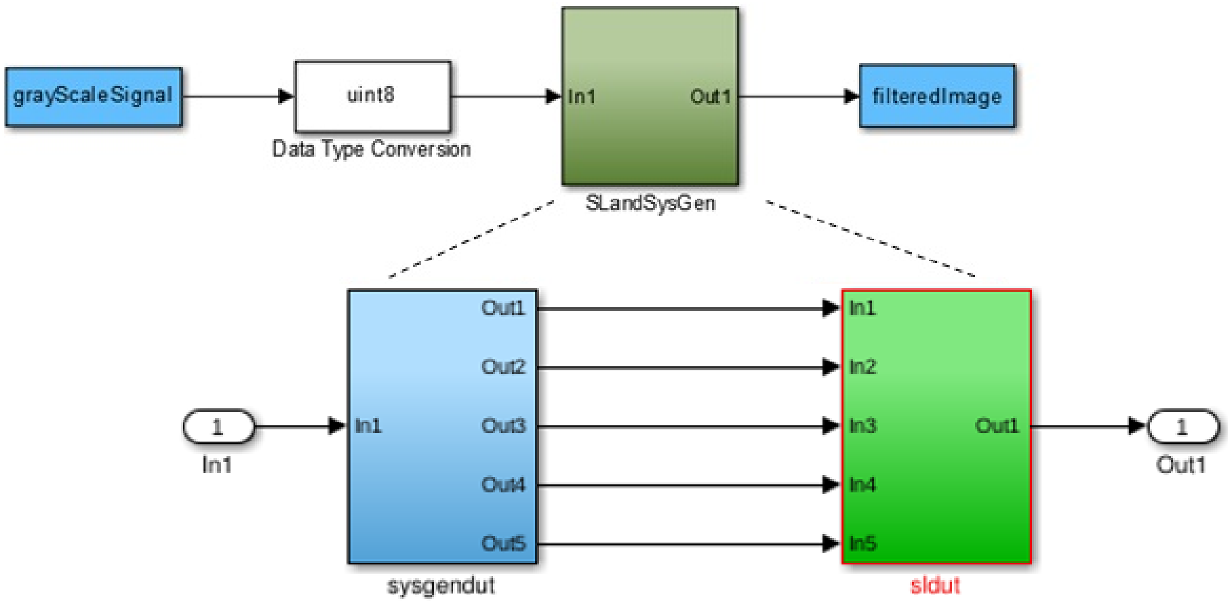
This example shows how to use Xilinx® System Generator for DSP with HDL Coder™.

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 in a Xilinx® System Generator Subsystem (hdlcoder_slsysgen/SLandSysGen/sysgendut). System Generator optimizes these blocks for Xilinx® 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_slsysgen');
open_system('hdlcoder_slsysgen/SLandSysGen');
```

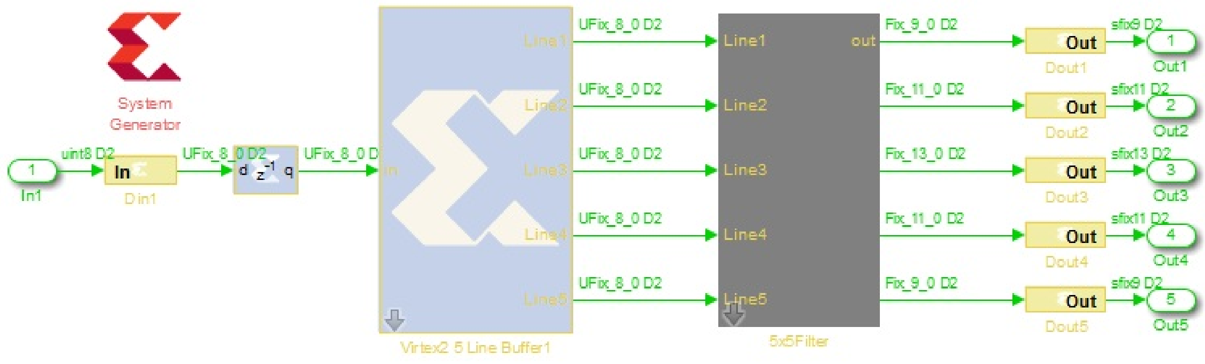


Create Xilinx® System Generator Subsystem

To create a Xilinx® System Generator subsystem:

1. Put the Xilinx® blocks in one subsystem and set its architecture to "Module" (the default value).
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.

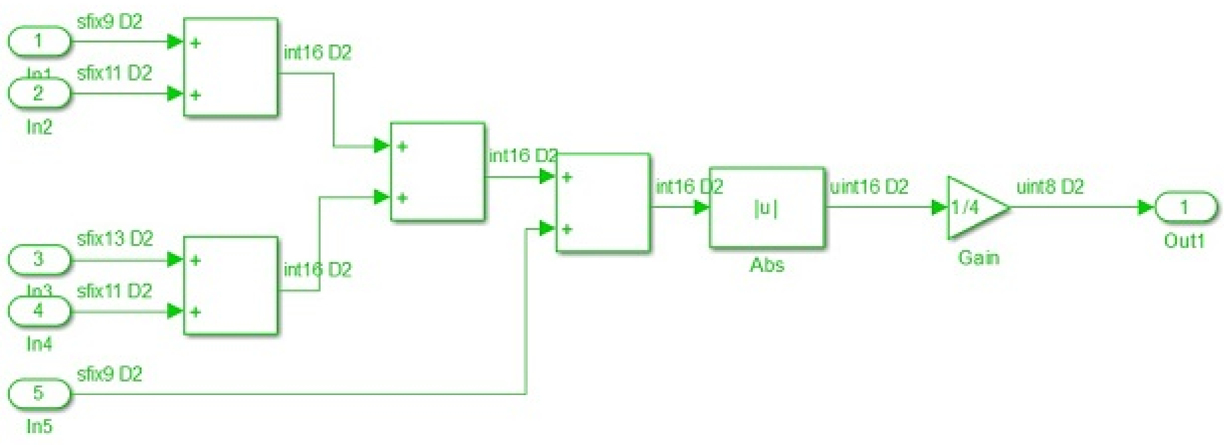
Gateway In blocks must not do non-trivial data type conversion. For example, a Gateway In block can convert between uint8 and UFix_8_0, but changing data sign, word length, or fraction length is not allowed.

Perform Model-Level Optimizations for Simulink® Components

In this example, a sum tree is modeled with Simulink® blocks. The distributed pipelining feature can take care of the speed optimization.

Here the Output Pipeline property of hdlcoder_slsysgen/SLandSysGen/Simulink Subsystem is set to "4" and the Distributed Pipelining property is set to "on". Pipeline registers inserted by the distributed pipelining feature will be pushed 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/sldut');
```



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 23-2.
- 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 18-40.
- 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 .
- 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		• Fast compile time in HDL simulator	• 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
Cosimulation model	• HDL Verifier	• 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	• HDL Verifier • Simulink Coder	• 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	• Run test in HDL simulator • No tunable parameters in stimulus generation

Test Bench	License Requirements	Pros	Cons
FPGA-in-the-loop	• HDL Verifier • HDL Verifier Support Package for Xilinx FPGA Boards or HDL Verifier Support Package for Altera FPGA Boards	• Run tests from Simulink, including changing parameters to affect input stimulus • Prototype hardware implementation of your DUT	• Long generation time due to synthesis into FPGA • Hardware setup

More About

- "HDL Code Generation Pane: Test Bench" on page 11-122
- "Set HDL Code Generation Options" on page 11-2

Generate a Cosimulation Model

In this section...

Note: To use this feature, your installation must include an HDL Verifier license.

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 .

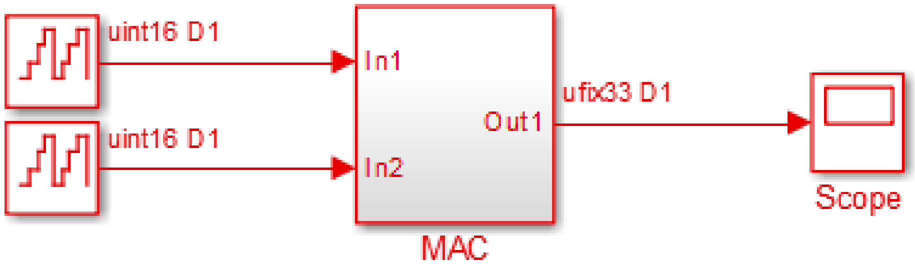
In addition to the generated model, HDL Coder generates a TCL script that launches and configures your cosimulation tool. Comments in the script file document clock, reset, and other timing signal information defined by the coder for the cosimulation tool.

Generating a Cosimulation Model from the GUI

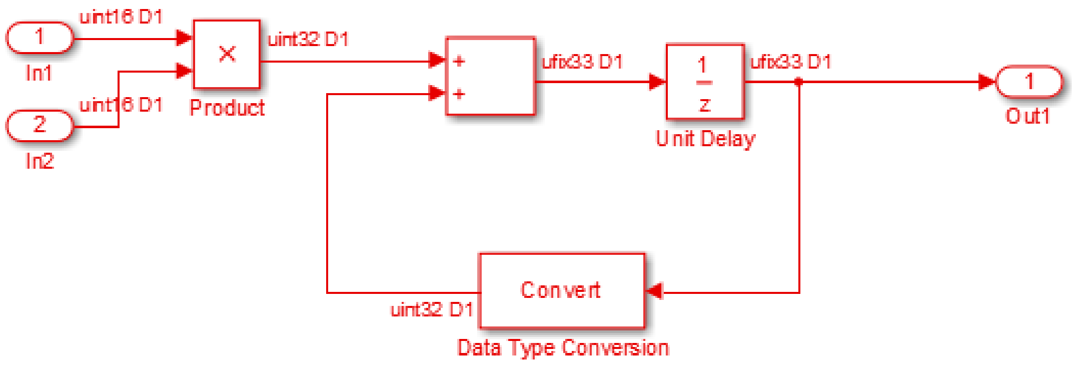
This example demonstrates the process for generating a cosimulation model. The example model, `hdl_cosim_demo1`, implements a simple multiply and accumulate (MAC) algorithm. Open the model by entering the name at the MATLAB command line:

```
hdl_cosim_demo1
```

The following figure shows the top-level model.

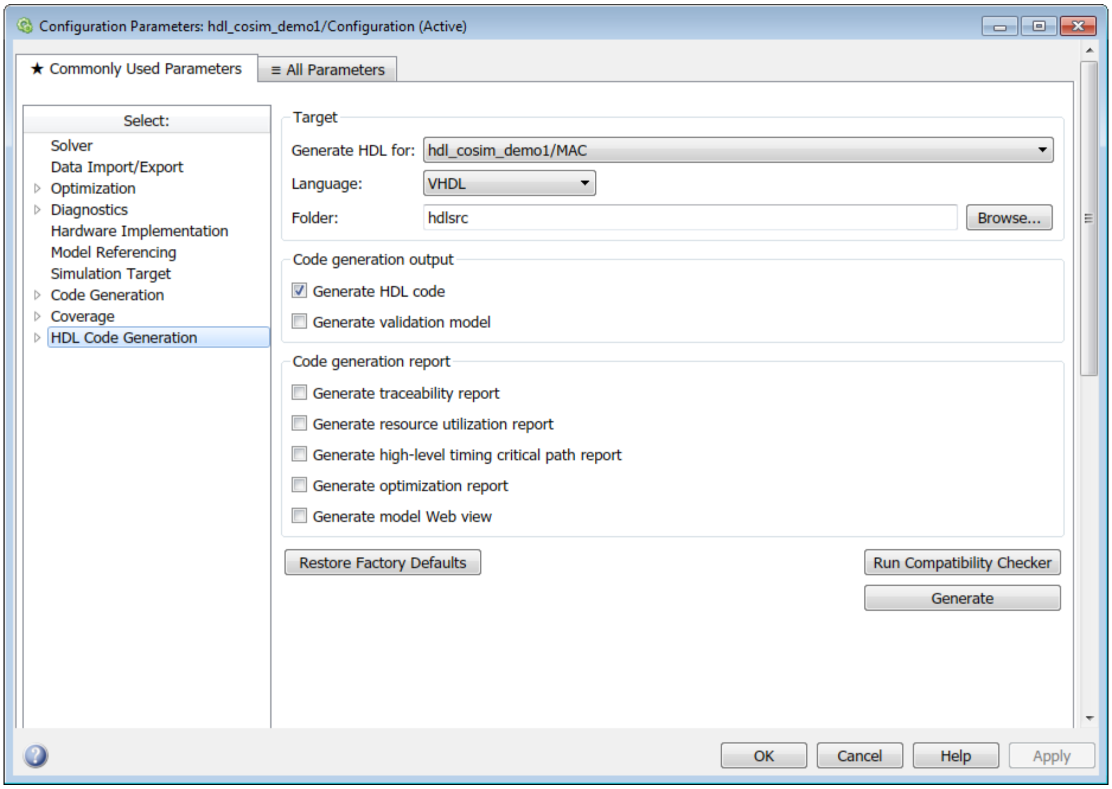


The DUT is the MAC subsystem.



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`.


Configuration Parameters: hdl_cosim_demo1/Configuration (Active)

Target
Generate HDL for: hdl_cosim_demo1/MAC
Language: VHDL
Folder: hdlsrc

Code generation output
☑ Generate HDL code
☐ Generate validation model

Code generation report
☐ Generate traceability report
☐ Generate resource utilization report
☐ Generate high-level timing critical path report
☐ Generate optimization report
☐ Generate model Web view

2 Click **Apply**.

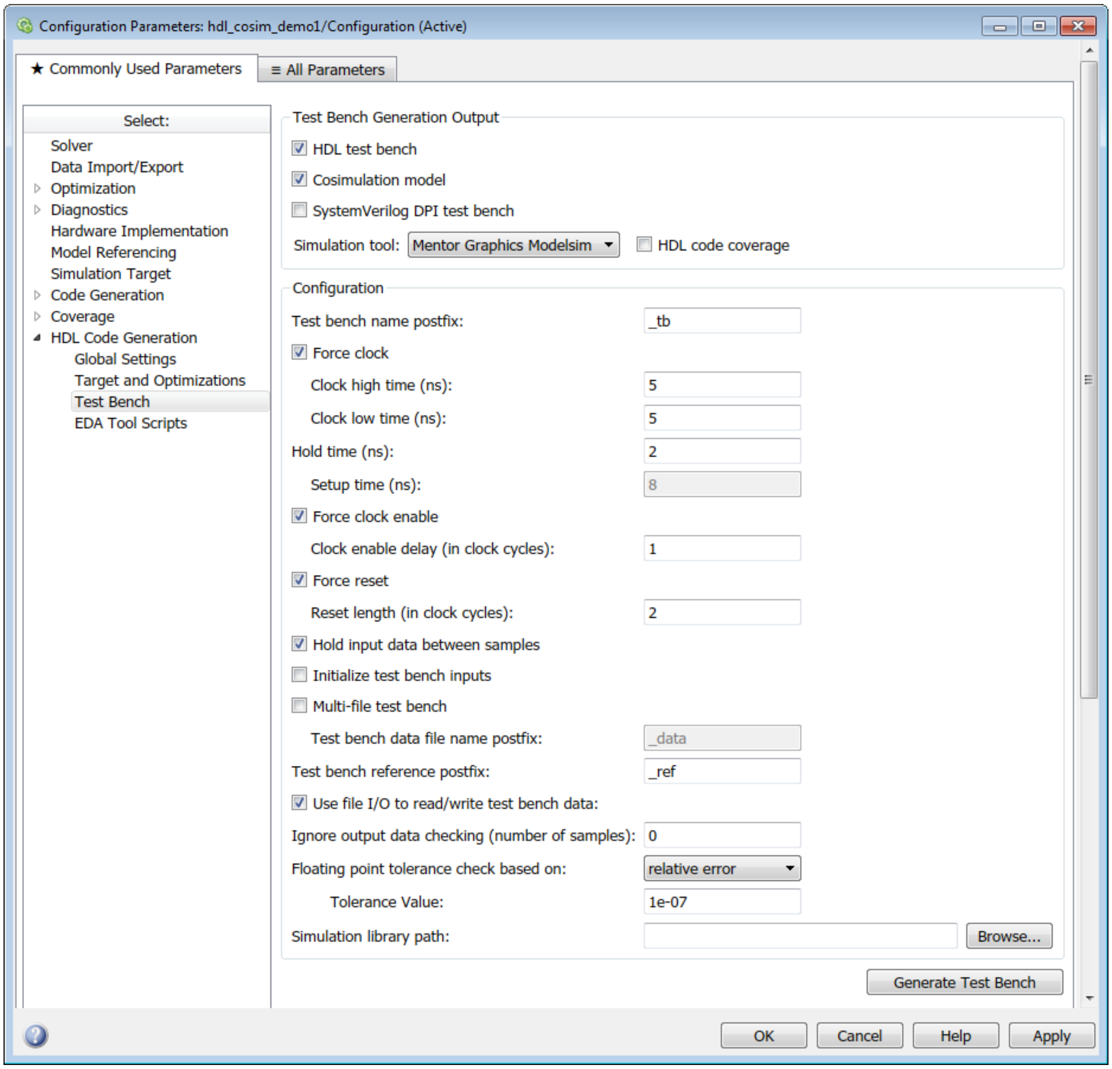
3 Click **Generate**. HDL Coder displays progress messages, as shown in the following listing:

```
### 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.


Configuration Parameters: hdl_cosim_demo1/Configuration (Active)

Test Bench Generation Output

- [✓] HDL test bench
- [✓] Cosimulation model
- [ ] SystemVerilog DPI test bench

Simulation tool: Mentor Graphics Modelsim [ ] HDL code coverage

Configuration

Test bench name postfix:	_tb
[✓] Force clock	
Clock high time (ns):	5
Clock low time (ns):	5
Hold time (ns):	2
Setup time (ns):	8
[✓] Force clock enable	
Clock enable delay (in clock cycles):	1
[✓] Force reset	
Reset length (in clock cycles):	2
[✓] Hold input data between samples	
[ ] Initialize test bench inputs	
[ ] Multi-file test bench	
Test bench data file name postfix:	_data
Test bench reference postfix:	_ref
[✓] Use file I/O to read/write test bench data:	
Ignore output data checking (number of samples):	0
Floating point tolerance check based on:	relative error
Tolerance Value:	1e-07
Simulation library path:	

3 Configure required test bench options. HDL Coder records option settings in a generated script file (see “The Cosimulation Script File” on page 18-56).

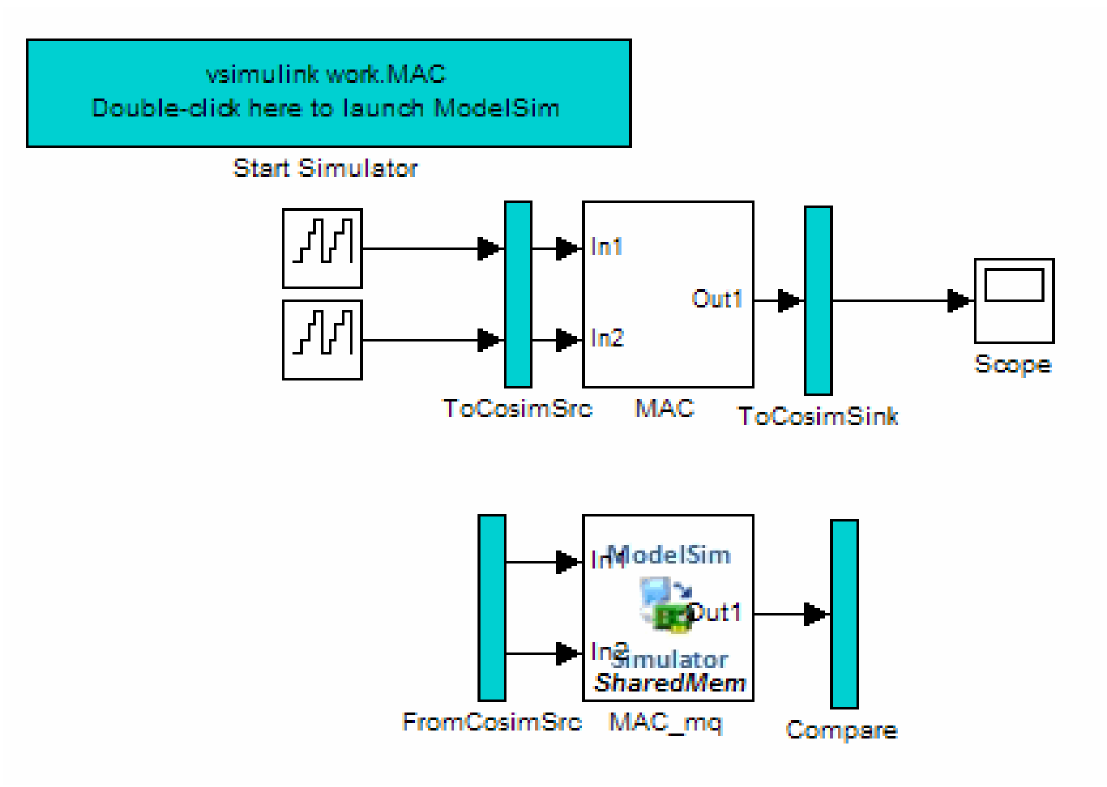
4 Click **Apply**.

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: hdlsrc\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 18-61.

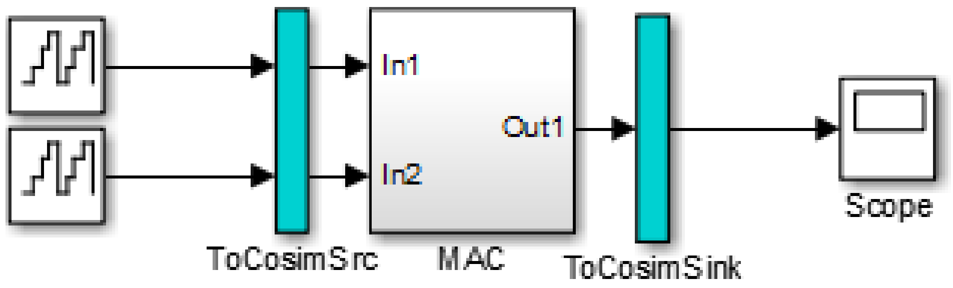
The next section, "Structure of the Generated Model" on page 18-46, describes the features of the model. Before running a cosimulation, become familiar with these features.

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

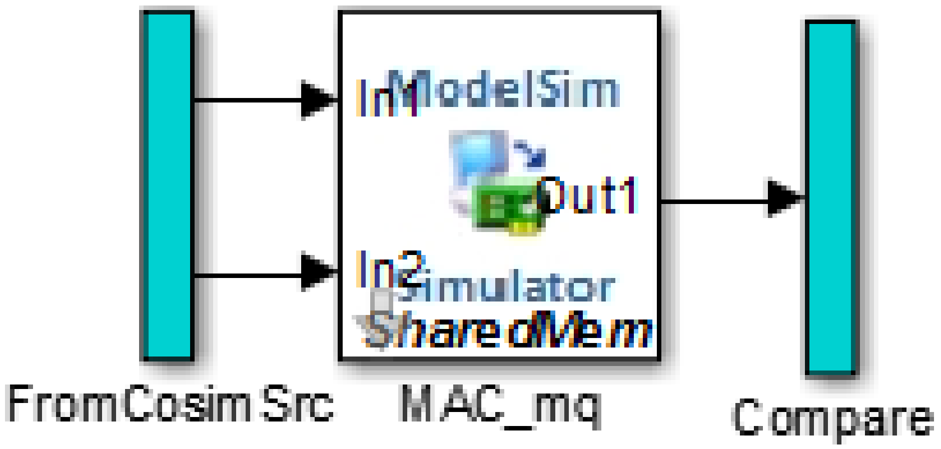
The model comprises two parallel signal paths. The *simulation path*, located in the upper half of the model window, is nearly identical to the original DUT. The purpose of the simulation path is to execute a normal Simulink simulation and provide a reference signal for comparison to the cosimulation 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 18-49).

Cosimulation Path

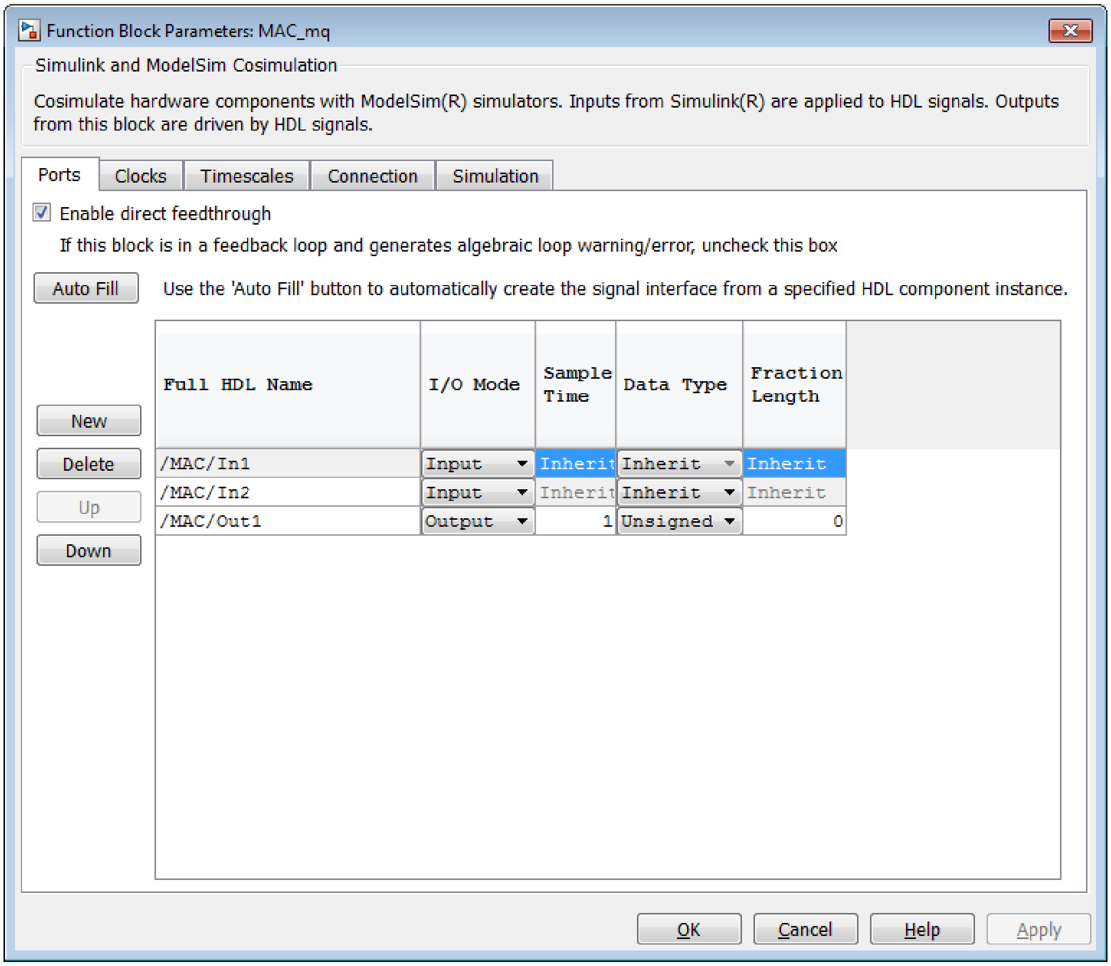
The *cosimulation path*, located in the lower half of the model window, contains the generated HDL Cosimulation block. The following figure shows the cosimulation path.



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 18-49).

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 18-51 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.


Function Block Parameters: MAC_mq

Simulink and ModelSim Cosimulation

Cosimulate hardware components with ModelSim(R) simulators. Inputs from Simulink(R) are applied to HDL signals. Outputs from this block are driven by HDL signals.

☑ Enable direct feedthrough

If this block is in a feedback loop and generates algebraic loop warning/error, uncheck this box

Use the 'Auto Fill' button to automatically create the signal interface from a specified HDL component instance.

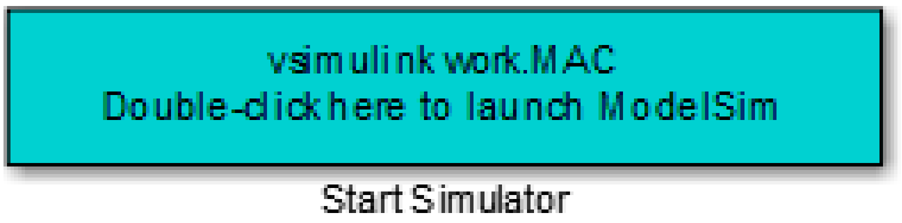
Full HDL Name	I/O Mode	Sample Time	Data Type	Fraction Length
/MAC/In1	Input	Inherit	Inherit	Inherit
/MAC/In2	Input	Inherit	Inherit	Inherit
/MAC/Out1	Output	1	Unsigned	0

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

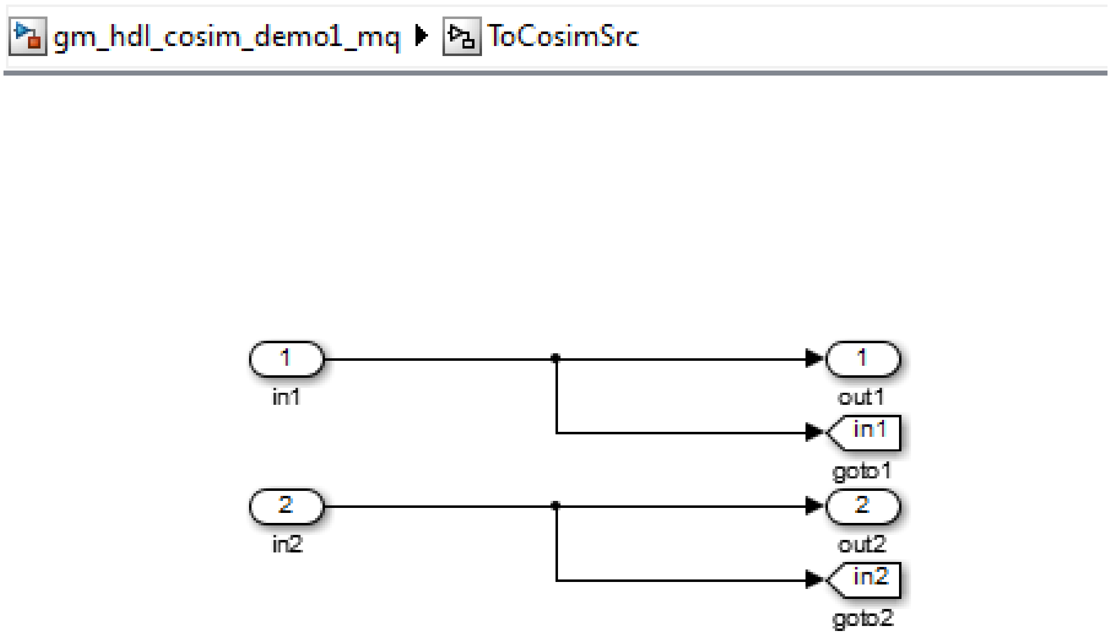
When you double-click the Start Simulator control, it launches the selected cosimulation tool and passes in a startup command to the tool. The Start Simulator icon displays the startup command, as shown in the following figure.

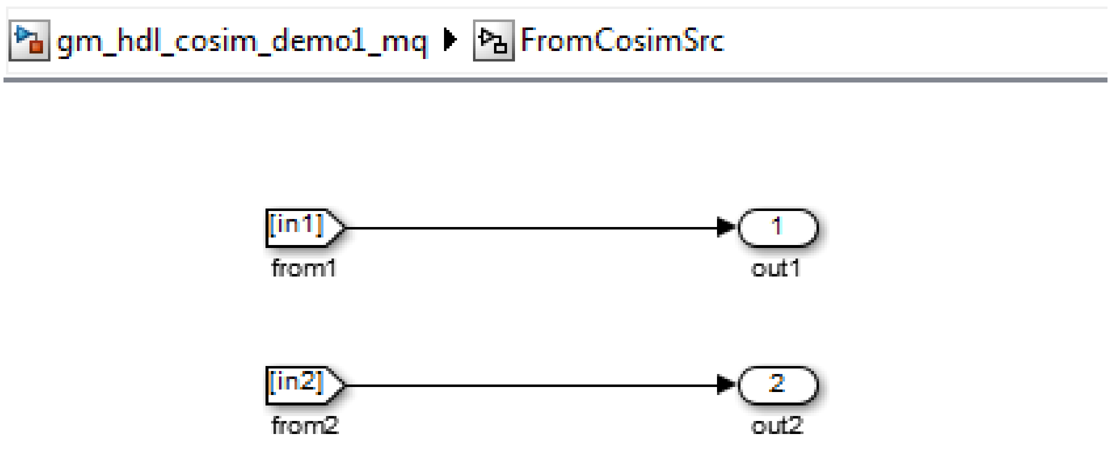


The commands executed when you double-click the Start Simulator icon launch and set up the cosimulation tool, but they do not start the actual cosimulation. “Launching a Cosimulation” on page 18-53 describes how to run a cosimulation with the generated model.

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.

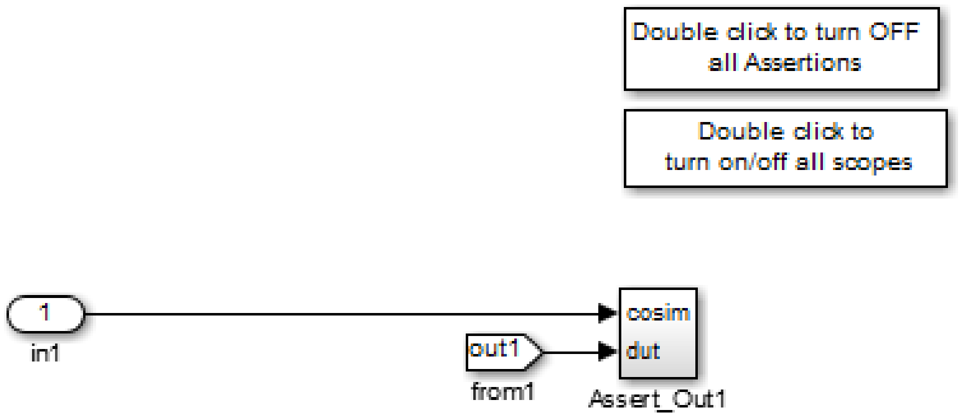

gm_hdl_cosim_demo1_mq
ToCosimSrc
1
in1
2
in2
1
out1
in1
goto1
2
out2
in2
goto2



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 18-58 for further information.

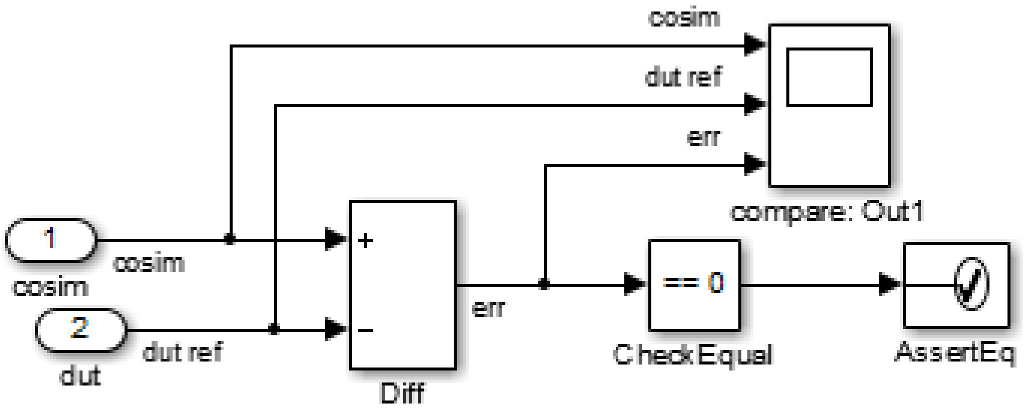
Controlling Assertions and Scope Displays

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.


Double click to turn OFF all Assertions
Double click to turn on/off all scopes
1
in1
out1
from1
cosim
dut
Assert_Out1

For each output of the DUT, HDL Coder generates an assertion checking subsystem (`Assert_Out`*N*). 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.

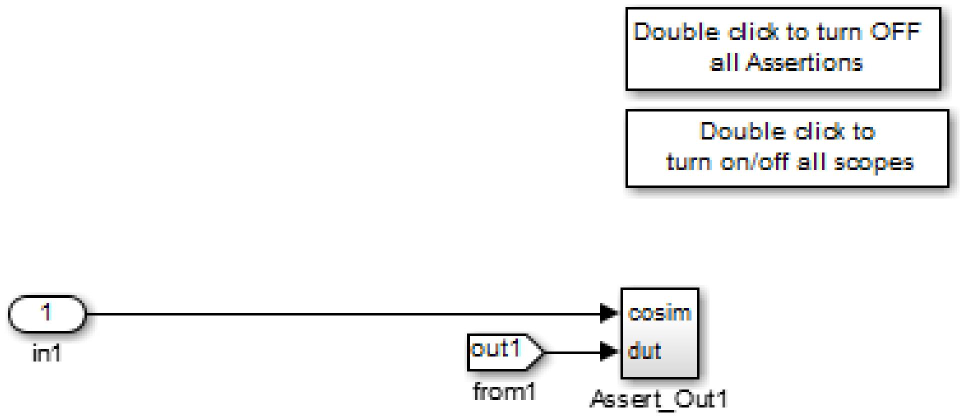
Tip: Assertion messages are warnings and do not stop simulation.

Launching a Cosimulation

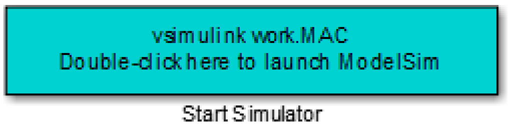
To run a cosimulation with the generated model:

1. 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 to turn OFF all Assertions
Double click to turn on/off all scopes
1
in1
out1
from1
cosim
dut
Assert_Out1

2 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.

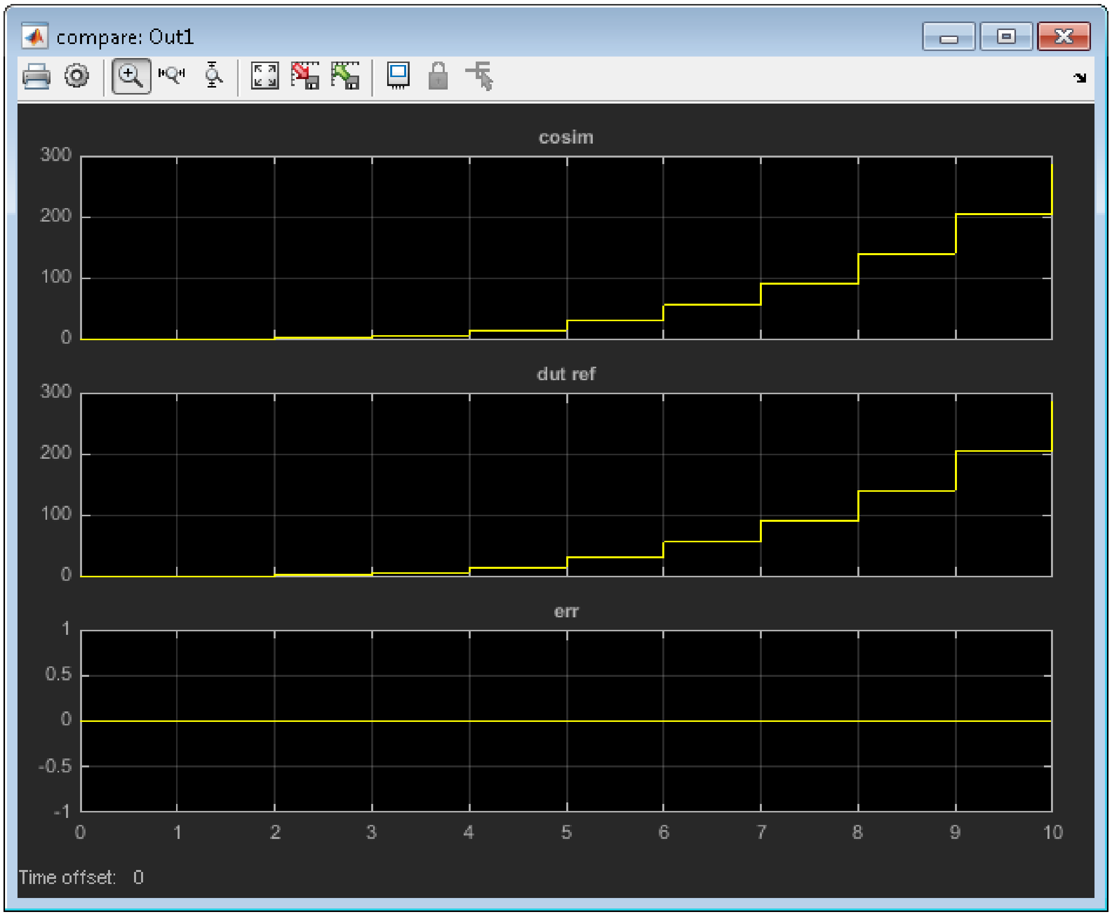
As the cosimulation runs, the HDL simulator displays messages like the following.

```
# 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:

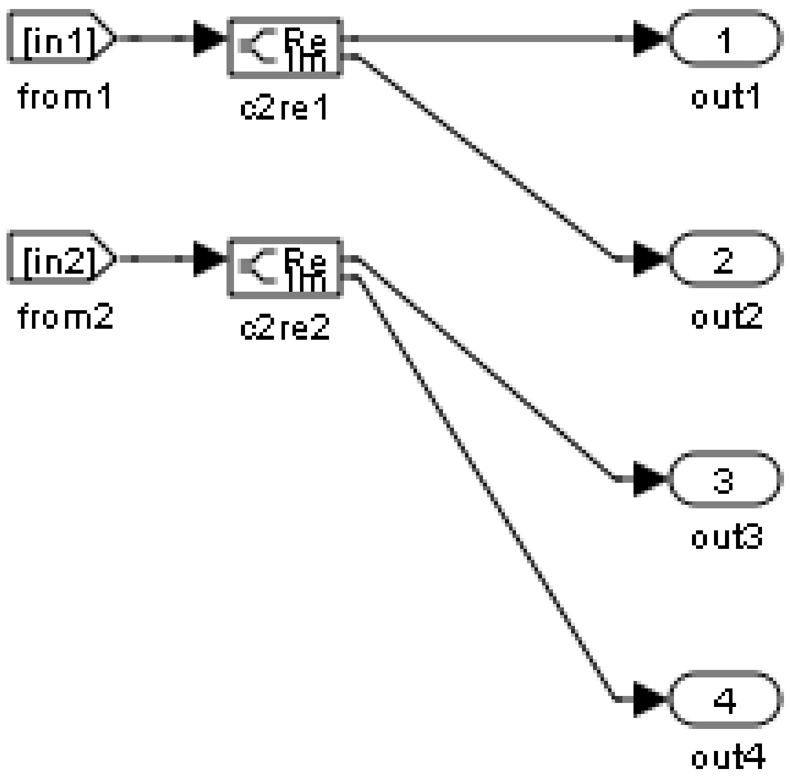
```
function tclCmds = gm_hdl_cosim_demo1_mq_tcl
tclCmds = {
    'do MAC_compile.do',...% Compile the generated code
    'vsimulink work.MAC',...% Initiate cosimulation
    'add wave  /MAC/clk',...% Add wave commands for chip input signals
    'add wave  /MAC/reset',...
    'add wave  /MAC/clk_enable',...
    'add wave  /MAC/In1',...
    'add wave  /MAC/In2',...
    'add wave  /MAC/ce_out',...% Add wave commands for chip output signals
    'add wave  /MAC/Out1',...
    'set 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

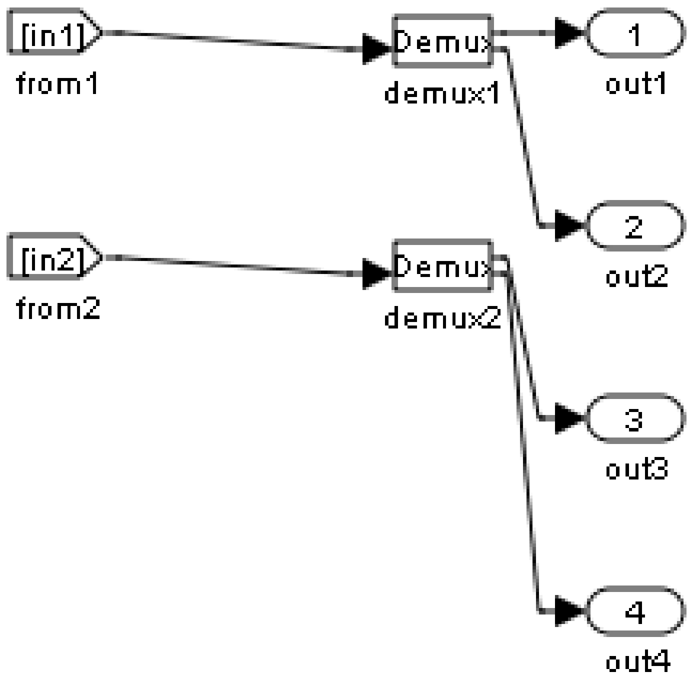
The generated cosimulation model automatically breaks complex inputs into real and imaginary parts. The following figure shows a `FromCosimSrc` subsystem that receives two complex input signals. The subsystem breaks the inputs into real and imaginary parts before passing them to the subsystem outputs.



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.
- 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 **Enable 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 12-24). 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.

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	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: • 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?

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	The update method only updates states. The output	The update method updates states and computes the output values.

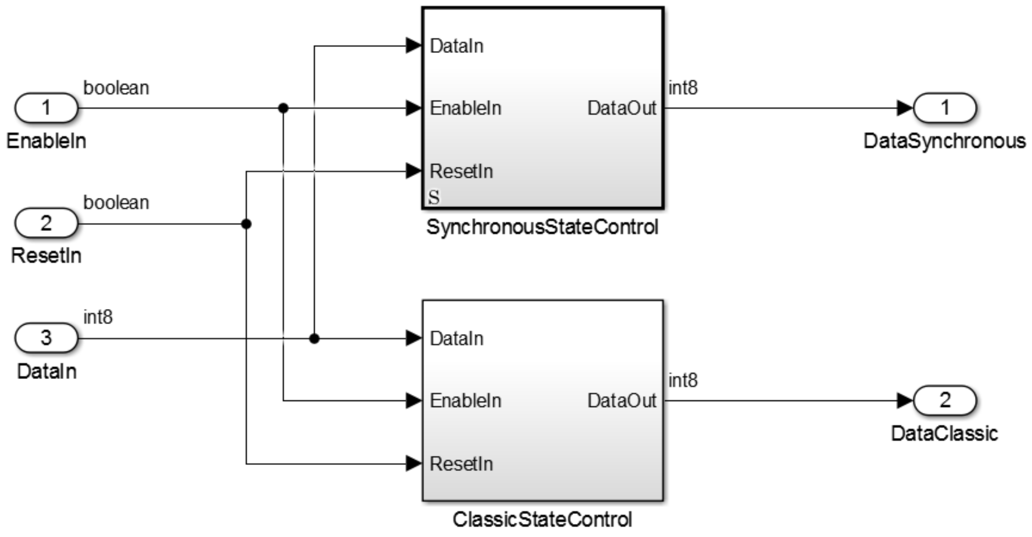
Functionality	Synchronous mode	Classic mode
• Initialize method: Initializes states. • Update method: Updates states. • Output method: Computes output values.	method computes the output values at each time step. For example, when you have enabled subsystems, the output value changes when the enable signal is low as it processes new input values. The output value matches the output from `Classic` mode when enable signal becomes high.	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.
HDL simulation behavior	More efficient on hardware.	Less efficient on hardware.
HDL code generation behavior	Generated HDL code is cleaner and uses fewer resources on hardware. For example, when you have enabled subsystems, HDL Coder does not generate bypass registers for each state update and uses fewer hardware resources.	Generated HDL code is not as clean and uses more hardware resources. For example, when you have enabled subsystems, HDL Coder generates bypass registers for each state update and uses more resources.

To learn more about when you can use the State Control block, see State Control.

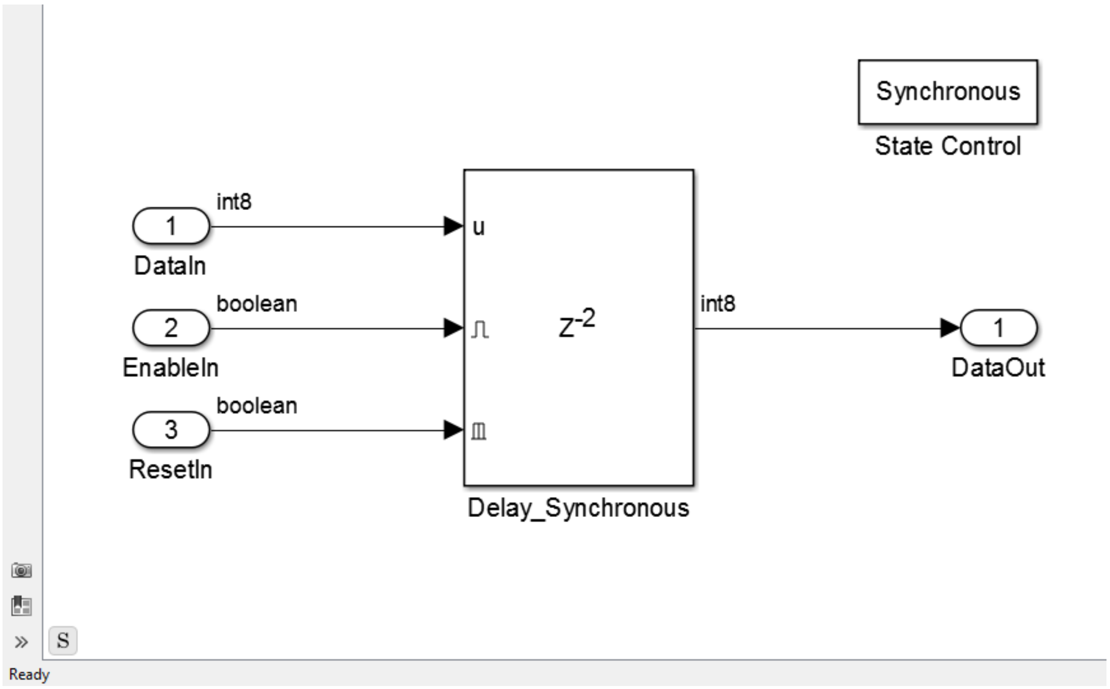
Synchronous Badge for Subsystems by Using Synchronous Mode

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`
	```
	`timescale 1 ns / 1 ns

	module SynchronousStateControl
	          (
	           clk,
	           reset,
	```
		```
		`timescale 1 ns / 1 ns

		module ClassicStateControl
		          (
		           clk,
		           reset,
		```

Functionality	Synchronous mode	Classic mode

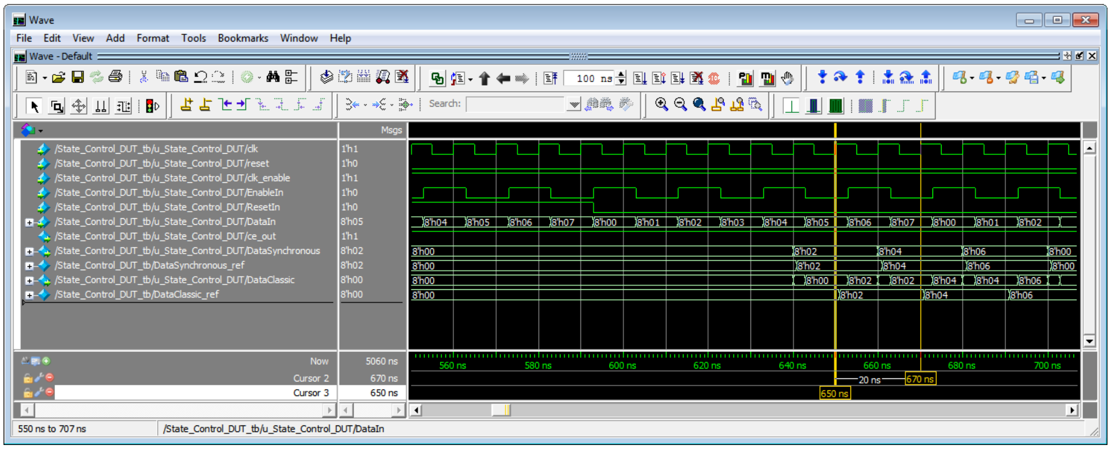
• **Reset type**: `Synchronous`	`          enb,`
	`          DataIn,`
	`          EnableIn,`
	`          ResetIn,`
	`          DataOut`
	`         );`
	
	`  input   clk;`
	`  input   reset;`
	`  input   enb;`
	`  input   signed [7:0] DataIn`
	`  input   EnableIn;`
	`  input   ResetIn;`
	`  output  signed [7:0] DataOut`
	
	`  reg signed [7:0] Delay_Synch`
	`  wire signed [7:0] Delay_Sync`
	`  wire signed [7:0] Delay_Sync`
	
	`  always @(posedge clk)`
	`    begin : Delay_Synchronous_`
	`      if (reset == 1'b1 || Res`
	`        Delay_Synchronous_reg[`
	`        Delay_Synchronous_reg[`
	`      end`
	`      else begin`
	`        if (enb && EnableIn) b`
	`          Delay_Synchronous_re`
	`          Delay_Synchronous_re`
	`        end`
	`      end`
	`    end`
	
	`  assign Delay_Synchronous_out`
	`  assign Delay_Synchronous_reg`
	`  assign Delay_Synchronous_reg`
	
	`  assign DataOut = Delay_Synch`
		`          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_byp`
		`  reg signed [7:0] Delay_Synchronous_reg int8 [2]`
		`  wire signed [7:0] Delay_Synchronous_by`
		`  wire signed [7:0] Delay_Synchronous_re`
		`  wire signed [7:0] Delay_Synchronous_de`
		`  wire signed [7:0] Delay_Synchronous_ou`
		
		`  always @(posedge clk)`
		`    begin : Delay_Synchronous_process`
		`      if (reset == 1'b1 || ResetIn == 1'`
		`        Delay_Synchronous_bypass <= 8'sb`
		`        Delay_Synchronous_reg[0] <= 8'sb`
		`        Delay_Synchronous_reg[1] <= 8'sb next[0];`
		`      end                         next[1];`
		`      else begin`
		`        if (enb && EnableIn) begin`
		`          Delay_Synchronous_bypass <= De`
		`          Delay_Synchronous_reg[0] <= De`
		`          Delay_Synchronous_reg[1] <= De`
		`        end`
		`      end                         eg[0];`
		`    end`
		
		`  assign Delay_Synchronous_delay_out = (`
		`            Delay_Synchronous_reg[1]);`

Functionality	Synchronous mode	Classic mode

	`endmodule  // SynchronousState`
	
	 • 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.
		`  assign Delay_Synchronous_out1 = (Enabl`
		 `             Delay_Synchronous_bypass);`
		 `  assign Delay_Synchronous_bypass_next =`
		 `  assign Delay_Synchronous_reg_next[0] =`
		 `  assign Delay_Synchronous_reg_next[1] =`
		
		 `  assign DataOut = Delay_Synchronous_out`
		
		 `endmodule  // ClassicStateControl`
		
		 • 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.


Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Search:
Msgs
/State_Control_DUT_tb/u_State_Control_DUT/clk 1'h1
/State_Control_DUT_tb/u_State_Control_DUT/reset 1'h0
/State_Control_DUT_tb/u_State_Control_DUT/clk_enable 1'h1
/State_Control_DUT_tb/u_State_Control_DUT/EnableIn 1'h0
/State_Control_DUT_tb/u_State_Control_DUT/ResetIn 1'h0
/State_Control_DUT_tb/u_State_Control_DUT/DataIn 8'h05
/State_Control_DUT_tb/u_State_Control_DUT/ce_out 1'h1
/State_Control_DUT_tb/u_State_Control_DUT/DataSynchronous 8'h02
/State_Control_DUT_tb/DataSynchronous_ref 8'h02
/State_Control_DUT_tb/u_State_Control_DUT/DataClassic 8'h00
/State_Control_DUT_tb/DataClassic_ref 8'h00
Now 5060 ns
Cursor 2 670 ns
Cursor 3 650 ns
560 ns
580 ns
600 ns
620 ns
640 ns
660 ns
680 ns
700 ns
20 ns
670 ns
650 ns
550 ns to 707 ns
/State_Control_DUT_tb/u_State_Control_DUT/DataIn

- 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 “HDL Code Generation from a Simulink Model”.

See Also

State Control

Stateflow HDL Code Generation Support

Introduction to Stateflow HDL Code Generation

In this section...

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.

Example

The hdlcodercfir model shows how to generate HDL code for a subsystem that includes Stateflow charts.

To open the model, at the command line, enter:

```
hdlcodercfir
```

Restrictions

HDL Coder does not support Stateflow blocks that contain messages for HDL code generation.

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.

Generate HDL for Mealy and Moore Finite State Machines

In this section...

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.

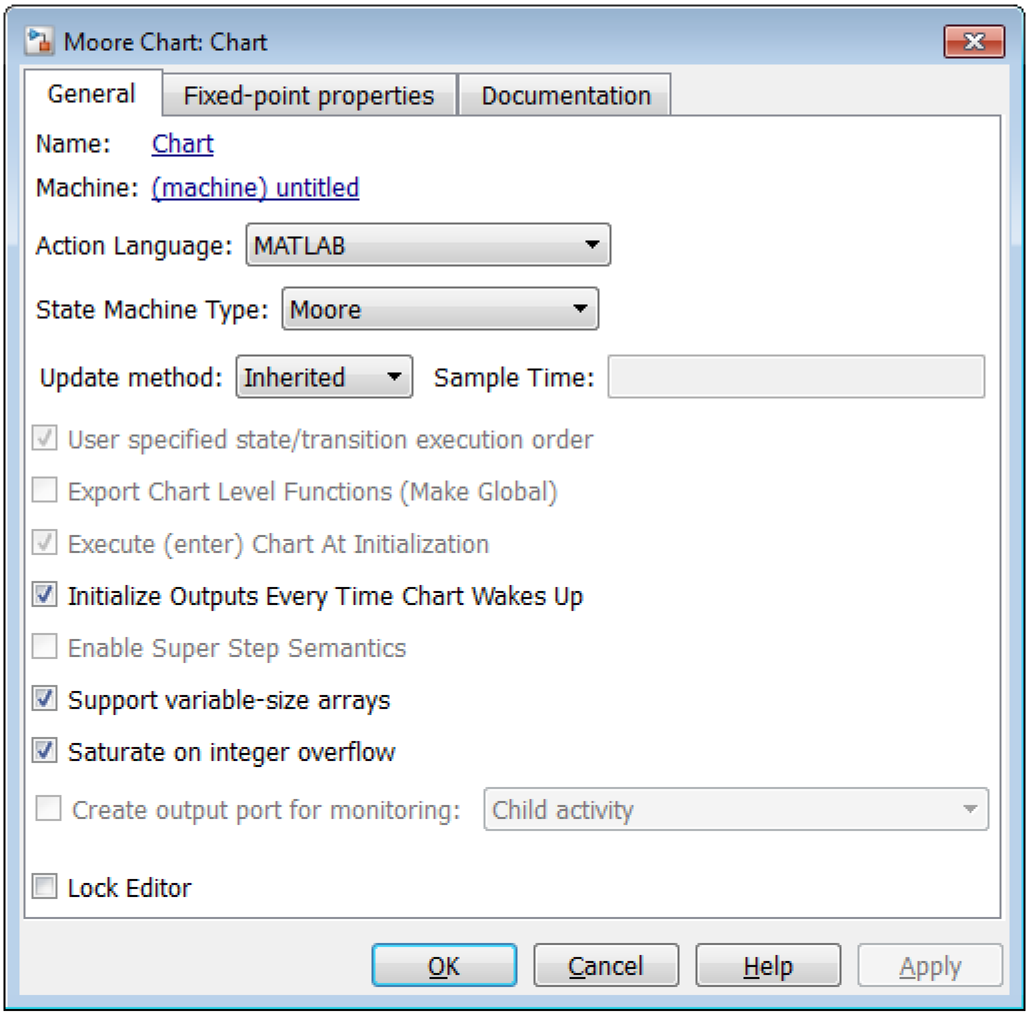
Generating HDL Code for a Moore Finite State Machine

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.


Moore Chart: Chart

Name: Chart

Machine: (machine) untitled

Action Language: MATLAB

State Machine Type: Moore

Update method: Inherited Sample Time:

[✓] User specified state/transition execution order

[ ] Export Chart Level Functions (Make Global)

[✓] Execute (enter) Chart At Initialization

[✓] Initialize Outputs Every Time Chart Wakes Up

[ ] Enable Super Step Semantics

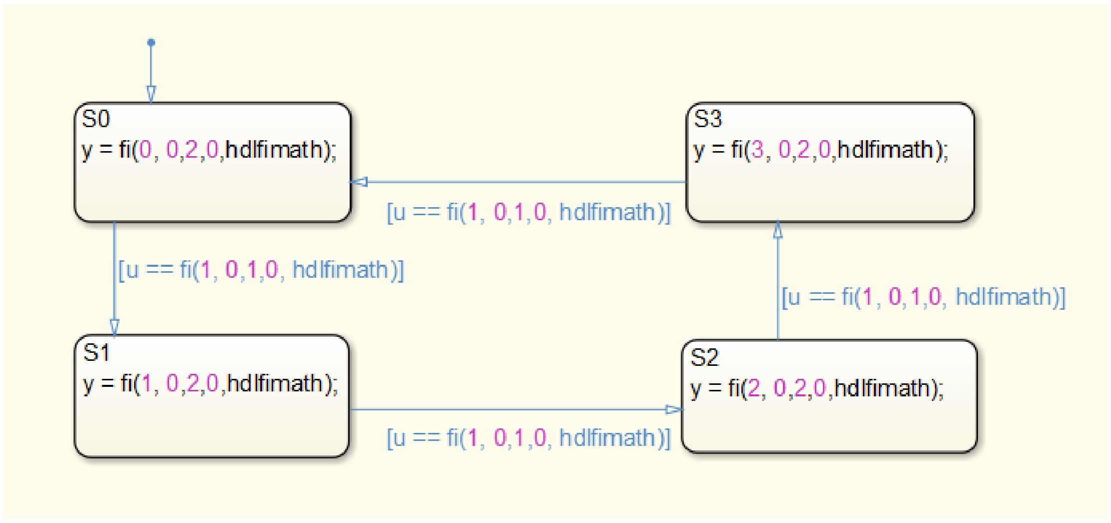
[✓] Support variable-size arrays

[✓] Saturate on integer overflow

[ ] Create output port for monitoring: Child activity

[ ] Lock Editor

The following figure shows a Stateflow chart of a Moore state machine that uses MATLAB as the action language.


S0
y = fi(0, 0,2,0,hdlfimath);
S3
y = fi(3, 0,2,0,hdlfimath);
[u == fi(1, 0,1,0, hdlfimath)]
[u == fi(1, 0,1,0, hdlfimath)]
[u == fi(1, 0,1,0, hdlfimath)]
S1
y = fi(1, 0,2,0,hdlfimath);
S2
y = fi(2, 0,2,0,hdlfimath);
[u == fi(1, 0,1,0, hdlfimath)]

The Verilog code generated for the Moore chart:

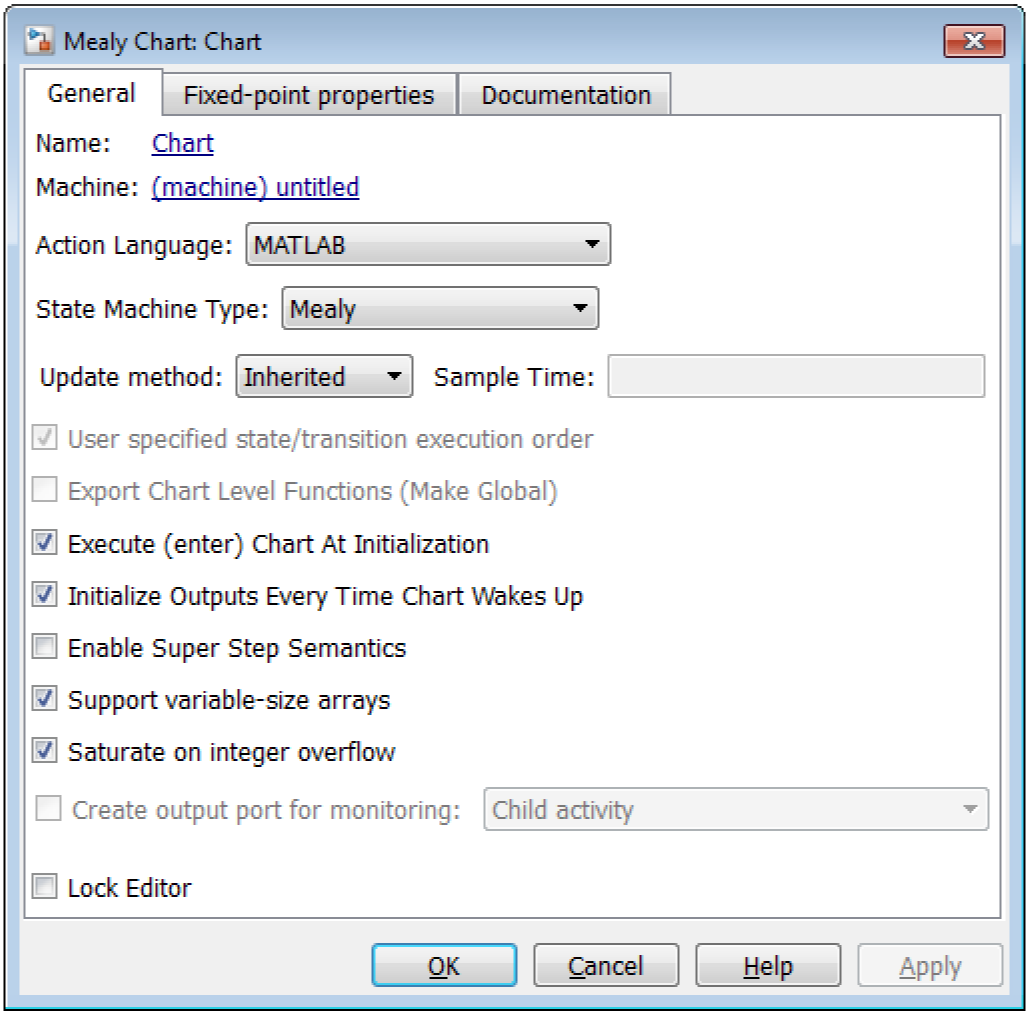
```
always @(posedge clk or posedge reset)
  begin : Moore_Chart_1_process
    if (reset == 1'b1) begin
      is_Moore_Chart <= is_Moore_Chart_IN_S0;
    end
    else begin
      if (enb) begin
        case ( is_Moore_Chart)
          is_Moore_Chart_IN_S0 :
            begin
              if (u == 8'sb00000001) 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 Chart: Chart

General | Fixed-point properties | Documentation

Name: Chart

Machine: (machine) untitled

Action Language: MATLAB

State Machine Type: Mealy

Update method: Inherited   Sample Time:

[✓] User specified state/transition execution order

[ ] Export Chart Level Functions (Make Global)

[✓] Execute (enter) Chart At Initialization

[✓] Initialize Outputs Every Time Chart Wakes Up

[ ] Enable Super Step Semantics

[✓] Support variable-size arrays

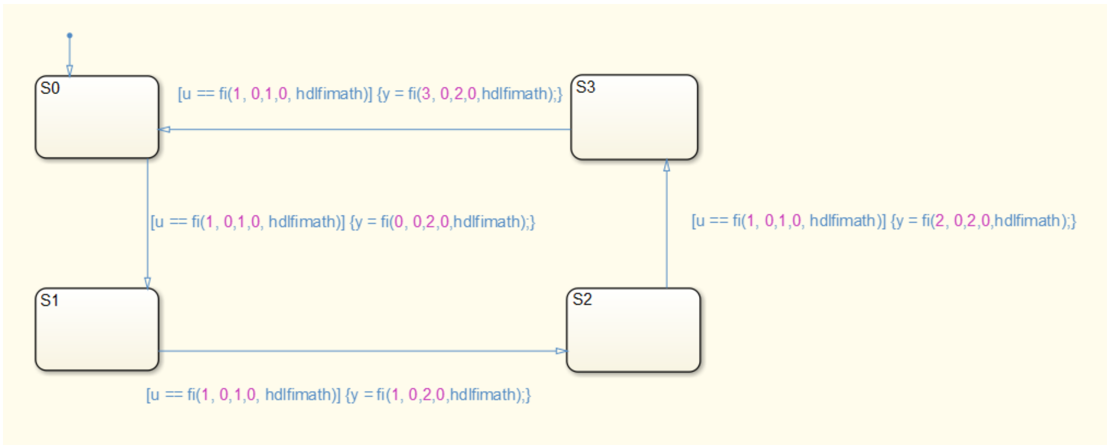
[✓] Saturate on integer overflow

[ ] Create output port for monitoring: Child activity

[ ] Lock Editor

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.


S0
[u == fi(1, 0,1,0, hdlfimath)] {y = fi(3, 0,2,0,hdlfimath);}
S3
[u == fi(1, 0,1,0, hdlfimath)] {y = fi(0, 0,2,0,hdlfimath);}
[u == fi(1, 0,1,0, hdlfimath)] {y = fi(2, 0,2,0,hdlfimath);}
S1
S2
[u == fi(1, 0,1,0, hdlfimath)] {y = fi(1, 0,2,0,hdlfimath);}

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 19-13

Design Patterns Using Advanced Chart Features

In this section...

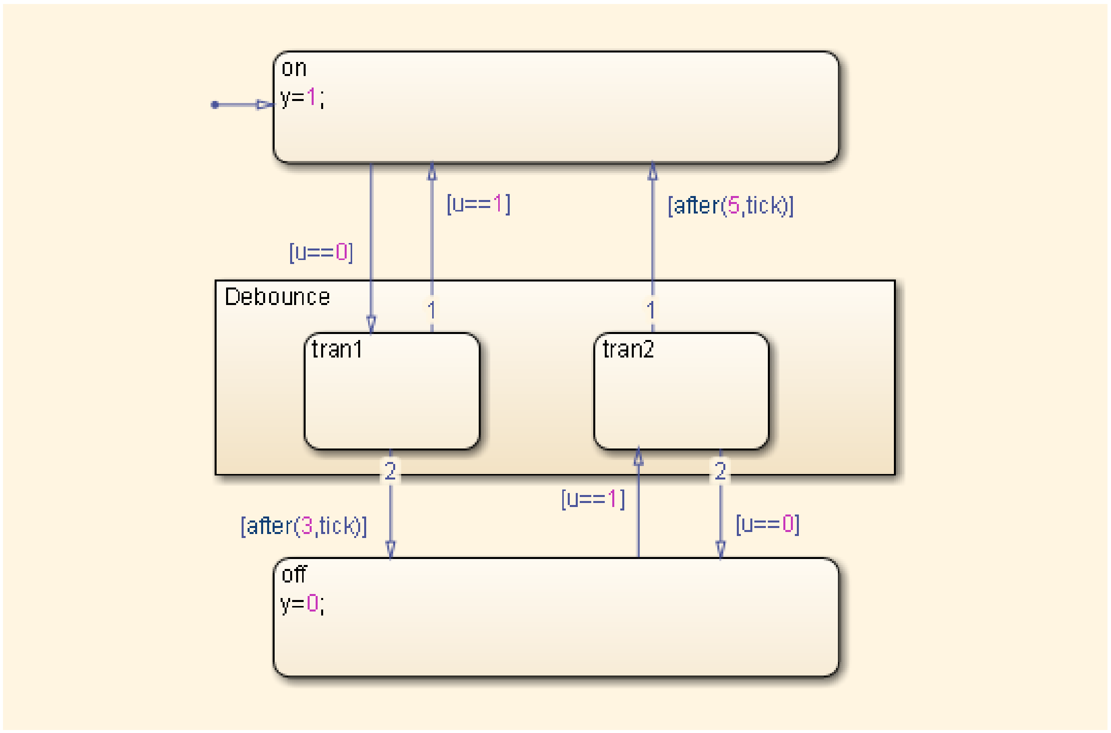
Temporal Logic

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.

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

For detailed information about temporal logic, see "Control Chart Execution 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.



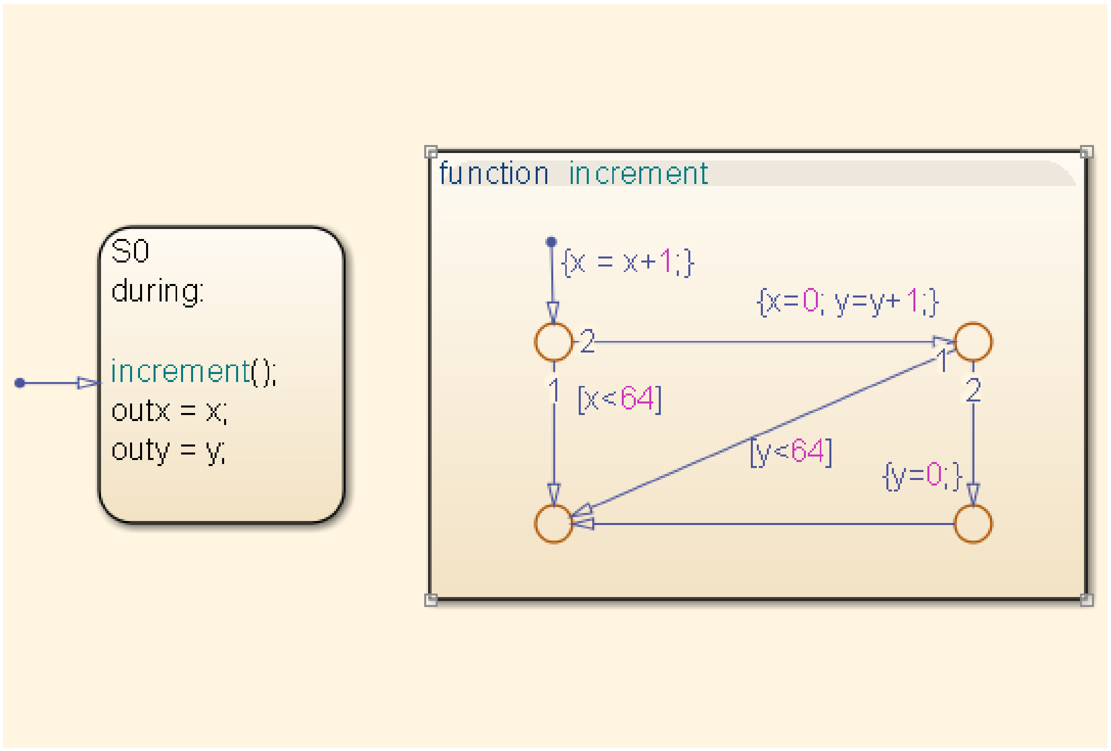
The following code excerpt shows VHDL code generated from this chart.

```
Chart_1_output : PROCESS (is_Chart, u, temporalCounter_i1, y_reg)
    VARIABLE temporalCounter_i1_temp : unsigned(7 DOWNTO 0);
  BEGIN
    temporalCounter_i1_temp := temporalCounter_i1;
    is_Chart_next <= is_Chart;
    y_reg_next <= y_reg;
    IF temporalCounter_i1 < 7 THEN
      temporalCounter_i1_temp := temporalCounter_i1 + 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_i1_temp >= 3 THEN
          is_Chart_next <= IN_off;
          y_reg_next <= 0.0;
        END IF;
      WHEN IN_tran2 =>
        IF temporalCounter_i1_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_i1_temp := to_unsigned(0, 8);
        END IF;
      WHEN OTHERS =>
        IF u = 0.0 THEN
          is_Chart_next <= IN_tran1;
          temporalCounter_i1_temp := to_unsigned(0, 8);
        END IF;
    END CASE;

    temporalCounter_i1_next <= temporalCounter_i1_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 “Stateflow Hierarchy of 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.

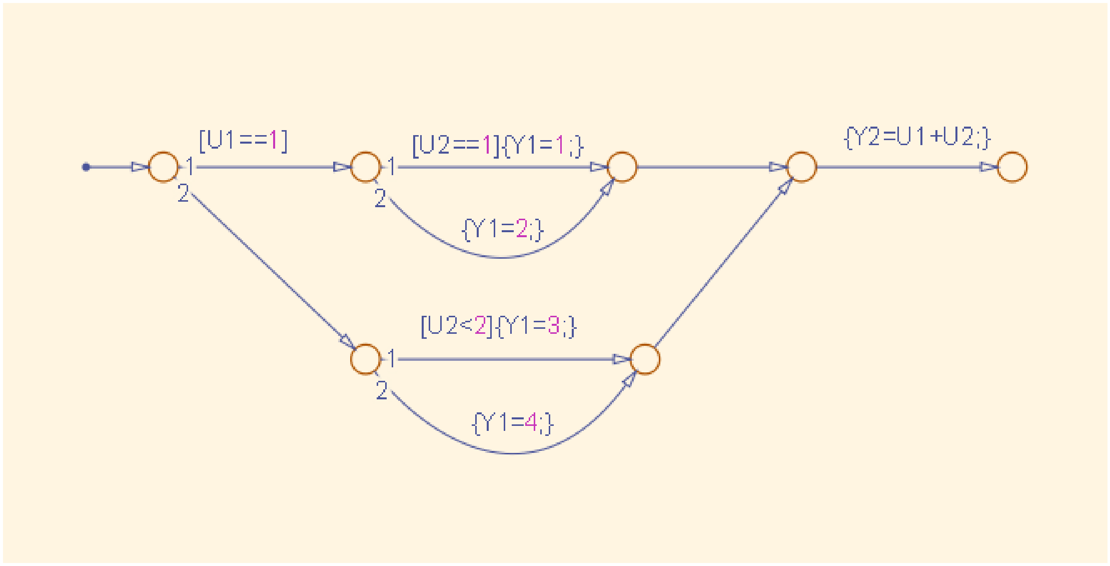
Stateless Charts

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

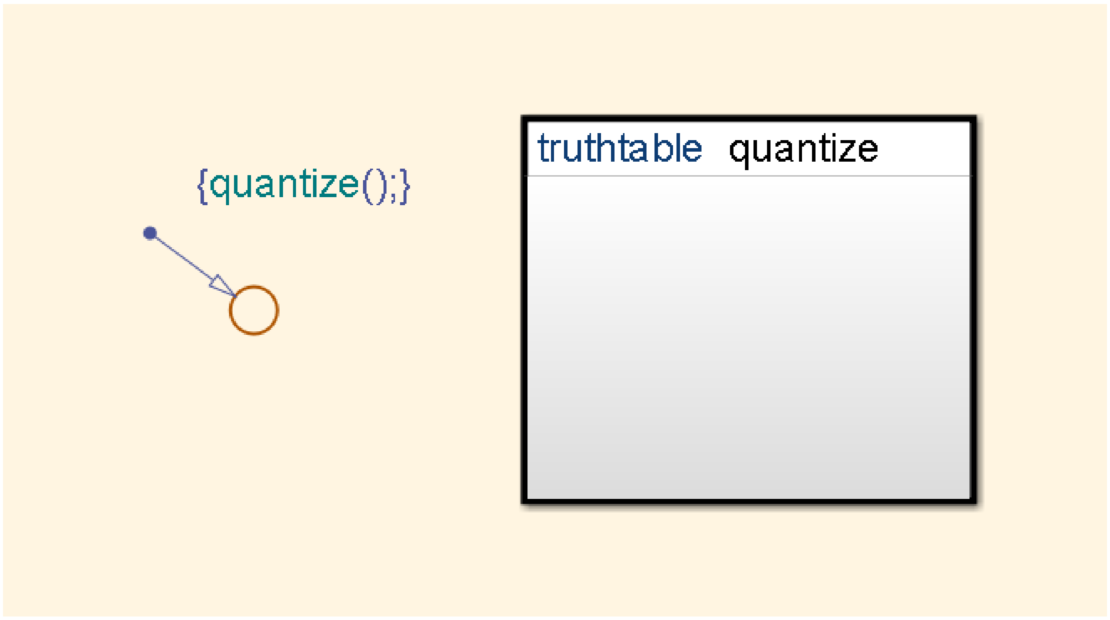
This section examines a Truth Table function in a chart, and the VHDL code generated for the chart.

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:

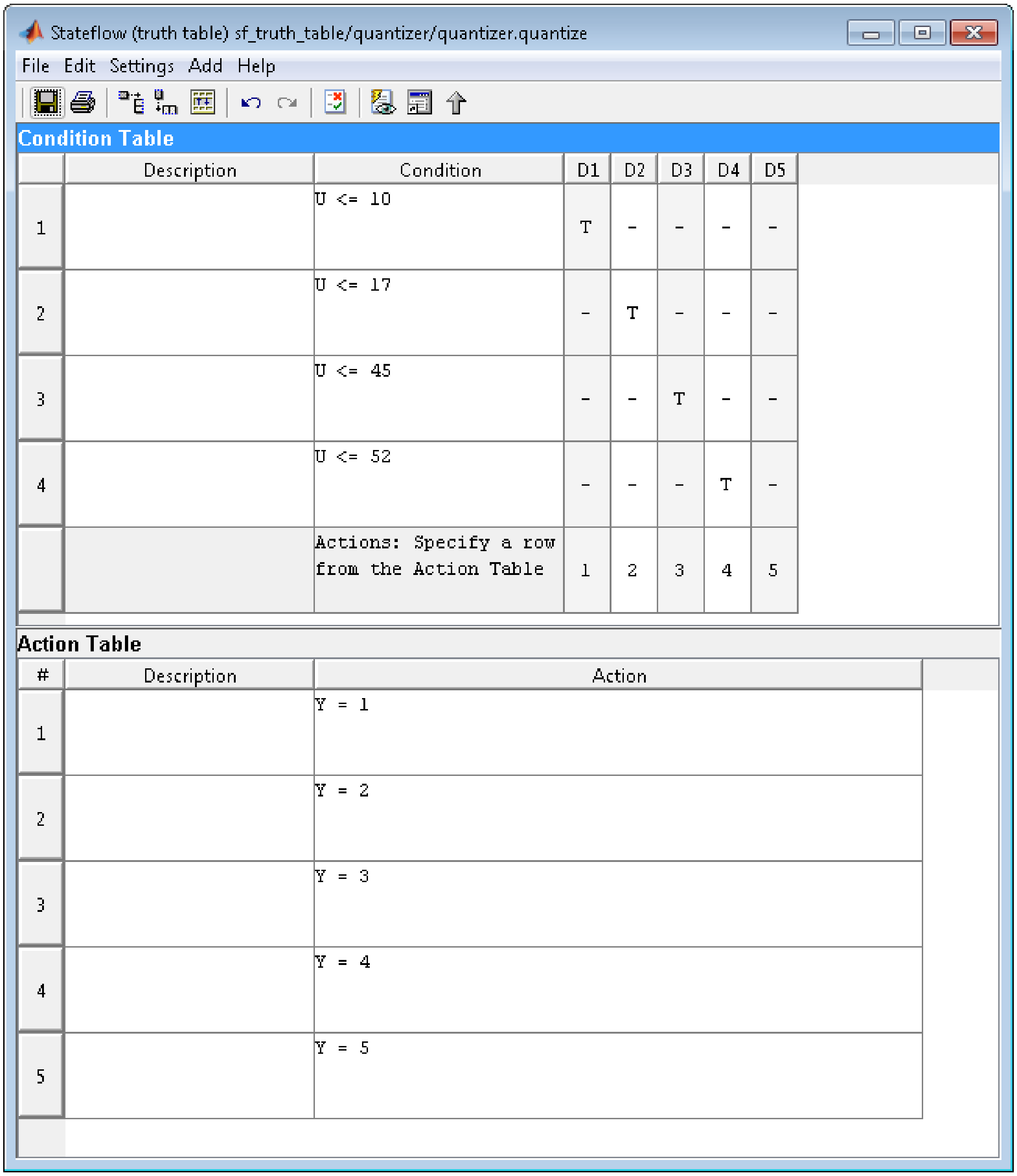
```
Y = 1 when 0  <= U <= 10
Y = 2 when 10 <  U <= 17
Y = 3 when 17 <  U <= 45
Y = 4 when 45 <  U <= 52
Y = 5 when 52 <  U
```

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.


Stateflow (truth table) sf_truth_table/quantizer/quantizer.quantize
File Edit Settings Add Help

Condition Table

	Description	Condition	D1	D2	D3	D4	D5
1		U <= 10	T	-	-	-	-
2		U <= 17	-	T	-	-	-
3		U <= 45	-	-	T	-	-
4		U <= 52	-	-	-	T	-
		Actions: Specify a row from the Action Table	1	2	3	4	5

Action Table

#	Description	Action
1		Y = 1
2		Y = 2
3		Y = 3
4		Y = 4
5		Y = 5

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).

Generating HDL Code with the MATLAB Function Block

HDL Applications for the MATLAB Function Block

The MATLAB Function block contains a MATLAB function in a model. The function's inputs and outputs are represented by 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, the HDL Coder software generates two main HDL code 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.

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 deinterleavers
- Modulators and demodulators
- Multipath channel models; impairment models
- Timing recovery algorithms
- Viterbi algorithm; Maximum Likelihood Sequence Estimation (MLSE)
- Adaptive equalizer algorithms

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
```

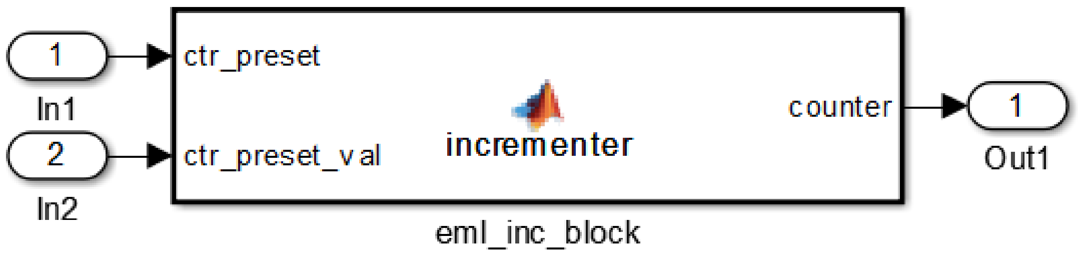
Code Generation from a MATLAB Function Block

In this section...

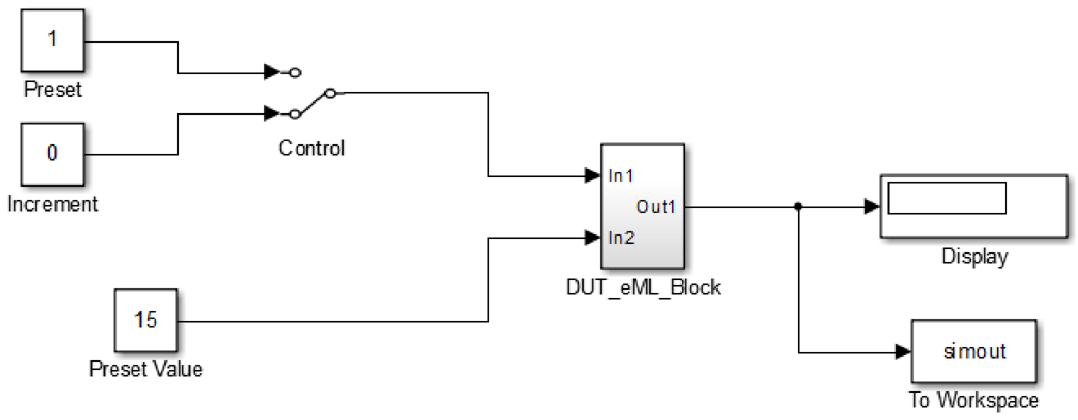
Counter Model Using the MATLAB Function block

In this tutorial, you construct and configure a simple model, `eml_hdl_incrementer_tut`, and then generate VHDL code from the model. `eml_hdl_incrementer_tut` includes a MATLAB Function block that implements a simple fixed-point counter function, `incrementer`. The `incrementer` 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_incrementer
```

After you open the model, save a copy of it to your local folder as `eml_hdl_incrementer_tut`.

The Incrementer Function Code

The following code listing gives the complete `incrementer` function definition:

```
function counter = incrementer(ctr_preset, ctr_preset_val)
% The function incrementer 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);
    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 subfolders 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.m` command. The `hdlsetup` command uses the `set_param` function to set up models for HDL code generation quickly and consistently. See “Initializing Model Parameters with hdlsetup” for further information about `hdlsetup`.

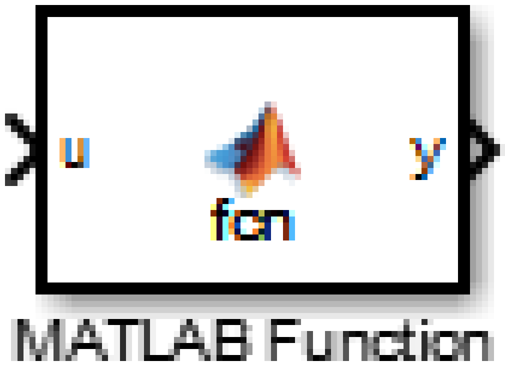
To set the model parameters:

1. Create a new model.
2. Save the model as `eml_hdl_incrementer_tut`.
3. At the MATLAB command prompt, type:

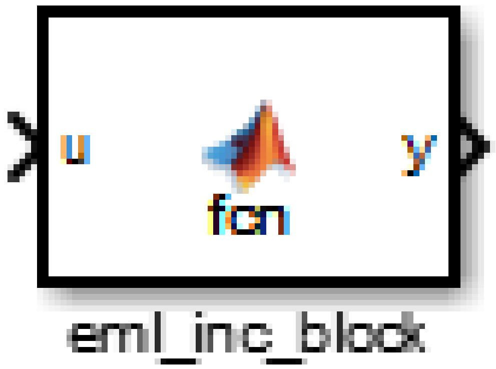
   ```
   hdlsetup('eml_hdl_incrementer_tut');
   ```
4. Open the Configuration Parameters dialog box.
5. Set the following **Solver** options, which are useful in simulating this model:
   - **Fixed step size**: 1
   - **Stop time**: 5
6. Click **OK** to save your changes and close the Configuration Parameters 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.

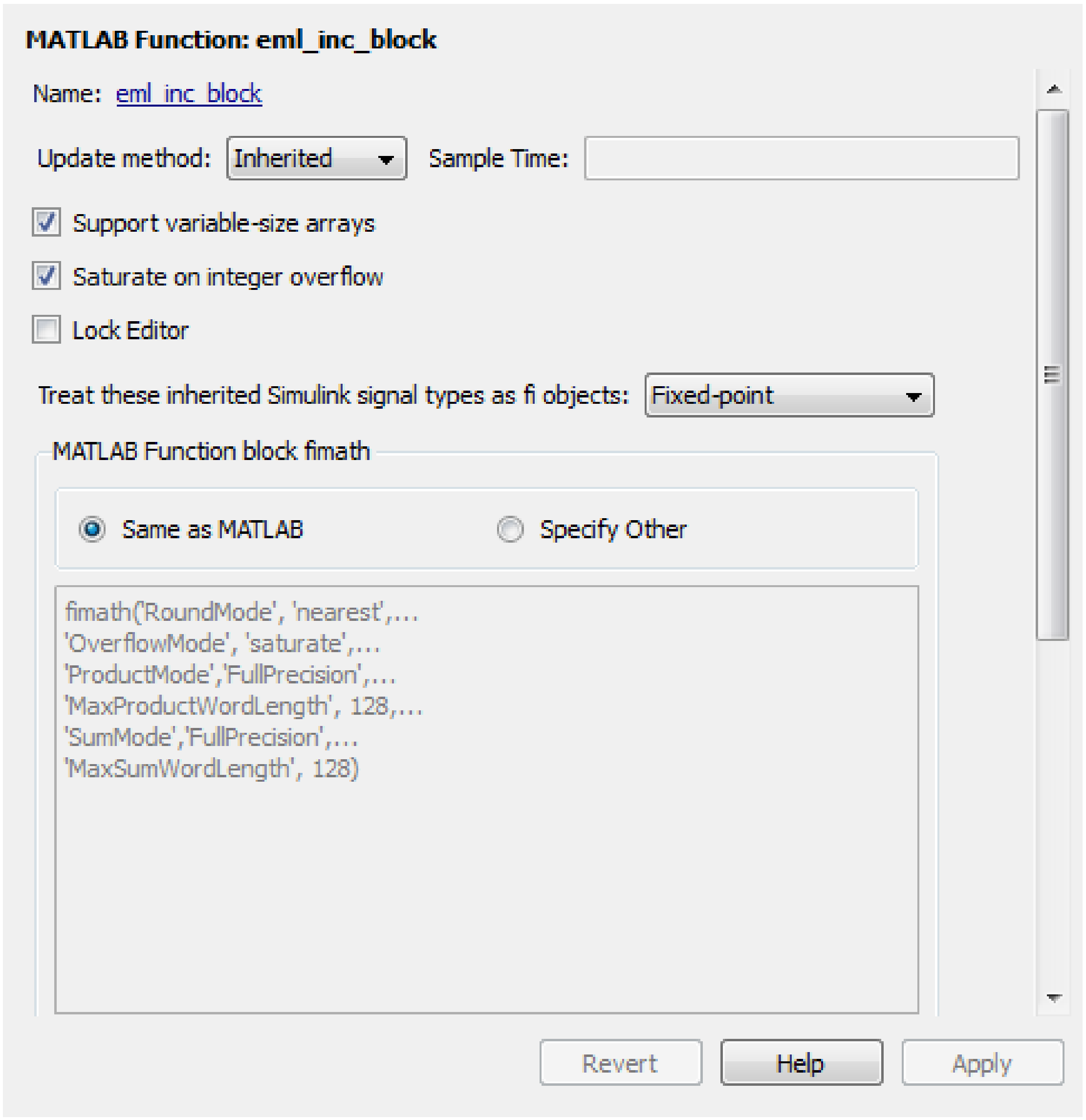
Set Fixed-Point Options for the MATLAB Function Block

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_incrementer_tut` model that you created in “Adding a MATLAB Function Block to the Model” on page 20-8.
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.



MATLAB Function: eml_inc_block

Name: eml_inc_block

Update method: Inherited
Sample Time:

[✓] Support variable-size arrays

[✓] Saturate on integer overflow

[ ] Lock Editor

Treat these inherited Simulink signal types as fi objects: Fixed-point

MATLAB Function block fimath

[● Same as MATLAB] ○ Specify Other

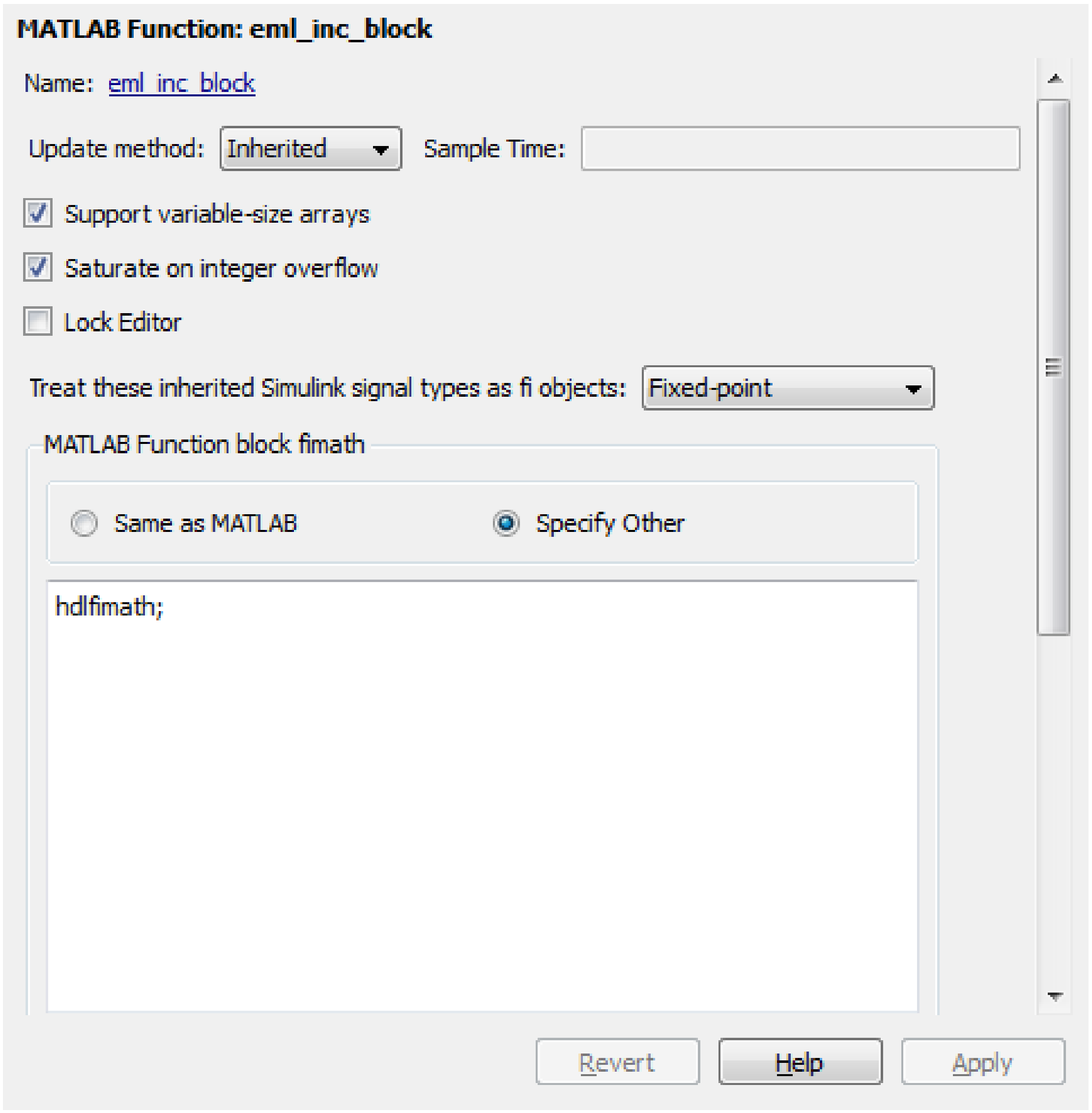
```
fimath('RoundMode', 'nearest',...
'OverflowMode', 'saturate',...
'ProductMode','FullPrecision',...
'MaxProductWordLength', 128,...
'SumMode','FullPrecision',...
'MaxSumWordLength', 128)
```

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.


MATLAB Function: eml_inc_block

Name: eml_inc_block

Update method: Inherited Sample Time:

[✓] Support variable-size arrays

[✓] Saturate on integer overflow

[ ] Lock Editor

Treat these inherited Simulink signal types as fi objects: Fixed-point

MATLAB Function block fimath

Same as MATLAB [✓ Specify Other]

```
hdlfimath;
```

7 Close the Ports and Data Manager.

8 Save the model.

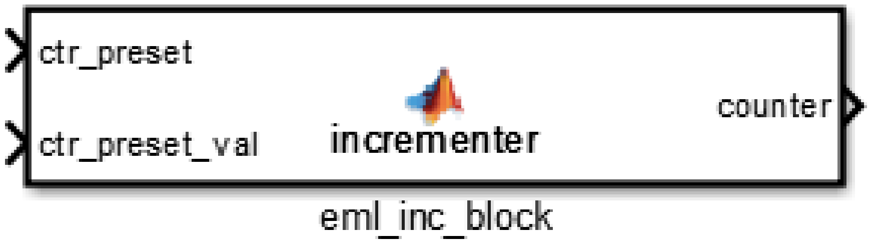
Programming the MATLAB Function Block

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_incrementer_tut` model that you created in “Adding a MATLAB Function Block to the Model” on page 20-8.
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 20-6, 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 20-14 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.

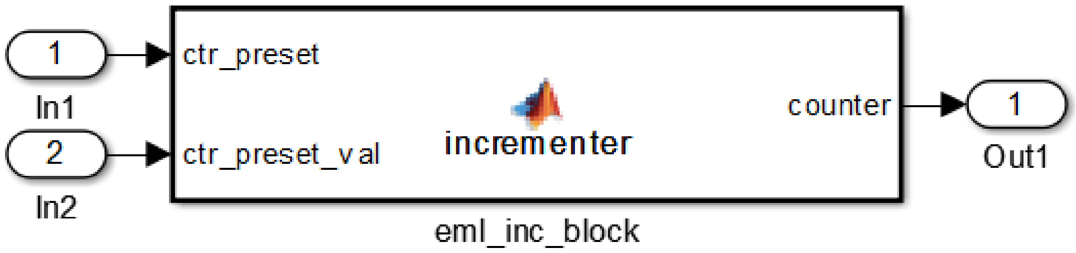
2 Select **Diagram** > **Subsystem & Model Reference** > **Create Subsystem from Selection**.

  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_incrementer_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.

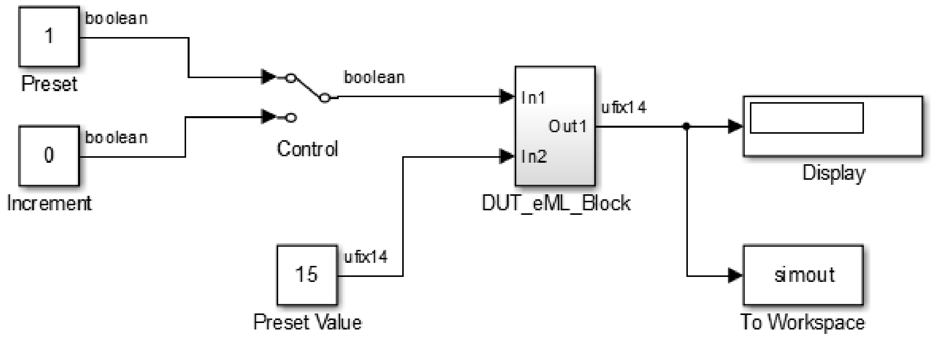
The build process builds an S-function for use in simulation. The build process includes generation of C code for the S-function. The code generation messages you see during the build process refer to generation of C code, not HDL code generation.

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. Select **Display** > **Signals & Ports** > **Port Data Types**.
2. Select **Simulation** > **Update Diagram** (or press **Ctrl+D**) to compile the model. This triggers a rebuild of the code. After the model compiles, the block diagram updates to show the port data types.

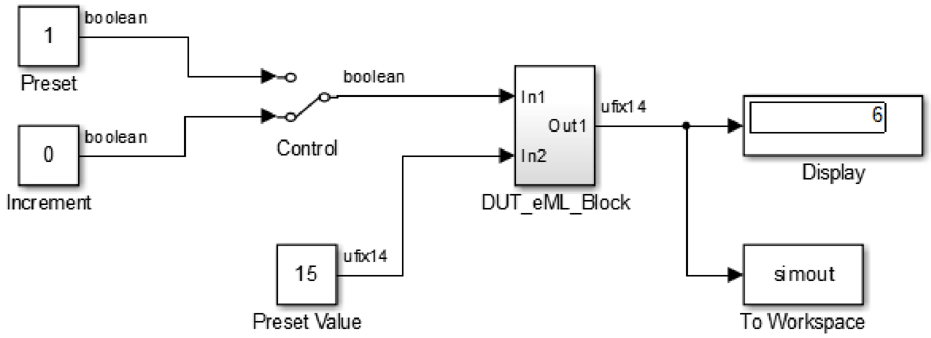


3 Save the model.

Simulating the eml_hdl_incrementer_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_incrementer_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_incrementer_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 16-48).

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.

Generate Instantiable Code for Functions

In this section...
"How To Generate Instantiable Code for Functions" on page 20-22
"Generate Code Inline for Specific Functions" on page 20-23
"Limitations for Instantiable Code Generation for Functions" on page 20-23

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.

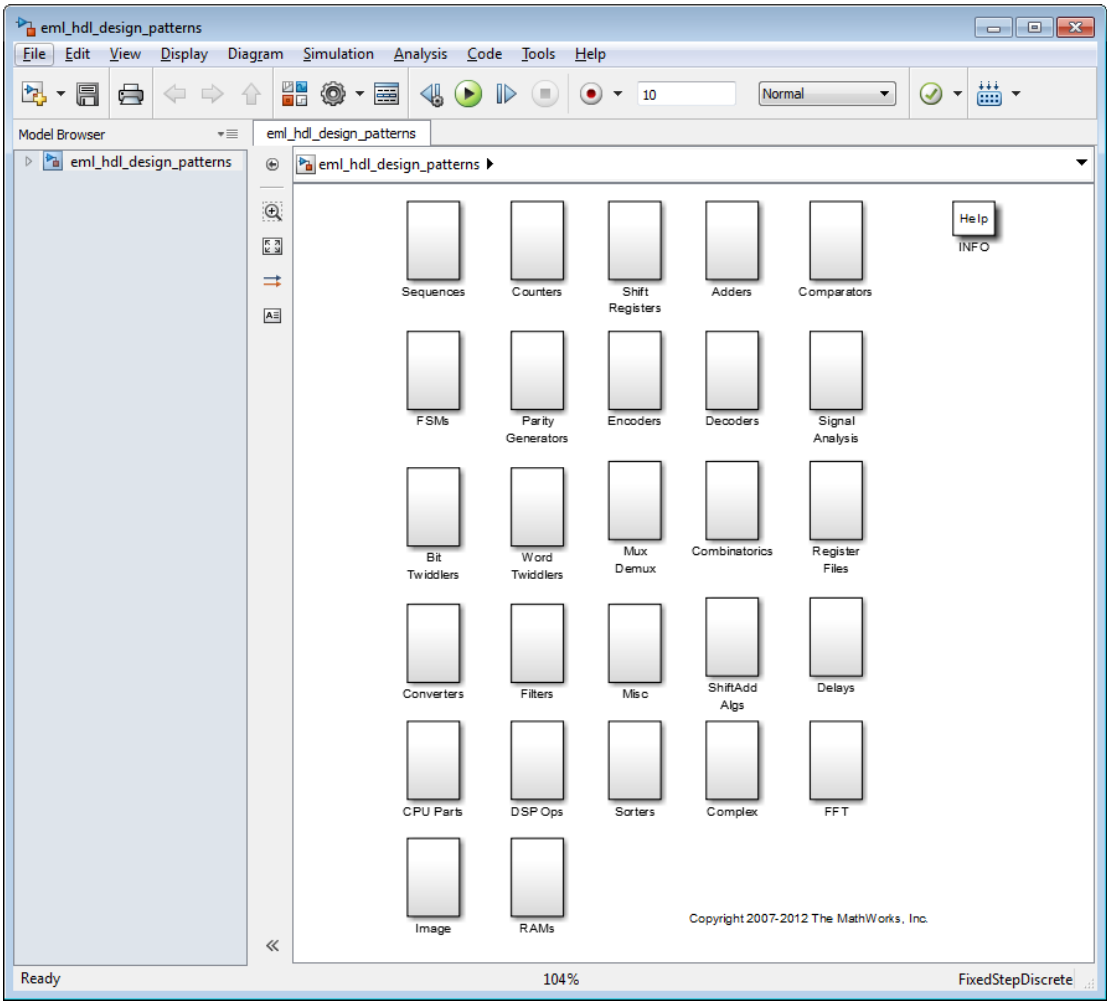
If you enable `InstantiateFunctions`, `UseMatrixTypesInHDL` has no effect.

MATLAB Function Block Design Patterns for HDL

In this section...

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 11-70.

Efficient Fixed-Point Algorithms

The MATLAB Function block supports fixed point arithmetic 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 (sfix7_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: `(sfix5_En2)`
- b: `(sfix5_En3)`
- y: `(sfix7_En4)`

Before HDL code generation, the operation

```
expr = (a*b) - (a+b);
```

is broken down internally into the following substeps:

1. `tmul = a * b;`
2. `tadd = a + b;`
3. `tsub = tmul - tadd;`
4. `y = tsub;`

Based on the `fimath` settings (see "Design Guidelines for the MATLAB Function Block" on page 20-35) 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'`, substep 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 substep 3 is computed as `sfix11_En5`. Accordingly, substep 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.

VHDL code excerpt:

```
 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 fsm_SFHDL;
```

Verilog code excerpt:

```
//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 coder 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 `log2(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` and`FSMs/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' syntheses 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 = presetData;
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.

Design Guidelines for the MATLAB Function Block

In this section...

Introduction

This section describes recommended practices when using the MATLAB Function block for HDL code generation.

By setting MATLAB Function block options as described in this section, you can significantly increase the efficiency of generated HDL code. See "Set Fixed-Point Options for the MATLAB Function Block" on page 20-10 for an example.

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.

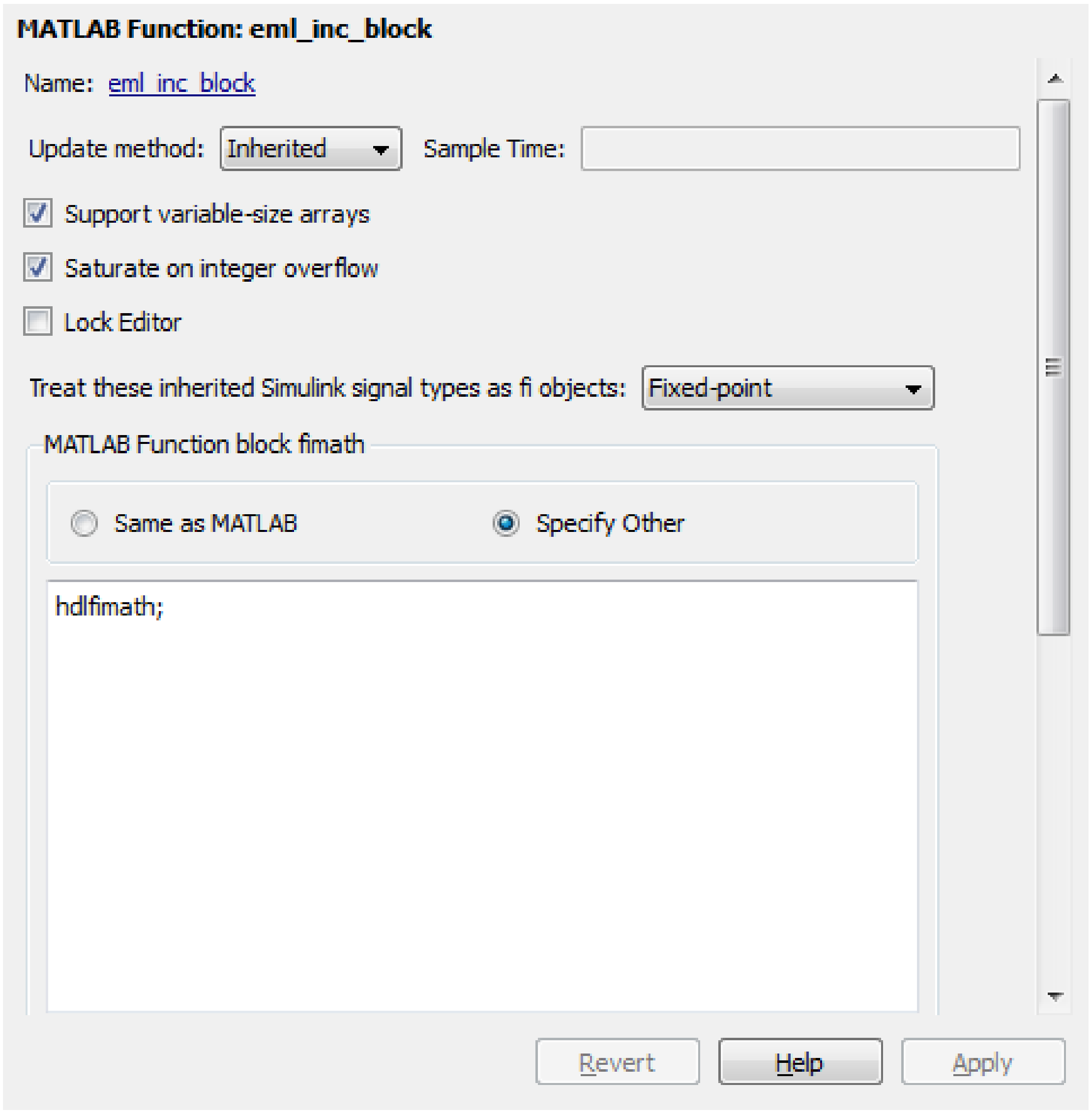
Such functions must be defined in files that are on the MATLAB Function block path. Use the `%#codegen` compilation directive to indicate that the MATLAB code is suitable for code generation. See "Function Definition" (MATLAB Coder) for information on how to create, compile, and invoke external functions.

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, select **Build** from the **Tools** menu in the MATLAB Function Block Editor (or press **CTRL+B**).

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. Replace the default **MATLAB Function block fimath** specification with a call to the `hdlfimath` function, as shown in the following figure.


MATLAB Function: eml_inc_block

Name: eml_inc_block

Update method: Inherited Sample Time:

[✓] Support variable-size arrays

[✓] Saturate on integer overflow

[ ] Lock Editor

Treat these inherited Simulink signal types as fi objects: Fixed-point

MATLAB Function block fimath

○ Same as MATLAB [● Specify Other]

```
hdlfimath;
```

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);
```

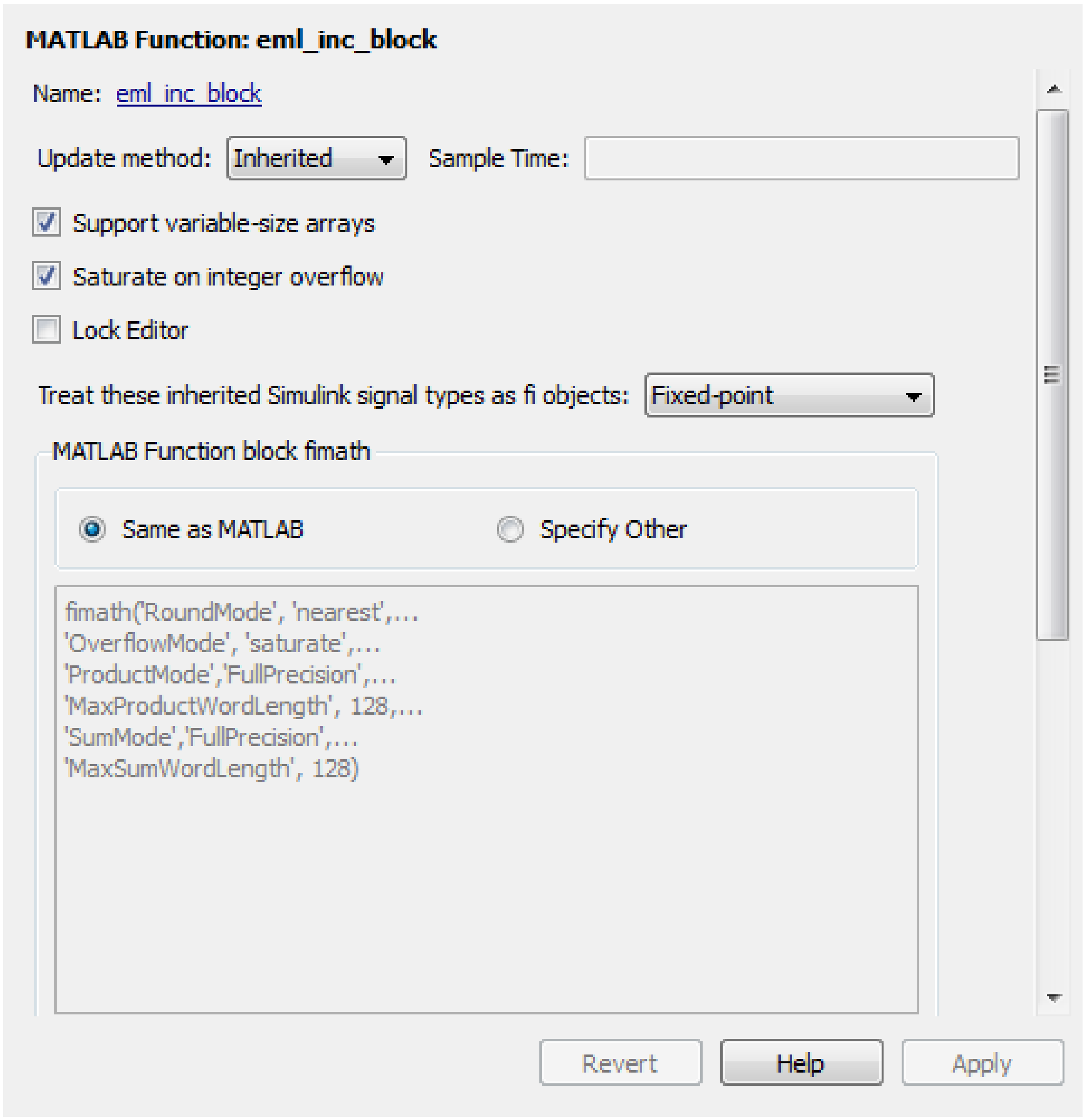
Note: Use of `'floor'` rounding mode for signed integer division will cause an error at code generation time. The HDL division operator does not support `'floor'` rounding mode. Use `'round'` mode, or else change the signed integer division operations to unsigned integer division.

Note: 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, as shown in the following figure.



MATLAB Function: eml_inc_block

Name: eml_inc_block

Update method: Inherited Sample Time:

[✓] Support variable-size arrays

[✓] Saturate on integer overflow

[ ] Lock Editor

Treat these inherited Simulink signal types as fi objects: Fixed-point

MATLAB Function block fimath

[● Same as MATLAB] ○ Specify Other

```
fimath('RoundMode', 'nearest',...
'OverflowMode', 'saturate',...
'ProductMode','FullPrecision',...
'MaxProductWordLength', 128,...
'SumMode','FullPrecision',...
'MaxSumWordLength', 128)
```

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.

Distributed Pipeline Insertion for MATLAB Function Blocks

In this section...

Overview

Distributed pipeline insertion is a special optimization for HDL code generated from MATLAB Function blocks or Stateflow charts. Distributed pipeline insertion lets you achieve higher clock rates in your HDL applications, at the cost of some amount of latency caused by the introduction of pipeline registers.

For general information on distributed pipeline insertion, including limitations, see "DistributedPipelining" on page 12-10.

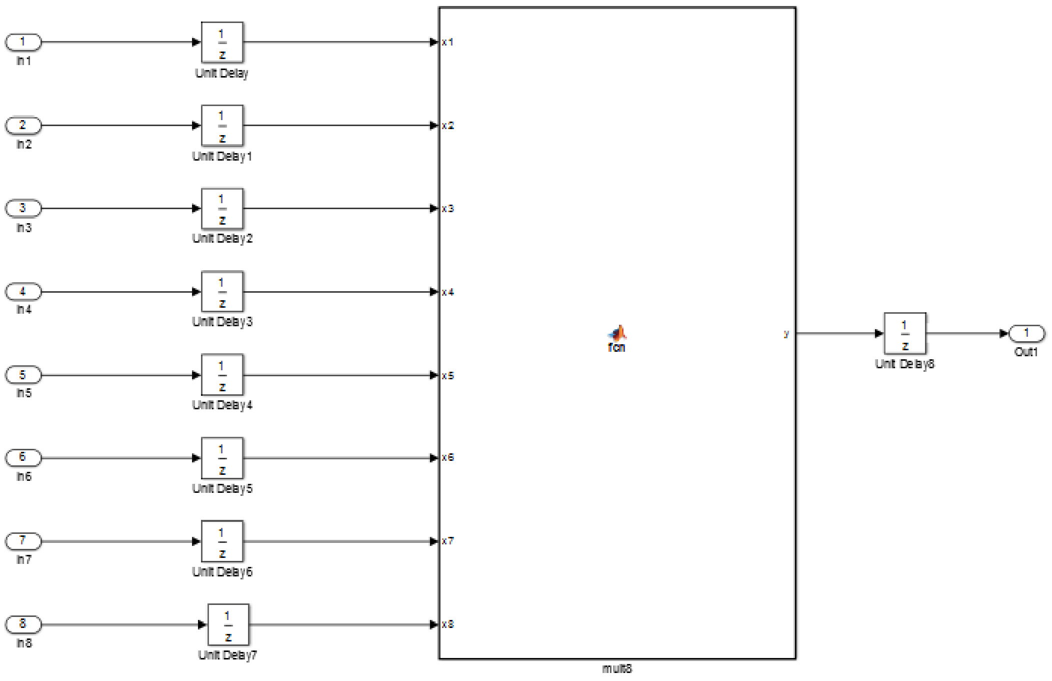
Distributed Pipelining in a Multiplier Chain

This example shows distributed pipeline insertion in a simple model that implements a chain of 5 multiplications.

To open the model, enter the following:

```
mpipe_multichain
```

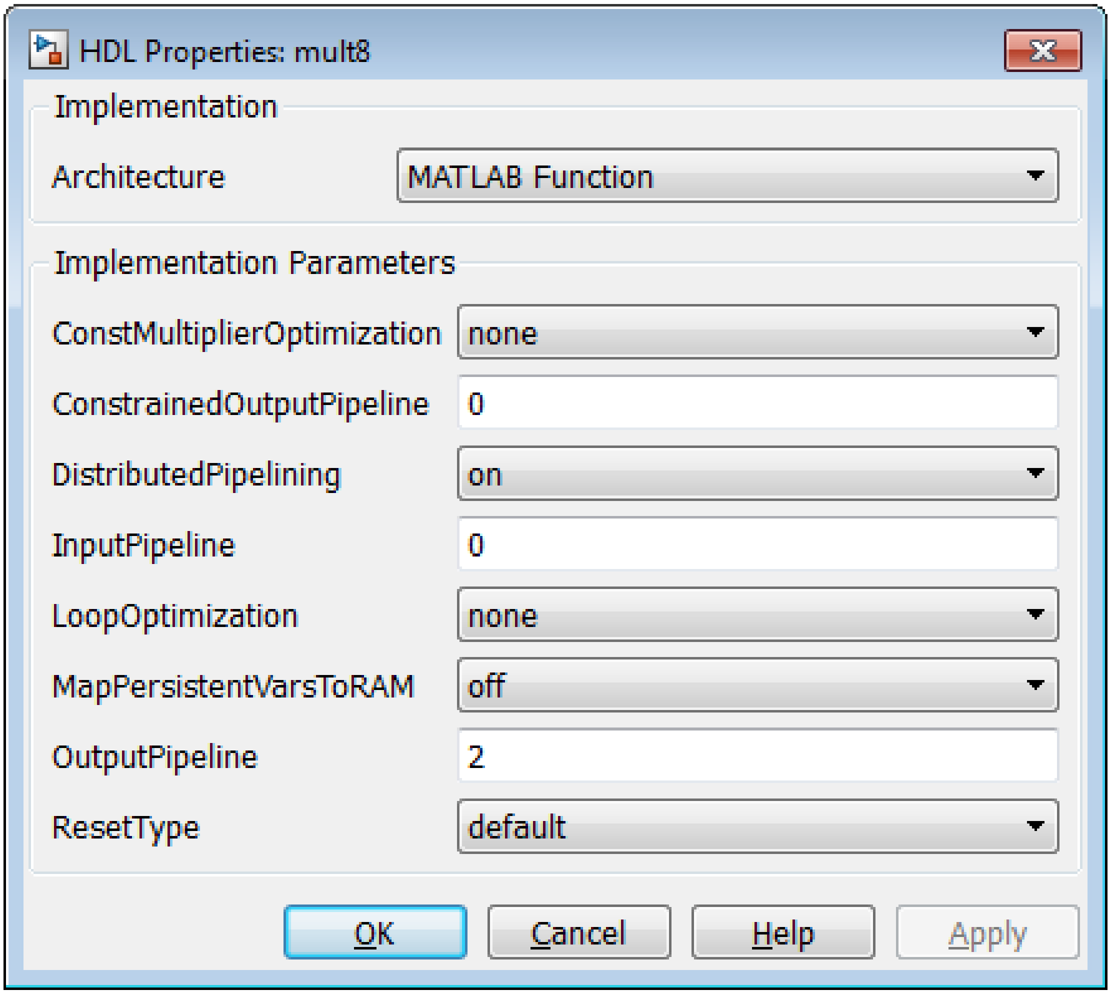
The root level model contains a subsystem `multi_chain`. The `multi_chain` subsystem functions as the device under test (DUT) from which to generate HDL code. The subsystem drives a MATLAB Function block, `mult8`. The following figure shows the subsystem.



The following shows a chain of multiplications as coded in the `mult8` MATLAB Function block:

```
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;
```

To apply distributed pipeline insertion to this block, use the HDL Properties dialog box for the `mult8` block. Specify generation of two pipeline stages for the MATLAB Function block, and enable the distributed pipeline optimization:



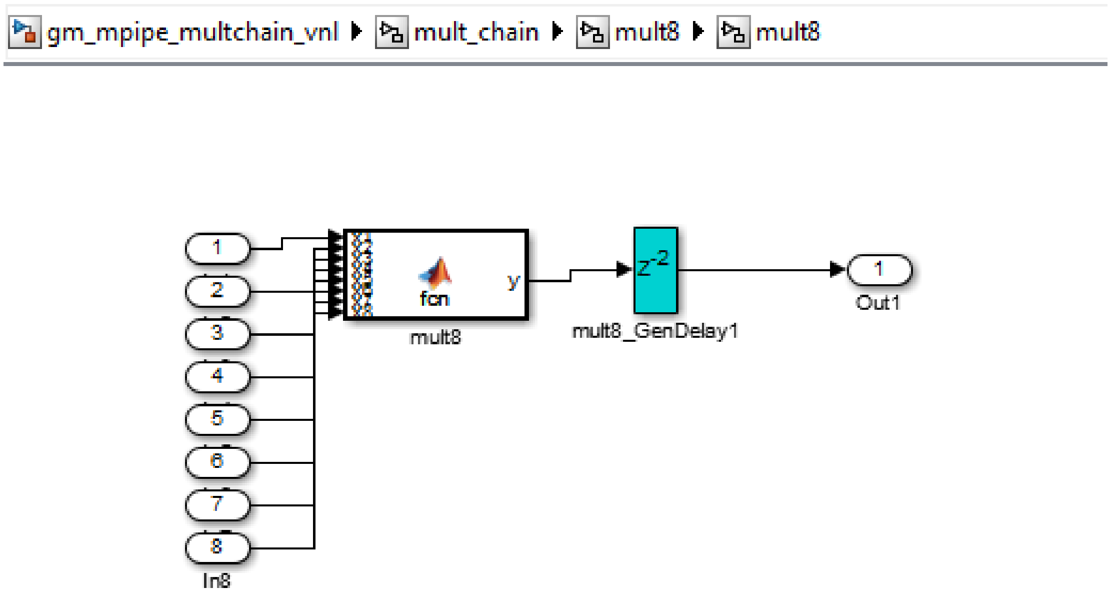
Architecture	MATLAB Function
ConstMultiplierOptimization	none
ConstrainedOutputPipeline	0
DistributedPipelining	on
InputPipeline	0
LoopOptimization	none
MapPersistentVarsToRAM	off
OutputPipeline	2
ResetType	default

In the Configuration Parameters dialog box, the top-level **HDL Code Generation** options specify that:

- VHDL code is generated from the subsystem `mpipe_multchain/mult_chain`.

- HDL Coder will generate code and display the generated model.

The insertion of two pipeline stages into the generated HDL code results in a latency of two clock cycles. In the generated model, a delay of two clock cycles is inserted before the output of the `mpipe_multchain/mult_chain/mult8` subsystem so that simulations of the model reflect the behavior of the generated HDL code. The following figure shows the inserted Delay block.



The following listing shows the complete architecture section of the generated code. Comments generated by HDL Coder indicate the pipeline register definitions.

```
ARCHITECTURE fsm_SFHDL OF mult8 IS

    SIGNAL pipe_var_0_1 : signed(7 DOWNTO 0);   -- Pipeline reg from stage 0 to stage 1
    SIGNAL b_pipe_var_0_1 : signed(7 DOWNTO 0);   -- Pipeline reg from stage 0 to stage 1
    SIGNAL c_pipe_var_0_1 : signed(7 DOWNTO 0);   -- Pipeline reg from stage 0 to stage 1
    SIGNAL d_pipe_var_0_1 : signed(7 DOWNTO 0);   -- Pipeline reg from stage 0 to stage 1
    SIGNAL pipe_var_1_2 : signed(7 DOWNTO 0);   -- Pipeline reg from stage 1 to stage 2
    SIGNAL b_pipe_var_1_2 : signed(7 DOWNTO 0);   -- Pipeline reg from stage 1 to stage 2
    SIGNAL pipe_var_0_1_next : signed(7 DOWNTO 0);
    SIGNAL b_pipe_var_0_1_next : signed(7 DOWNTO 0);
    SIGNAL c_pipe_var_0_1_next : signed(7 DOWNTO 0);
    SIGNAL d_pipe_var_0_1_next : signed(7 DOWNTO 0);
    SIGNAL pipe_var_1_2_next : signed(7 DOWNTO 0);
    SIGNAL b_pipe_var_1_2_next : signed(7 DOWNTO 0);
    SIGNAL y1 : signed(7 DOWNTO 0);
    SIGNAL y2 : signed(7 DOWNTO 0);
    SIGNAL y3 : signed(7 DOWNTO 0);
    SIGNAL y4 : signed(7 DOWNTO 0);
    SIGNAL y5 : signed(7 DOWNTO 0);
    SIGNAL y6 : signed(7 DOWNTO 0);
    SIGNAL mul_temp : signed(15 DOWNTO 0);
    SIGNAL mul_temp_0 : signed(15 DOWNTO 0);
    SIGNAL mul_temp_1 : signed(15 DOWNTO 0);
    SIGNAL mul_temp_2 : signed(15 DOWNTO 0);
    SIGNAL mul_temp_3 : signed(15 DOWNTO 0);
    SIGNAL mul_temp_4 : signed(15 DOWNTO 0);
    SIGNAL mul_temp_5 : signed(15 DOWNTO 0);

BEGIN
    initialize_mult8 : PROCESS (clk, reset)
    BEGIN
        IF reset = '1' THEN
            pipe_var_0_1 <= to_signed(0, 8);
            b_pipe_var_0_1 <= to_signed(0, 8);
            c_pipe_var_0_1 <= to_signed(0, 8);
            d_pipe_var_0_1 <= to_signed(0, 8);
            pipe_var_1_2 <= to_signed(0, 8);
            b_pipe_var_1_2 <= to_signed(0, 8);
        ELSIF clk'EVENT AND clk= '1' THEN
            IF clk_enable= '1' THEN
                pipe_var_0_1 <= pipe_var_0_1_next;
                b_pipe_var_0_1 <= b_pipe_var_0_1_next;
                c_pipe_var_0_1 <= c_pipe_var_0_1_next;
                d_pipe_var_0_1 <= d_pipe_var_0_1_next;
                pipe_var_1_2 <= pipe_var_1_2_next;
                b_pipe_var_1_2 <= b_pipe_var_1_2_next;
            END IF;
        END IF;
    END PROCESS initialize_mult8;

    -- This block supports an embeddable subset of the MATLAB language.
    -- See the help menu for details.
    --y = (x1+x2)+(x3+x4)+(x5+x6)+(x7+x8);
    mul_temp <= signed(x1) * signed(x2);

    y1 <= "01111111" WHEN (mul_temp(15) = '0') AND (mul_temp(14 DOWNTO 7) /= "00000000")
        ELSE "10000000" WHEN (mul_temp(15) = '1') AND (mul_temp(14 DOWNTO 7) /= "11111111")
        ELSE mul_temp(7 DOWNTO 0);

    mul_temp_0 <= signed(x3) * signed(x4);

    y2 <= "01111111" WHEN (mul_temp_0(15) ='0') AND (mul_temp_0(14 DOWNTO 7) /= "00000000")
    ELSE "10000000" WHEN (mul_temp_0(15) = '1') AND (mul_temp_0(14 DOWNTO 7) /= "11111111")
    ELSE mul_temp_0(7 DOWNTO 0);

    mul_temp_1 <= signed(x5) * signed(x6);

   y3 <= "01111111" WHEN (mul_temp_1(15) = '0') AND (mul_temp_1(14 DOWNTO 7) /= "00000000")
   ELSE "10000000" WHEN (mul_temp_1(15) = '1') AND (mul_temp_1(14 DOWNTO 7) /= "11111111")
   ELSE mul_temp_1(7 DOWNTO 0);

    mul_temp_2 <= signed(x7) * signed(x8);

    y4 <= "01111111" WHEN (mul_temp_2(15)= '0')AND (mul_temp_2(14 DOWNTO 7) /= "00000000")
    ELSE "10000000" WHEN (mul_temp_2(15) = '1') AND (mul_temp_2(14 DOWNTO 7) /= "11111111")
    ELSE mul_temp_2(7 DOWNTO 0);

    mul_temp_3 <= pipe_var_0_1 * b_pipe_var_0_1;

    y5 <= "01111111" WHEN (mul_temp_3(15) = '0') AND (mul_temp_3(14 DOWNTO 7)/= "00000000")
    ELSE "10000000" WHEN (mul_temp_3(15) = '1') AND (mul_temp_3(14 DOWNTO 7) /= "11111111")
    ELSE mul_temp_3(7 DOWNTO 0);

    mul_temp_4 <= c_pipe_var_0_1 * d_pipe_var_0_1;

    y6 <= "01111111" WHEN (mul_temp_4(15)='0') AND (mul_temp_4(14 DOWNTO 7) /= "00000000")
    ELSE "10000000" WHEN (mul_temp_4(15) = '1') AND (mul_temp_4(14 DOWNTO 7) /= "11111111")
    ELSE mul_temp_4(7 DOWNTO 0);

    mul_temp_5 <= pipe_var_1_2 * b_pipe_var_1_2;

    y <= "01111111" WHEN (mul_temp_5(15) = '0') AND (mul_temp_5(14 DOWNTO 7) /= "00000000")
    ELSE "10000000" WHEN (mul_temp_5(15) = '1') AND (mul_temp_5(14 DOWNTO 7) /= "11111111")
    ELSE std_logic_vector(mul_temp_5(7 DOWNTO 0));

    b_pipe_var_1_2_next <= y6;
    pipe_var_1_2_next <= y5;
    d_pipe_var_0_1_next <= y4;
    c_pipe_var_0_1_next <= y3;
    b_pipe_var_0_1_next <= y2;
    pipe_var_0_1_next <= y1;
END fsm_SFHDL;
```

Generating Scripts for HDL Simulators and Synthesis Tools

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 21-4.
- 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 21-8.

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	`HDLCompileFilePostfix`	`_compile.do`
Simulation	`HDLSimFilePostfix`	`_sim.do`
Synthesis	`HDLSynthFilePostfix`	Depends on the selected synthesis tool. See HDLSynthTool.

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` Default:`'vlib %s\n'`	Format name passed to `fprintf` to write the `Init` section of the compilation script. The implicit argument is the contents of the `VHDLLibraryName` property, which defaults to`'work'`. You can override the default `Init` string (`'vlib work\n'`) by changing the value of `VHDLLibraryName`.
Name: `HDLCompileVHDLCmd` Default: `'vcom %s %s\n'`	Format name passed to `fprintf` to write the `Cmd` section of the compilation script for VHDL files. The two implicit arguments 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).
Name: `HDLCompileVerilogCmd` Default: `'vlog %s %s\n'`	Format name passed to `fprintf` to write the `Cmd` section of the compilation script for Verilog files. The two implicit arguments 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).
Name:`HDLCompileTerm` Default:`''`	Format name passed to `fprintf` to write the termination portion of the compilation script.
Name: `HDLSimInit` Default:	Format name passed to `fprintf` to write the initialization section of the simulation script.

Property Name and Default	Description
`['onbreak resume\n',...` `'onerror resume\n']`	
Name: `HDLSimCmd` Default: `'vsim -novopt %s.%s\n'`	Format name passed to `fprintf` to write the simulation command. If your target language is VHDL, the first implicit argument is the value of the `VHDLLibraryName` property. If your target language is Verilog, the first implicit argument is `'work'`. The second implicit argument is the top-level module or entity name.
Name: `HDLSimViewWaveCmd` Default: `'add wave sim:%s\n'`	Format name passed to `fprintf` 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` Default: `'run -all\n'`	Format name passed to `fprintf` to write the `Term` 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 HDLSynthTool.
Name: `HDLSynthInit`	Format name passed to `fprintf` to write the `Init` section of the synthesis script. The content of the format name is specific to the selected synthesis tool. See HDLSynthTool.
Name: `HDLSynthCmd`	Format name passed to `fprintf` to write the `Cmd` section of the synthesis script. The content of the string is specific to the selected synthesis tool. See HDLSynthTool.
Name: `HDLSynthTerm`	Format name passed to `fprintf` to write the `Term` section of the synthesis script. The content of the string is specific to the selected synthesis tool. See HDLSynthTool.

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"
```

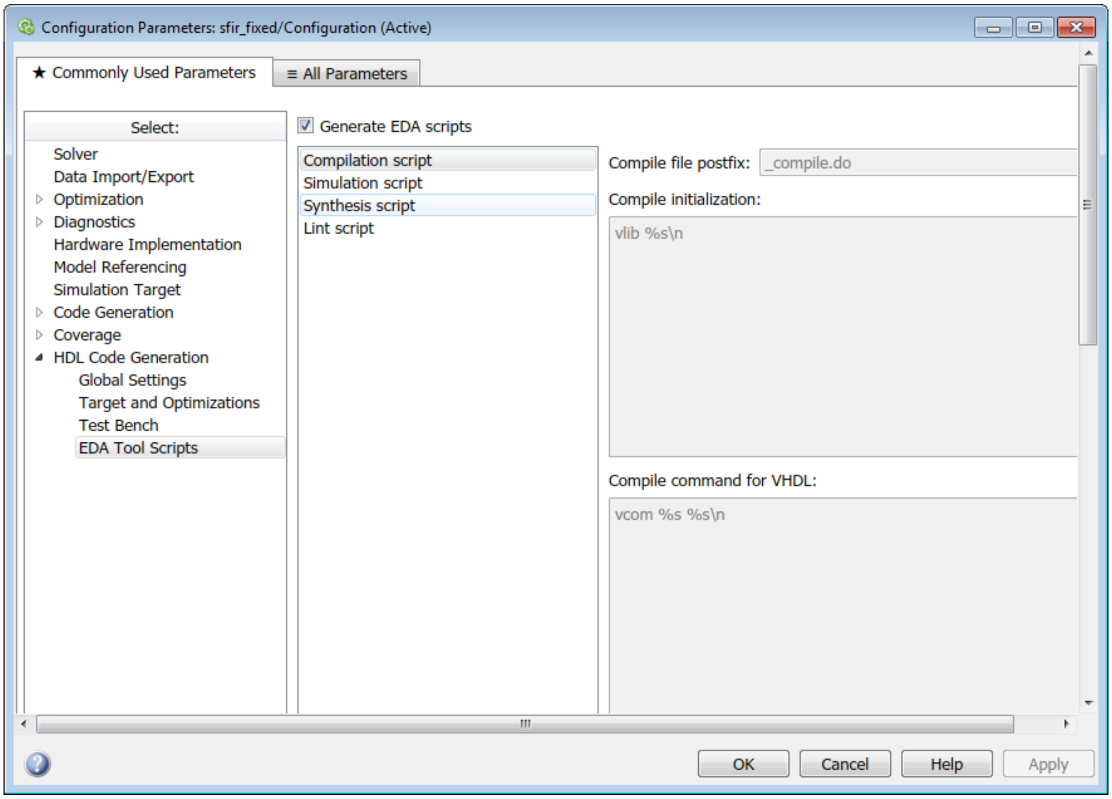
Configure Compilation, Simulation, Synthesis, and Lint Scripts

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 21-4.

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.



Generate EDA scripts: ☑

Compile file postfix: _compile.do

Compile initialization: vlib %s\n

Compile command for VHDL: vcom %s %s\n

3 The **Generate EDA scripts** option controls the generation of script files. By default, this option is selected.

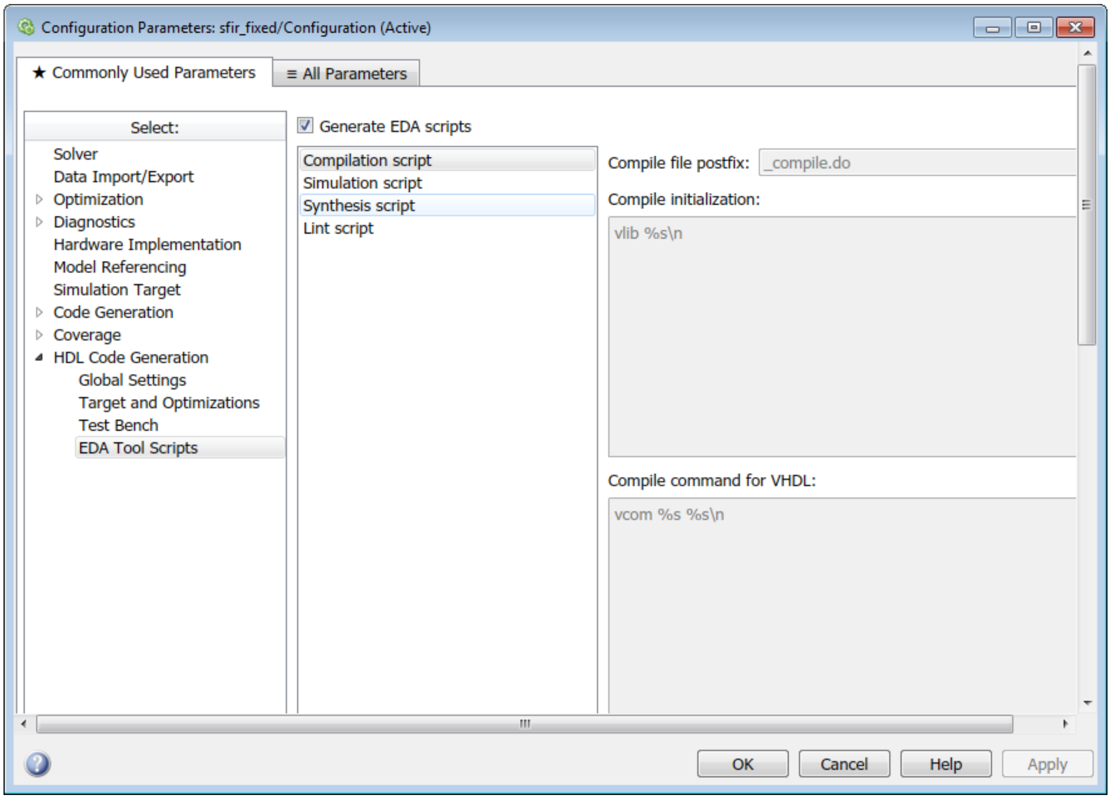
If you want to disable script generation, clear this check box and click **Apply**.

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 21-9 for further information.
- **Simulation script**: Options related to customizing scripts for HDL simulators. See “Simulation Script Options” on page 21-11 for further information.
- **Synthesis script**: Options related to customizing scripts for synthesis tools. See “Synthesis Script Options” on page 21-13 for further information.

Compilation Script Options

The following figure shows the **Compilation script** pane, with options set to their default values.


Configuration Parameters: sfir_fixed/Configuration (Active)

★ Commonly Used Parameters | ≡ All Parameters

Select:
- Solver
- Data Import/Export
- Optimization
- Diagnostics
- Hardware Implementation
- Model Referencing
- Simulation Target
- Code Generation
- Coverage
- HDL Code Generation
  - Global Settings
  - Target and Optimizations
  - Test Bench
  - EDA Tool Scripts

☑ Generate EDA scripts

- Compilation script
- Simulation script
- Synthesis script
- Lint script

Compile file postfix: _compile.do

Compile initialization:

```
vlib %s\n
```

Compile command for VHDL:

```
vcom %s %s\n
```

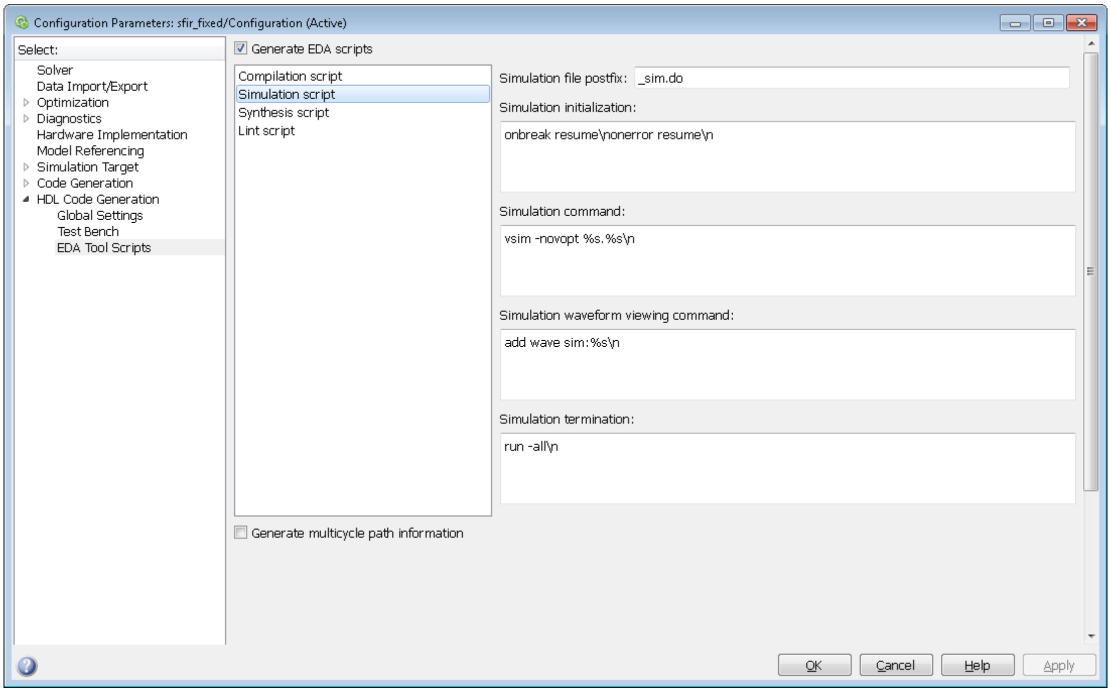
The following table summarizes the **Compilation script** options.

Option and Default	Description
Compile file postfix' `'_compile.do'`	Postfix appended to the DUT name or test bench name to form the script file name.
Name: **Compile initialization** Default:`'vlib %s\n'`	Format name passed to `fprintf` to write the `Init` section of the compilation script. The argument is the contents of the `VHDLLibraryName` property, which defaults to`'work'`. You can override the default `Init'vlib work\n'` by changing the value of `VHDLLibraryName`.

Option and Default	Description
Name: **Compile command for VHDL** Default: `'vcom %s %s\n'`	Format name passed to `fprintf` to write the `Cmd` section of the compilation script for VHDL files. The two arguments are the contents of the `SimulatorFlags` property option and the filename of the current entity or module. To omit the flags, set `SimulatorFlags` to `''` (the default).
Name: **Compile command for Verilog** Default: `'vlog %s %s\n'`	Format name passed to `fprintf` to write the `Cmd` section of the compilation script for Verilog files. The two arguments are the contents of the `SimulatorFlags` property and the filename of the current entity or module. To omit the flags, set `SimulatorFlags` to `''` (the default).
Name: **Compile termination** Default:`''`	Format name passed to `fprintf` 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.



Generate EDA scripts: ✓

Simulation file postfix: _sim.do

Simulation initialization:

```
onbreak resume\nonerror resume\n
```

Simulation command:

```
vsim -novopt %s.%s\n
```

Simulation waveform viewing command:

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

Simulation termination:

```
run -all\n
```

Generate multicycle path information: ☐

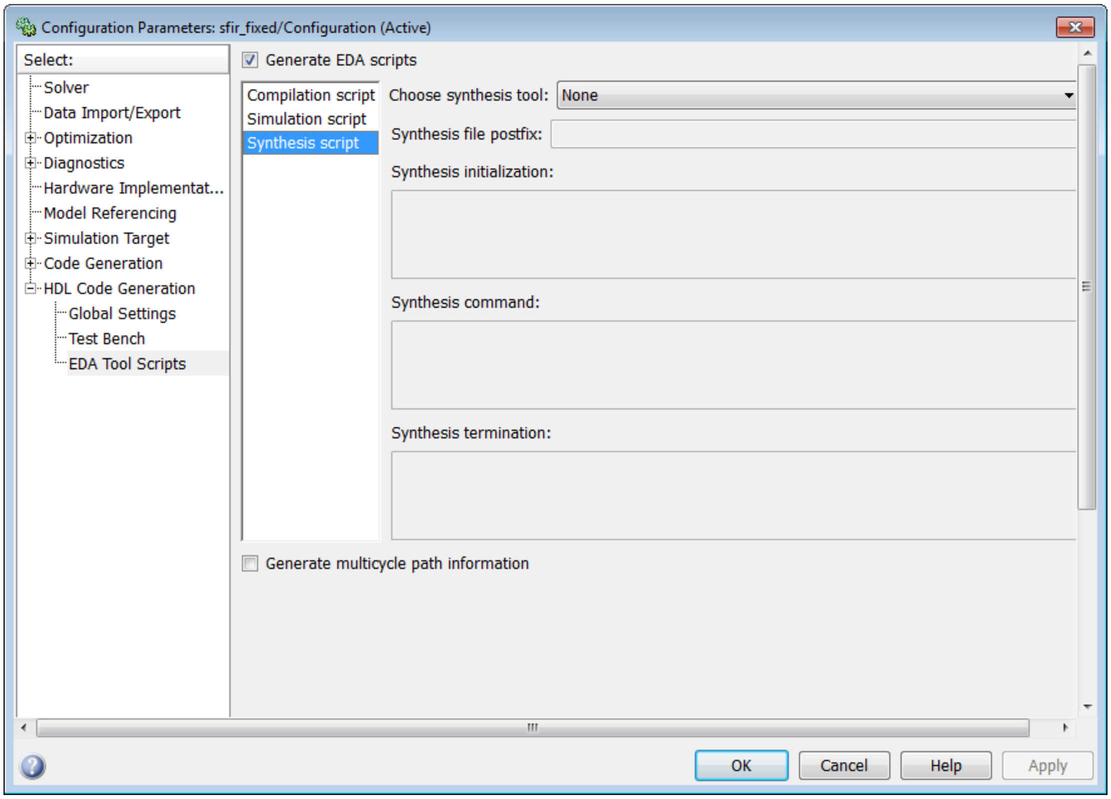
The following table summarizes the **Simulation script** options.

Option and Default	Description
Simulation file postfix `'_sim.do'`	Postfix appended to the model name or test bench name to form the simulation script file name.
Simulation initialization Default: `['onbreak resume\nonerror resume\n']`	Format name passed to `fprintf` to write the initialization section of the simulation script.
Simulation command Default: `'vsim -novopt %s.%s\n'`	Format name passed to `fprintf` to write the simulation command. 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'`.

Option and Default	Description
	The second implicit argument is the top-level module or entity name.
Simulation waveform viewing command Default: `'add wave sim:%s\n'`	Format name passed to `fprintf` to write the simulation script waveform viewing command. The top-level module or entity signal names are implicit arguments.
Simulation termination Default: `'run -all\n'`	Format name passed to `fprintf` 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.


Configuration Parameters: sfir_fixed/Configuration (Active)

Select:
- Solver
- Data Import/Export
- Optimization
- Diagnostics
- Hardware Implementat...
- Model Referencing
- Simulation Target
- Code Generation
- HDL Code Generation
  - Global Settings
  - Test Bench
  - EDA Tool Scripts

☑ Generate EDA scripts

Compilation script / Simulation script / Synthesis script

Choose synthesis tool: None

Synthesis file postfix:

Synthesis initialization:

Synthesis command:

Synthesis termination:

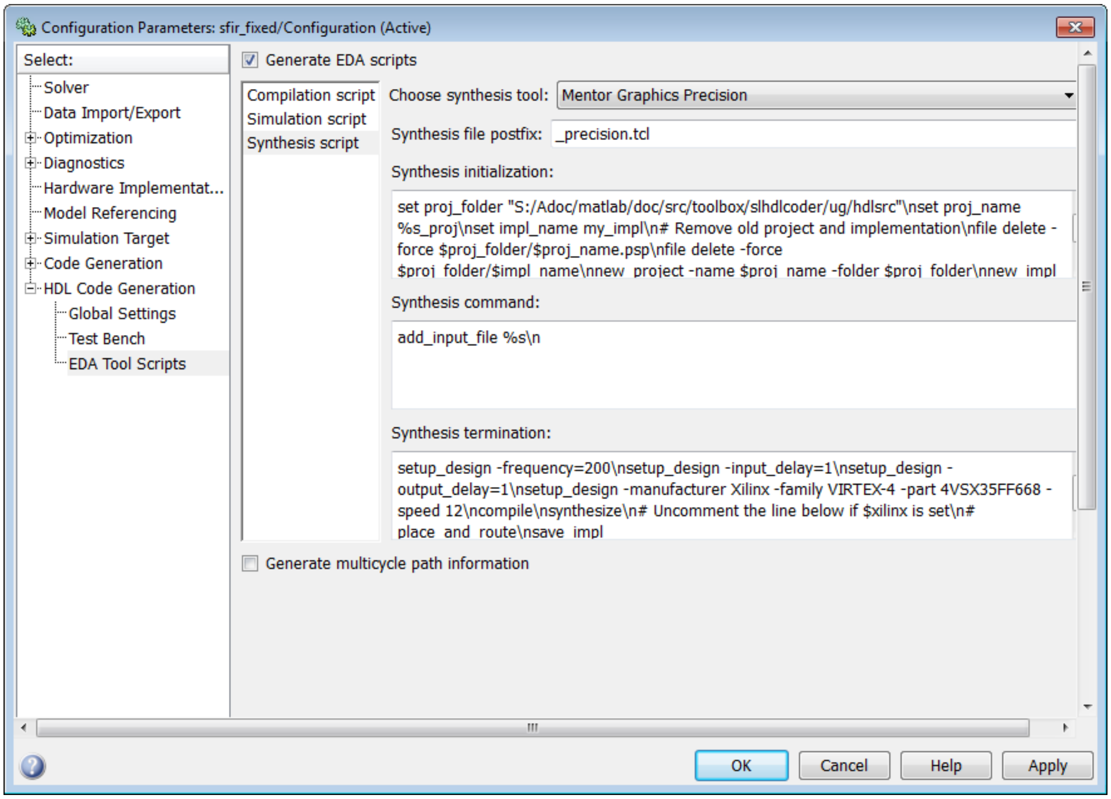
☐ Generate multicycle path information

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.



☑ Generate EDA scripts

Choose synthesis tool: Mentor Graphics Precision

Synthesis file postfix: _precision.tcl

Synthesis initialization:

```
set proj_folder "S:/Adoc/matlab/doc/src/toolbox/slhdlcoder/ug/hdlsrc"\nset proj_name
%s_proj\nset impl_name my_impl\n# Remove old project and implementation\nfile delete -
force $proj_folder/$proj_name.psp\nfile delete -force
$proj_folder/$impl_name\nnew_project -name $proj_name -folder $proj_folder\nnew_impl
```

Synthesis command:

```
add_input_file %s\n
```

Synthesis termination:

```
setup_design -frequency=200\nsetup_design -input_delay=1\nsetup_design -
output_delay=1\nsetup_design -manufacturer Xilinx -family VIRTEX-4 -part 4VSX35FF668 -
speed 12\ncompile\nsynthesize\n# Uncomment the line below if $xilinx is set\n#
place_and_route\nsave_impl
```

☐ Generate multicycle path information

The following table summarizes the **Synthesis script** options.

Option Name	Description
Choose synthesis tool	`None` (default): do not generate a synthesis script `Xilinx ISE`: generate a synthesis script for Xilinx ISE `Microsemi Libero`: generate a synthesis script for Microsemi Libero `Mentor Graphics Precision`: generate a synthesis script for Mentor Graphics Precision `Altera Quartus II`: generate a synthesis script for Altera Quartus II `Synopsys Synplify Pro`: generate a synthesis script for Synopsys Synplify Pro `Xilinx Vivado`: generate a synthesis script for Xilinx Vivado

Option Name	Description
	`Custom`: generate a custom synthesis script

Synthesis file postfix	Your choice of synthesis tool 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`

Synthesis initialization	Format name passed to `fprintf` to write the `Init` section of the synthesis script. The default string is a synthesis project creation command. The implicit argument is the top-level module or entity name. The content of the string is specific to the selected synthesis tool.
Synthesis command	Format name passed to `fprintf` to write the `Cmd` section of the synthesis script. The implicit argument is the file name of the entity or module. The content of the string is specific to the selected synthesis tool.
Synthesis termination	Format name passed to `fprintf` to write the `Term` section of the synthesis script. The content of the string is specific to the selected synthesis tool.

Add Synthesis Attributes

To learn how to add synthesis attributes in the generated HDL code for multiplier mapping, see “DSPStyle” on page 12-11.

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

Open and Run Tasks in the HDL Workflow Advisor

In this section...
"Open the HDL Workflow Advisor" on page 22-2
"Run Tasks in the HDL Workflow Advisor" on page 22-3

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 and other information obtained during synthesis
- Complete automated workflows for selected FPGA development target devices and Simulink Real-Time™, including FPGA-in-the-loop simulation

Open the HDL Workflow Advisor

To start the HDL Workflow Advisor from a model:

1. Open your model and select the DUT subsystem
2. Right-click the DUT subsystem and select **HDL Code** > **HDL Workflow Advisor**

   The HDL Workflow Advisor initializes and appears.

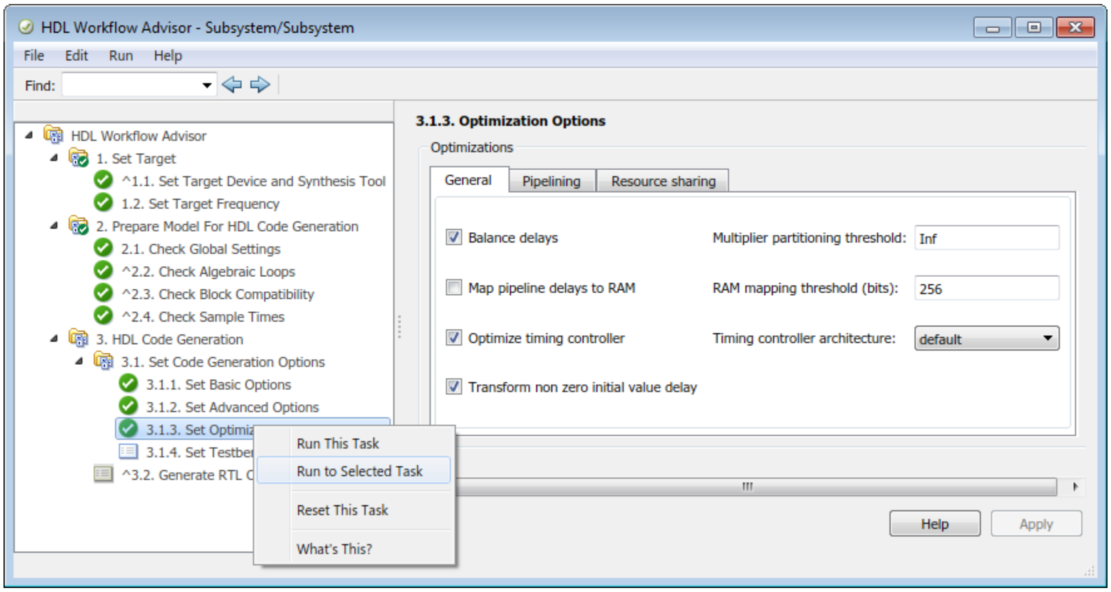
To start the HDL Workflow Advisor from the command line, enter `hdladvisor(`*`system`*`)`. *`system`* is a handle or name of the model or subsystem that you want to check. For more information, see the hdladvisor function reference page.

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 content 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 many 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 from the Simulink model, prepare the 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 generate HDL code from the model, 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**.


HDL Workflow Advisor - Subsystem/Subsystem
File
Edit
Run
Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis Tool
1.2. Set Target Frequency
2. Prepare Model For HDL Code Generation
2.1. Check Global Settings
^2.2. Check Algebraic Loops
^2.3. Check Block Compatibility
^2.4. Check Sample Times
3. HDL Code Generation
3.1. Set Code Generation Options
3.1.1. Set Basic Options
3.1.2. Set Advanced Options
3.1.3. Set Optimiz
3.1.4. Set Testbe
^3.2. Generate RTL C
Run This Task
Run to Selected Task
Reset This Task
What's This?
Help
Apply

3.1.3. Optimization Options

Optimizations

General | Pipelining | Resource sharing

- [✓] Balance delays — Multiplier partitioning threshold: Inf
- [ ] Map pipeline delays to RAM — RAM mapping threshold (bits): 256
- [✓] Optimize timing controller — Timing controller architecture: default
- [✓] Transform non zero initial value delay

To rerun a task that you have already run, reset the 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**.

Related Examples

- "Save and Restore HDL Workflow Advisor State" on page 22-5
- "Fix a Workflow Advisor Warning or Failure" on page 22-9
- "View and Save HDL Workflow Advisor Reports" on page 22-12
- "HDL Workflow Advisor Tasks" on page 25-2
- "Generate Code Using the HDL Workflow Advisor" on page 22-17

Save and Restore HDL Workflow Advisor State

In this section...

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 and other information obtained during synthesis
- Complete automated workflows for selected FPGA development target devices and Simulink Real-Time, including FPGA-in-the-loop simulation

How the Save and Restore Process Works

By default, the HDL Coder software saves the state of the most recent HDL Workflow Advisor session. The next time 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.

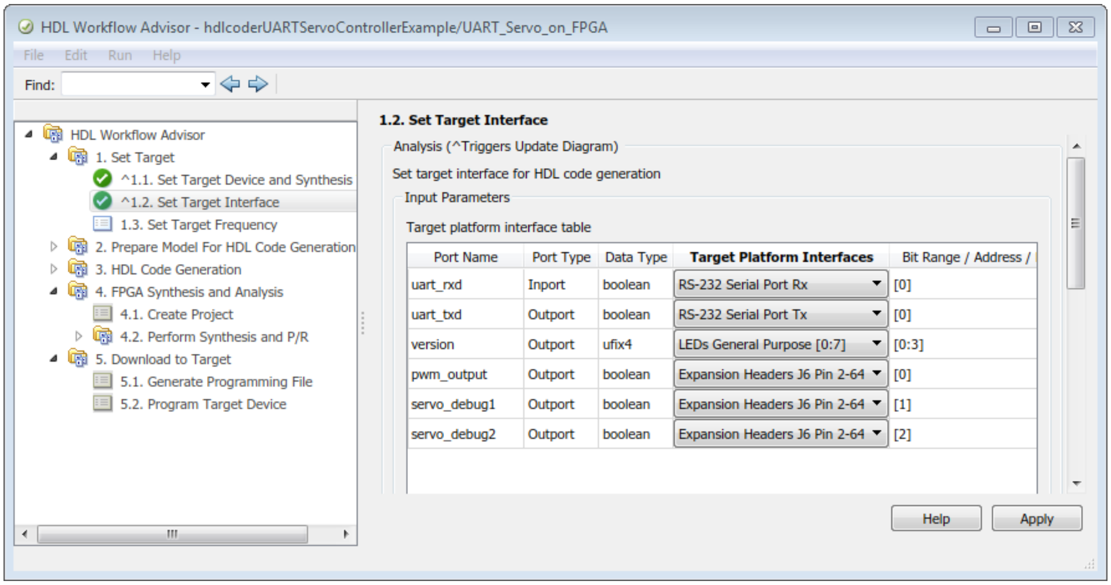
Limitations of the Save and Restore Process

The save and restore process has the following limitations:

- Operations that you perform outside the HDL Workflow Advisor is not included in the save/restore process.
- The state of HDL Workflow Advisor tasks involving third-party tools are not saved or restored.

Save the HDL Workflow Advisor State

You can create and save a restore point after completion of a task sequence. For example, the following figure shows the HDL Workflow Advisor after completion of the **Set Target Interface** task.


HDL Workflow Advisor - hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA
File Edit Run Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis
^1.2. Set Target Interface
1.3. Set Target Frequency
2. Prepare Model For HDL Code Generation
3. HDL Code Generation
4. FPGA Synthesis and Analysis
4.1. Create Project
4.2. Perform Synthesis and P/R
5. Download to Target
5.1. Generate Programming File
5.2. Program Target Device
Help
Apply

1.2. Set Target Interface

Analysis (^Triggers Update Diagram)

Set target interface for HDL code generation

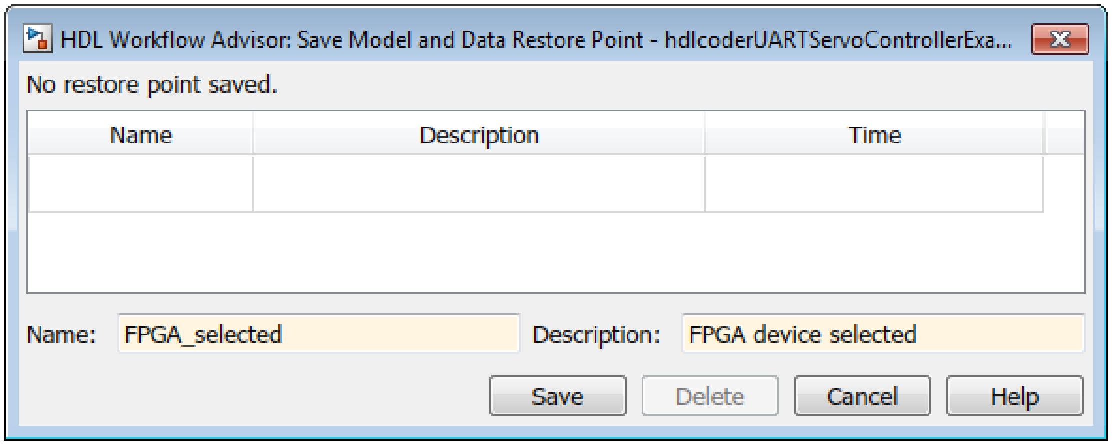
Input Parameters

Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address /
uart_rxd	Inport	boolean	RS-232 Serial Port Rx	[0]
uart_txd	Outport	boolean	RS-232 Serial Port Tx	[0]
version	Outport	ufix4	LEDs General Purpose [0:7]	[0:3]
pwm_output	Outport	boolean	Expansion Headers J6 Pin 2-64	[0]
servo_debug1	Outport	boolean	Expansion Headers J6 Pin 2-64	[1]
servo_debug2	Outport	boolean	Expansion Headers J6 Pin 2-64	[2]

To save the HDL Workflow Advisor settings:

1. In the HDL Workflow Advisor, select **File** > **Save Restore Point As**.
2. In the **Name** field, enter a name for the restore point.
3. In the **Description** field, you can add an optional description of the restore point.


HDL Workflow Advisor: Save Model and Data Restore Point - hdlcoderUARTServoControllerExa...

No restore point saved.

Name	Description	Time

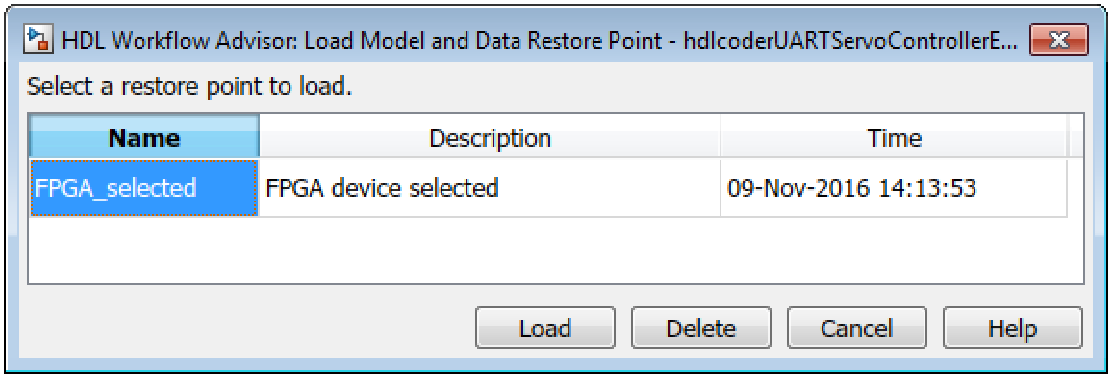
Name: FPGA_selected Description: FPGA device selected

4 Click **Save**. The HDL Workflow Advisor saves a restore point of the current settings.

Restore the HDL Workflow Advisor State

To load a restore point:

1 In the HDL Workflow Advisor, select **File** > **Load Restore Point**.


HDL Workflow Advisor: Load Model and Data Restore Point - hdlcoderUARTServoControllerE...

Select a restore point to load.

Name	Description	Time
FPGA_selected	FPGA device selected	09-Nov-2016 14:13:53

2 Select the restore point that you want.

3 Click **Load**.

The HDL Workflow Advisor issues a warning that the restoration overwrites current settings.

4 Click **Load** to load the restore point you selected. The HDL Workflow Advisor restores the previously saved state.

Related Examples

- “Open and Run Tasks in the HDL Workflow Advisor” on page 22-2
- “Fix a Workflow Advisor Warning or Failure” on page 22-9
- “View and Save HDL Workflow Advisor Reports” on page 22-12
- “HDL Workflow Advisor Tasks” on page 25-2
- “Generate Code Using the HDL Workflow Advisor” on page 22-17

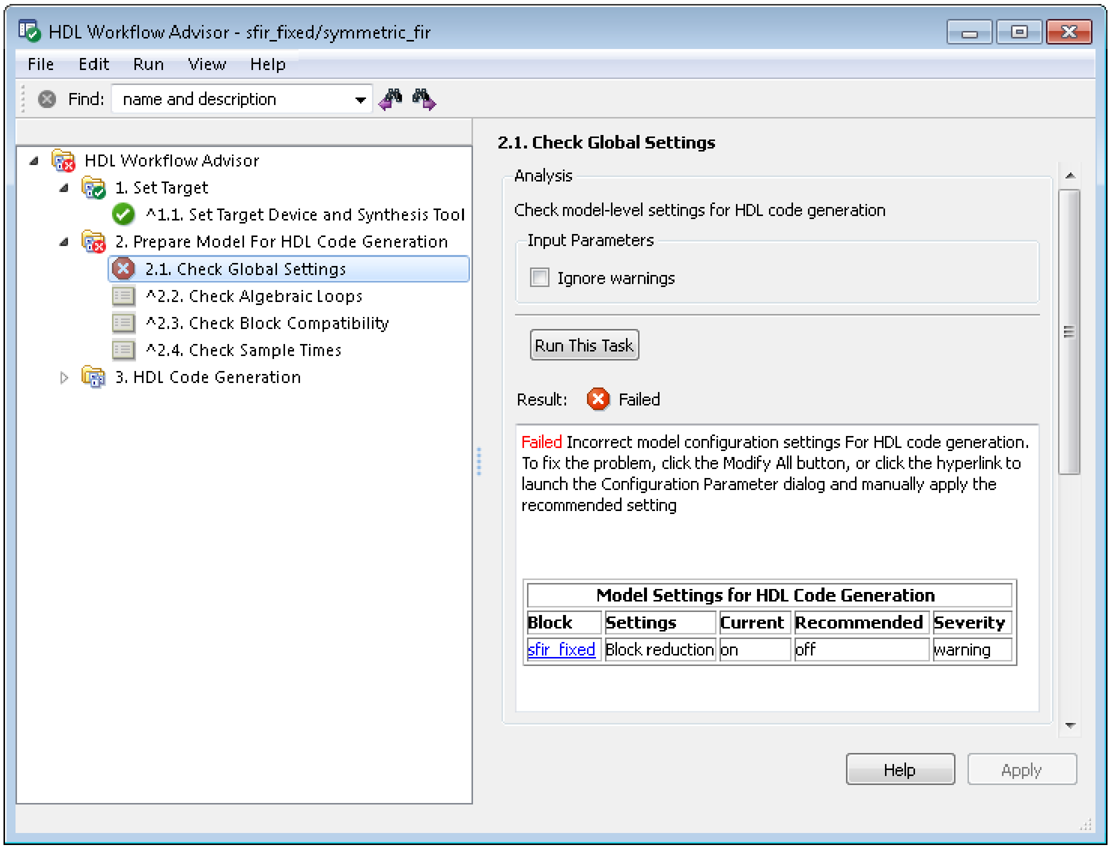
Fix a Workflow Advisor Warning or Failure

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 and other information obtained during synthesis
- Complete automated workflows for selected FPGA development target devices and Simulink Real-Time, including FPGA-in-the-loop simulation

In the HDL Workflow Advisor, if a task terminates due to a warning or failure condition, the right pane shows information about the problems. This information appears in a **Result** subpane. The **Result** subpane also suggests model settings you can use to fix the problems.

Some tasks have an **Action** subpane that lets you apply the recommended actions listed in the **Result** subpane automatically. In the following example, the **Check Global Settings** task has failed, displaying an incorrect model setting in the **Result** pane.


HDL Workflow Advisor - sfir_fixed/symmetric_fir
File Edit Run View Help
Find: name and description
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis Tool
2. Prepare Model For HDL Code Generation
2.1. Check Global Settings
^2.2. Check Algebraic Loops
^2.3. Check Block Compatibility
^2.4. Check Sample Times
3. HDL Code Generation
2.1. Check Global Settings
Analysis
Input Parameters
Ignore warnings
Run This Task
Result: Failed
Help
Apply

Check model-level settings for HDL code generation

Failed Incorrect model configuration settings For HDL code generation. To fix the problem, click the Modify All button, or click the hyperlink to launch the Configuration Parameter dialog and manually apply the recommended setting

Model Settings for HDL Code Generation

Block	Settings	Current	Recommended	Severity
sfir_fixed	Block reduction	on	off	warning

The `Action` subpane, below the **Result** subpane, contains a **Modify All** button. To fix the problems that appear in the **Result** subpane, click the **Modify All** button.

After you click **Modify All**, the **Result** subpane reports the changes that were applied. The task status is reset, enabling you to rerun the task and proceed to the following tasks.

Related Examples

- “Open and Run Tasks in the HDL Workflow Advisor” on page 22-2
- “Save and Restore HDL Workflow Advisor State” on page 22-5
- “View and Save HDL Workflow Advisor Reports” on page 22-12

- “HDL Workflow Advisor Tasks” on page 25-2
- “Generate Code Using the HDL Workflow Advisor” on page 22-17

View and Save HDL Workflow Advisor Reports

In this section...
"Viewing HDL Workflow Advisor Reports" on page 22-12
"Saving HDL Workflow Advisor Reports" on page 22-15

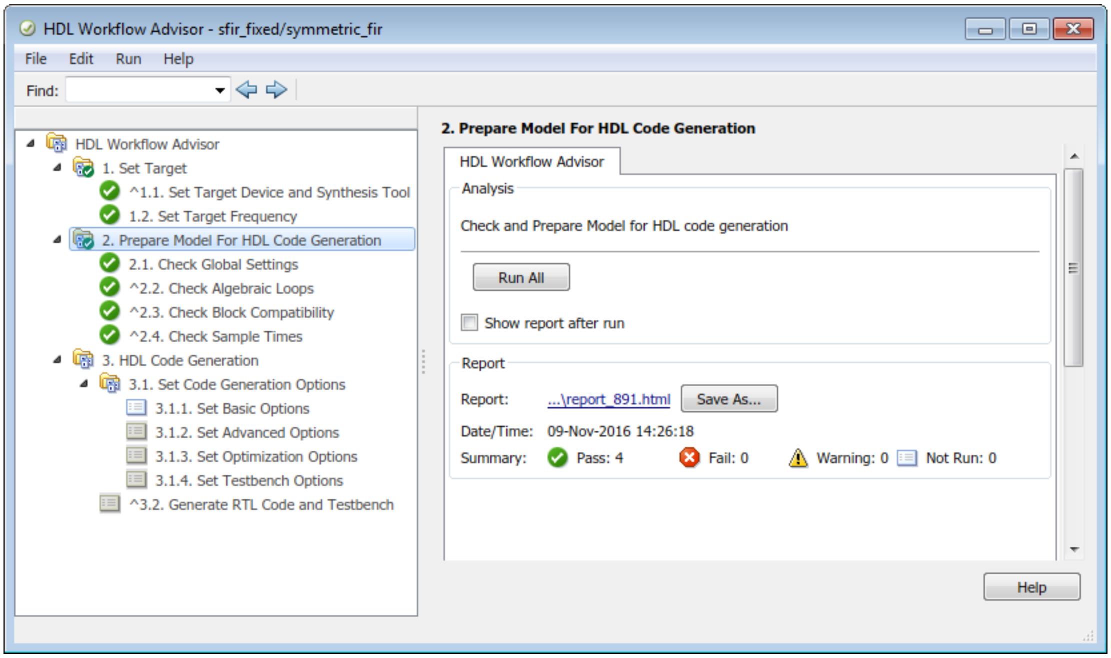
The HDL Workflow Advisor guides you through the stages of generating HDL code for a Simulink subsystem, and the FPGA design process, such as:

- 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 and other information obtained during synthesis
- Complete automated workflows for selected FPGA development target devices and Simulink Real-Time, including FPGA-in-the-loop simulation

Viewing HDL Workflow Advisor Reports

When the HDL Workflow Advisor runs tasks, it automatically generates an HTML report of task results. Each folder in the HDL Workflow Advisor contains a report for the checks within that folder and its subfolders.

You can access reports by selecting a folder and clicking the link in the **Report** subpane. In the following example, the **Prepare Model For HDL Code Generation** folder is selected.


HDL Workflow Advisor - sfir_fixed/symmetric_fir
File
Edit
Run
Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis Tool
1.2. Set Target Frequency
2. Prepare Model For HDL Code Generation
2.1. Check Global Settings
^2.2. Check Algebraic Loops
^2.3. Check Block Compatibility
^2.4. Check Sample Times
3. HDL Code Generation
3.1. Set Code Generation Options
3.1.1. Set Basic Options
3.1.2. Set Advanced Options
3.1.3. Set Optimization Options
3.1.4. Set Testbench Options
^3.2. Generate RTL Code and Testbench
HDL Workflow Advisor
Run All
Save As...
Help

2. Prepare Model For HDL Code Generation

Analysis

Check and Prepare Model for HDL code generation

☐ Show report after run

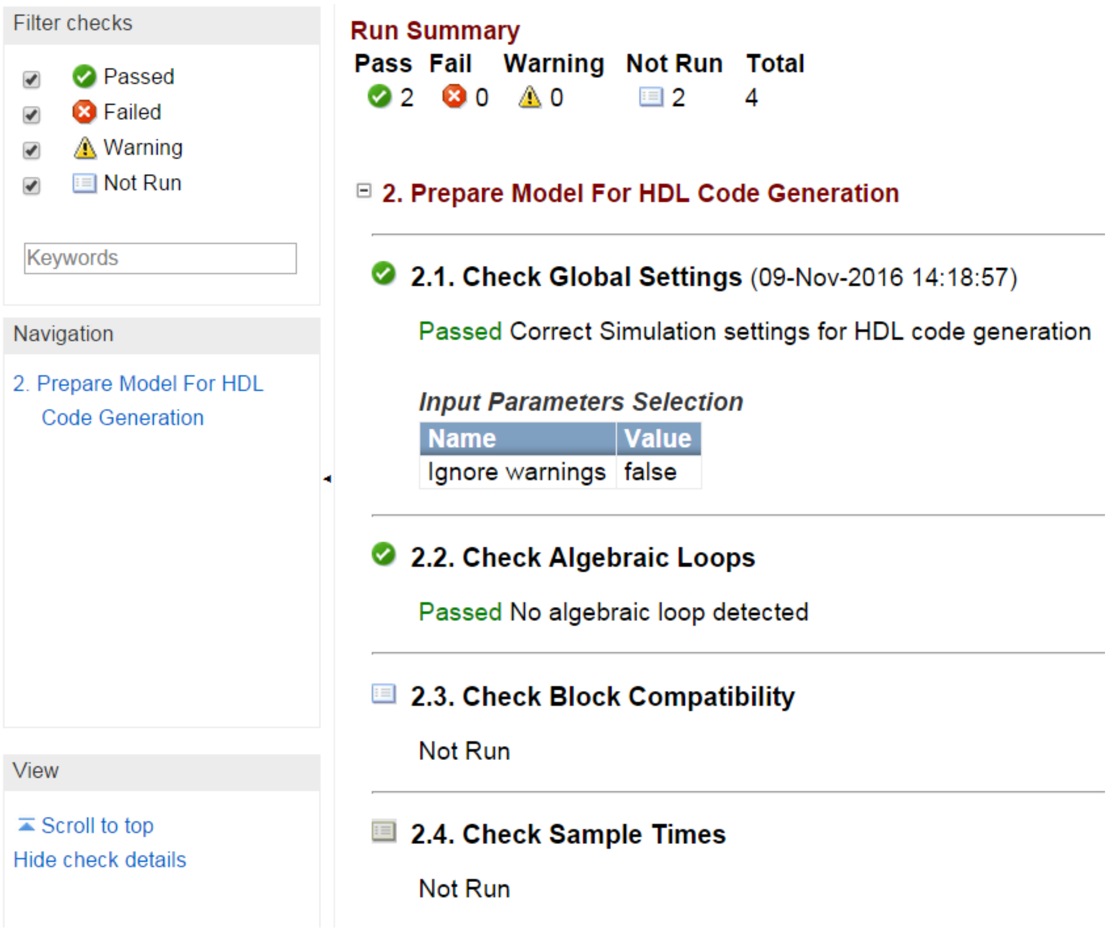
Report

Report: ...\report_891.html

Date/Time: 09-Nov-2016 14:26:18

Summary: ✓ Pass: 4 ✗ Fail: 0 ⚠ Warning: 0 ▤ Not Run: 0

The following report shows typical results for a run of the **Prepare Model For HDL Code Generation** tasks.



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 (09-Nov-2016 14:18:57)

Passed Correct Simulation settings for HDL code generation

Input Parameters Selection

Name	Value
Ignore warnings	false

2.2. Check Algebraic Loops

Passed No algebraic loop detected

2.3. Check Block Compatibility

Not Run

2.4. Check Sample Times

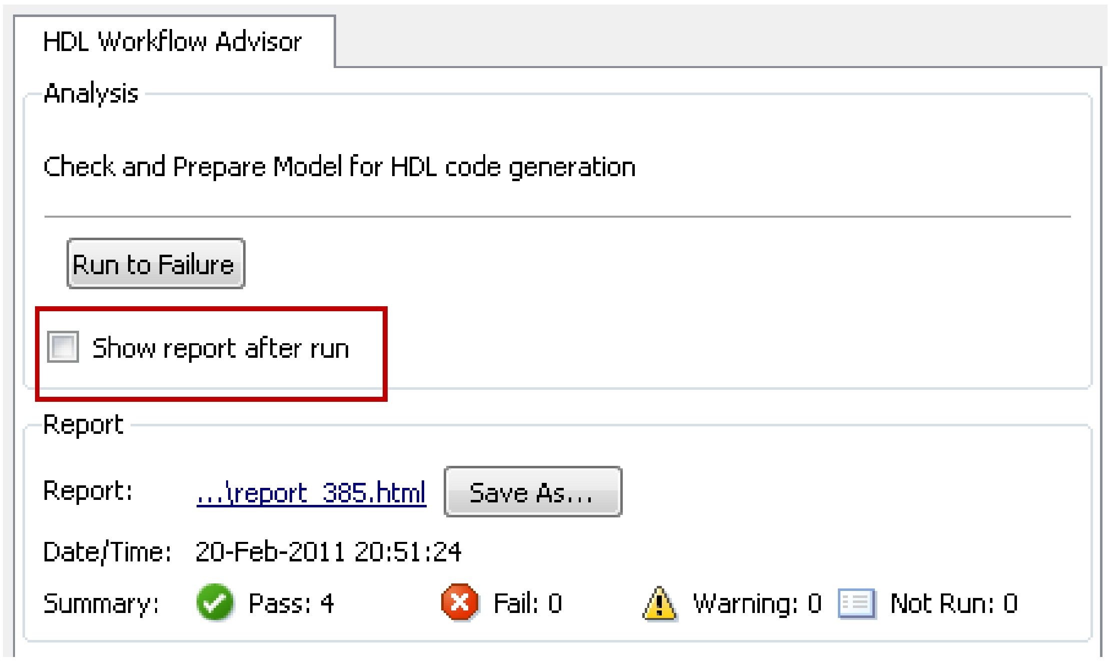
Not Run

As you run checks, the HDL Workflow Advisor updates the reports with the latest information for each check in the folder. A message appears in the report when you run the checks at different times. Timestamps indicate when checks have been run. The time of the current run appears at the top right of the report. Checks that occurred during previous runs have a timestamp following the check name.

You can manipulate the report to show only what you are interested in viewing as follows:

- The check boxes under **Run Summary** allow you to view only the checks with the status that you are interested in viewing. For example, you can remove the checks that have not run by clearing the check box next to the **Not Run** status.
- Minimize folder results in the report by clicking the minus sign next to the folder name. When you minimize a folder, the report updates to display a run summary for that folder.

Select **Show report after run** to view the report for a folder automatically each time the tasks for the folder runs.



Report: ...\report_385.html

Date/Time: 20-Feb-2011 20:51:24

Summary: Pass: 4 Fail: 0 Warning: 0 Not Run: 0

Saving HDL Workflow Advisor Reports

You can archive an HDL Workflow Advisor report by saving it to a new location. To save a report:

1 In the HDL Workflow Advisor, navigate to the folder that contains the report you want to save.

2. Select the folder that you want. The right pane of the HDL Workflow Advisor shows information about that folder, including a **Report** subpane.
3. In the **Report** subpane, click **Save As**.
4. In the Save As dialog box, navigate to the location where you want to save the report, and click **Save**. The HDL Workflow Advisor saves the report to the new location.

Note: If you rerun the HDL Workflow Advisor, the report is updated in the working folder, not in the save location. You can find the full path to the report in the title bar of the report window. Typically, the report is within the working folder: `slprj\modeladvisor\HDLAdv_\`*`model_name`*`\`*`DUT_name`*`\`.

Related Examples

- "Open and Run Tasks in the HDL Workflow Advisor" on page 22-2
- "Fix a Workflow Advisor Warning or Failure" on page 22-9
- "Save and Restore HDL Workflow Advisor State" on page 22-5
- "HDL Workflow Advisor Tasks" on page 25-2
- "Generate Code Using the HDL Workflow Advisor" on page 22-17

Generate Code Using the HDL Workflow Advisor

In this section...

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 and other information obtained during synthesis
- Complete automated workflows for selected FPGA development target devices and Simulink Real-Time, including FPGA-in-the-loop simulation

This example shows how to generate HDL code from a Simulink model using the HDL Workflow Advisor.

Create Working Folder and Copy Model

Prepare the model you use in this example, `sfir_fixed`, for code generation.

This example uses the Xilinx ISE synthesis tool, and assumes that your synthesis tool path is set up. You can also follow this example using Altera Quartus II.

1. Start MATLAB.
2. Create a folder named *`sl_hdlcoder_work`*. For example:

```
mkdir C:\work\sl_hdlcoder_work
```

You use *sl_hdlcoder_work* to store a local copy of the example model and to store folders and code generated by HDL Coder. The location of the folder does not matter, except that it should not be within the MATLAB folder tree.

3. Make the *sl_hdlcoder_work* folder your working folder. For example:

   ```
   cd C:\work\sl_hdlcoder_work
   ```

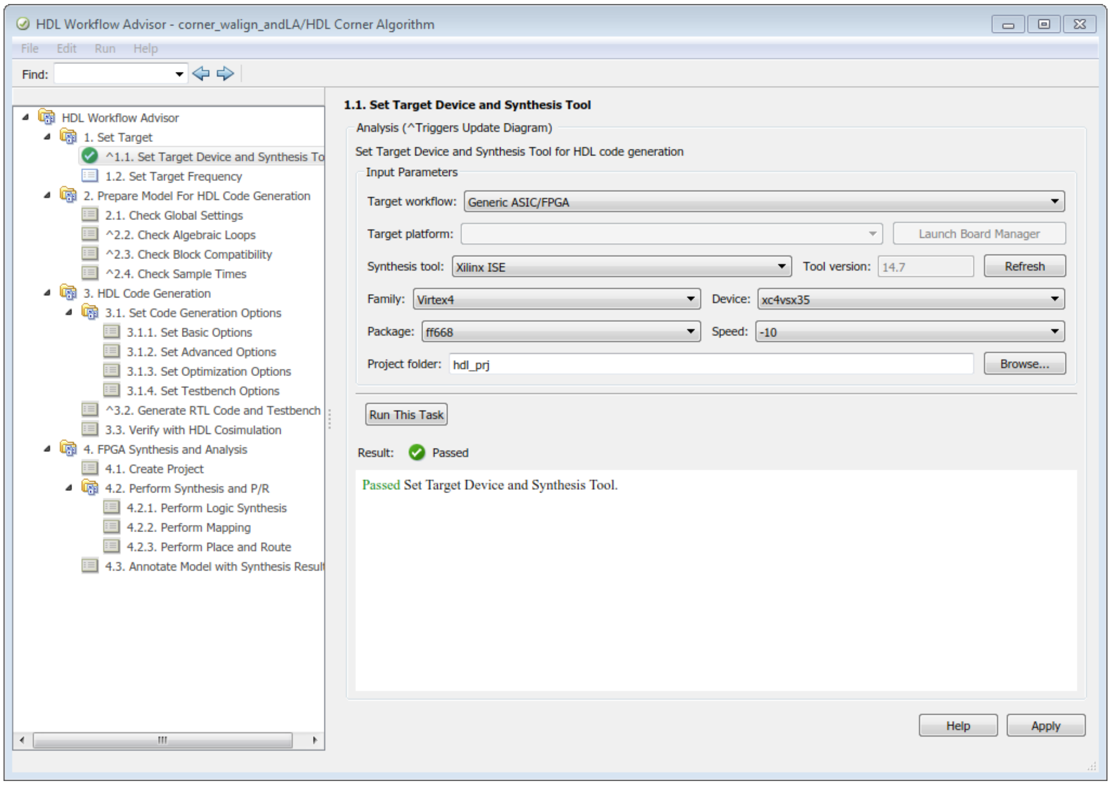
4. Open the `sfir_fixed` model.

   ```
   sfir_fixed
   ```

5. Save a copy of `sfir_fixed` in your *sl_hdlcoder_work* folder.

Generate Code

- Right-click the `symmetric_fir` subsystem and select **HDL Code** > **HDL Workflow Advisor**.
- In the **Set Target** > **Set Target Device and Synthesis Tool** step, for **Synthesis tool**, select **Xilinx ISE** and click **Run This Task**.


HDL Workflow Advisor - corner_walign_andLA/HDL Corner Algorithm
File Edit Run Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis To
1.2. Set Target Frequency
2. Prepare Model For HDL Code Generation
2.1. Check Global Settings
^2.2. Check Algebraic Loops
^2.3. Check Block Compatibility
^2.4. Check Sample Times
3. HDL Code Generation
3.1. Set Code Generation Options
3.1.1. Set Basic Options
3.1.2. Set Advanced Options
3.1.3. Set Optimization Options
3.1.4. Set Testbench Options
^3.2. Generate RTL Code and Testbench
3.3. Verify with HDL Cosimulation
4. FPGA Synthesis and Analysis
4.1. Create Project
4.2. Perform Synthesis and P/R
4.2.1. Perform Logic Synthesis
4.2.2. Perform Mapping
4.2.3. Perform Place and Route
4.3. Annotate Model with Synthesis Resul
Launch Board Manager
Refresh
Browse...
Run This Task
Help
Apply

1.1. Set Target Device and Synthesis Tool

Analysis (^Triggers Update Diagram)

Set Target Device and Synthesis Tool for HDL code generation

Input Parameters

Target workflow: Generic ASIC/FPGA

Target platform:

Synthesis tool: Xilinx ISE Tool version: 14.7

Family: Virtex4 Device: xc4vsx35

Package: ff668 Speed: -10

Project folder: hdl_prj

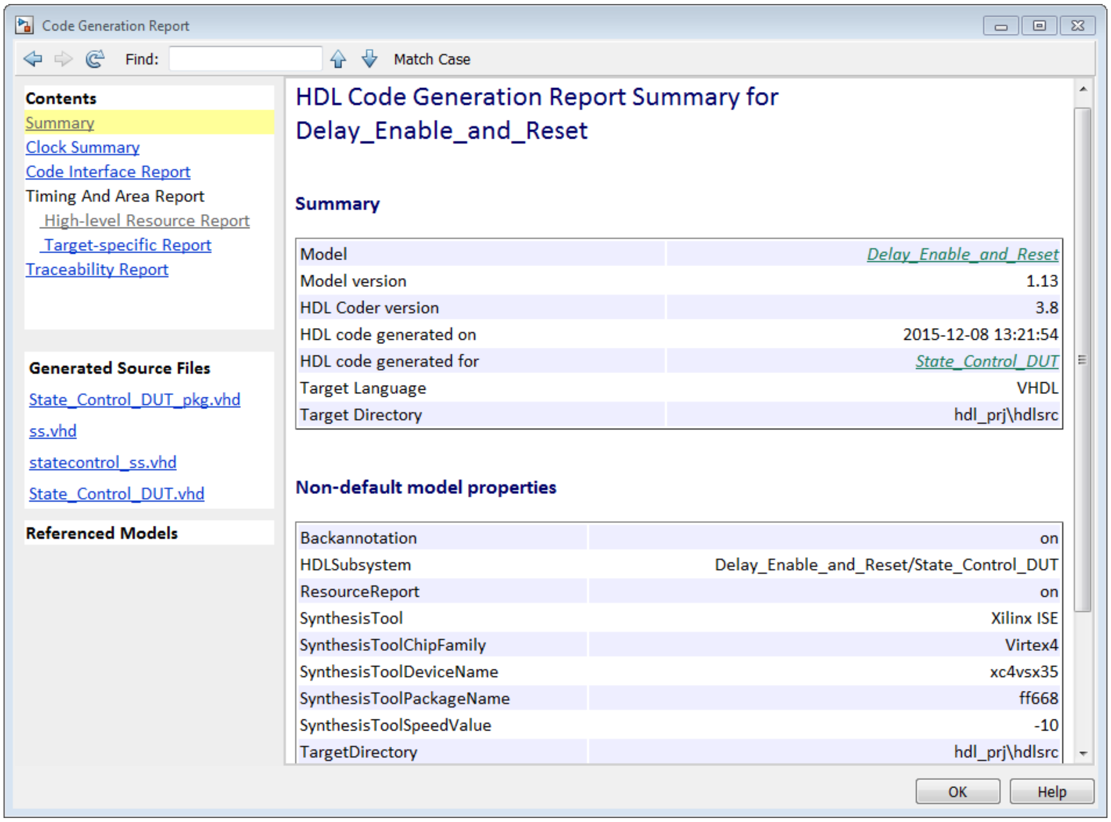
Result: Passed

Passed Set Target Device and Synthesis Tool.

- Right-click **Prepare Model For HDL Code Generation** and select **Run All**. The HDL Workflow Advisor checks the model for code generation compatibility.
- In the **HDL Code Generation** > **Set Code Generation Options** > **Set Basic Options** step, select the following options, then click **Apply**:

  - For **Language**, select **Verilog**.
  - Enable **Generate traceability report**.
  - Enable **Generate resource utilization report**.
- View the options available in the **Optimization** and **Coding style** tabs. You can use these options to modify the implementation and format of the generated code.
- Right-click the **HDL Code Generation** > **Generate RTL Code and Testbench** step, and select **Run to Selected Task**.

The code generation report, which includes the resource utilization and traceability reports, opens automatically. The resource utilization report shows the hardware resources your design implementation is using. The traceability report enables you to navigate between your model and the generated code.



Contents
Summary
Clock Summary
Code Interface Report
Timing And Area Report
High-level Resource Report
Target-specific Report
Traceability Report

Generated Source Files
State_Control_DUT_pkg.vhd
ss.vhd
statecontrol_ss.vhd
State_Control_DUT.vhd

Referenced Models

HDL Code Generation Report Summary for Delay_Enable_and_Reset

Summary

Model	Delay_Enable_and_Reset
Model version	1.13
HDL Coder version	3.8
HDL code generated on	2015-12-08 13:21:54
HDL code generated for	State_Control_DUT
Target Language	VHDL
Target Directory	hdl_prj\hdlsrc

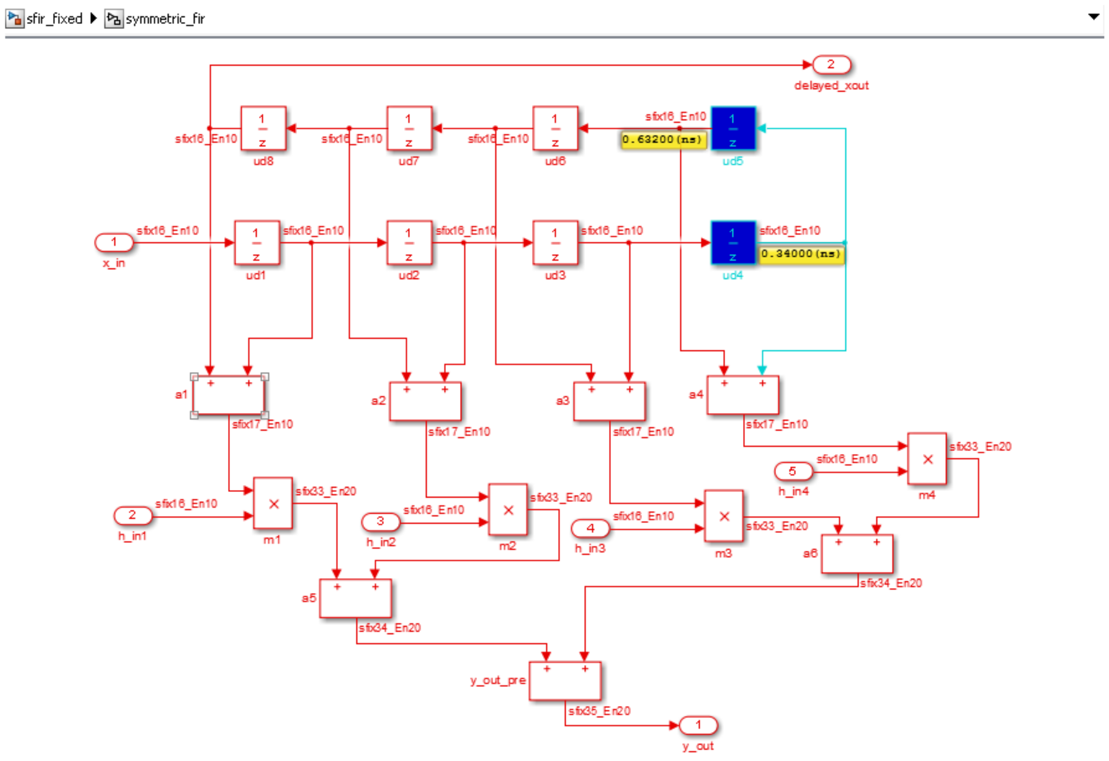
Non-default model properties

Backannotation	on
HDLSubsystem	Delay_Enable_and_Reset/State_Control_DUT
ResourceReport	on
SynthesisTool	Xilinx ISE
SynthesisToolChipFamily	Virtex4
SynthesisToolDeviceName	xc4vsx35
SynthesisToolPackageName	ff668
SynthesisToolSpeedValue	-10
TargetDirectory	hdl_prj\hdlsrc

Perform FPGA Synthesis and Analysis

1. In the **FPGA Synthesis and Analysis** > **Perform Synthesis and P/R** > **Perform Place and Route** task, clear **Skip this task** and click **Apply**.
2. Right-click **Annotate Model with Synthesis Result** and select **Run to Selected Task**.
3. View the annotated critical path in the model.

The critical path is colored cyan.



Related Examples

- “Open and Run Tasks in the HDL Workflow Advisor” on page 22-2
- “Save and Restore HDL Workflow Advisor State” on page 22-5
- “Fix a Workflow Advisor Warning or Failure” on page 22-9
- “View and Save HDL Workflow Advisor Reports” on page 22-12
- “HDL Workflow Advisor Tasks” on page 25-2

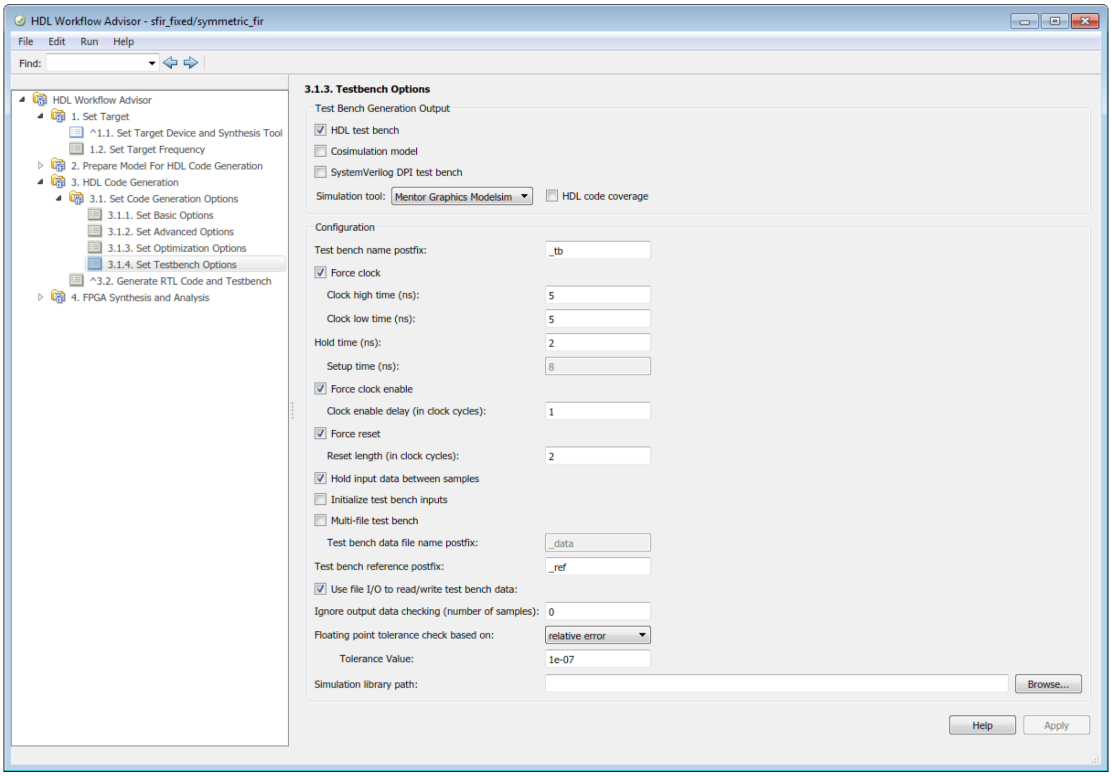
Generate Test Bench and Enable Code Coverage Using the HDL Workflow Advisor

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 and other information obtained during synthesis
- Complete automated workflows for selected FPGA development target devices and Simulink Real-Time, including FPGA-in-the-loop simulation

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 “Generate Code Using the HDL Workflow Advisor” on page 22-17.
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 18-37. After you select your test bench options, click **Apply**.


HDL Workflow Advisor - sfir_fixed/symmetric_fir
File Edit Run Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis Tool
1.2. Set Target Frequency
2. Prepare Model For HDL Code Generation
3. HDL Code Generation
3.1. Set Code Generation Options
3.1.1. Set Basic Options
3.1.2. Set Advanced Options
3.1.3. Set Optimization Options
3.1.4. Set Testbench Options
^3.2. Generate RTL Code and Testbench
4. FPGA Synthesis and Analysis
Browse...
Help
Apply

3.1.3. Testbench Options

Test Bench Generation Output

- [✓] HDL test bench
- [ ] Cosimulation model
- [ ] SystemVerilog DPI test bench

Simulation tool: Mentor Graphics Modelsim [ ] HDL code coverage

Configuration

Test bench name postfix:	_tb
[✓] Force clock	
Clock high time (ns):	5
Clock low time (ns):	5
Hold time (ns):	2
Setup time (ns):	8
[✓] Force clock enable	
Clock enable delay (in clock cycles):	1
[✓] Force reset	
Reset length (in clock cycles):	2
[✓] Hold input data between samples	
[ ] Initialize test bench inputs	
[ ] Multi-file test bench	
Test bench data file name postfix:	_data
Test bench reference postfix:	_ref
[✓] Use file I/O to read/write test bench data:	
Ignore output data checking (number of samples):	0
Floating point tolerance check based on:	relative error
Tolerance Value:	1e-07
Simulation library path:	

3 In Step 3.2, **Generate RTL Code and Testbench**, select **Generate testbench**. 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.

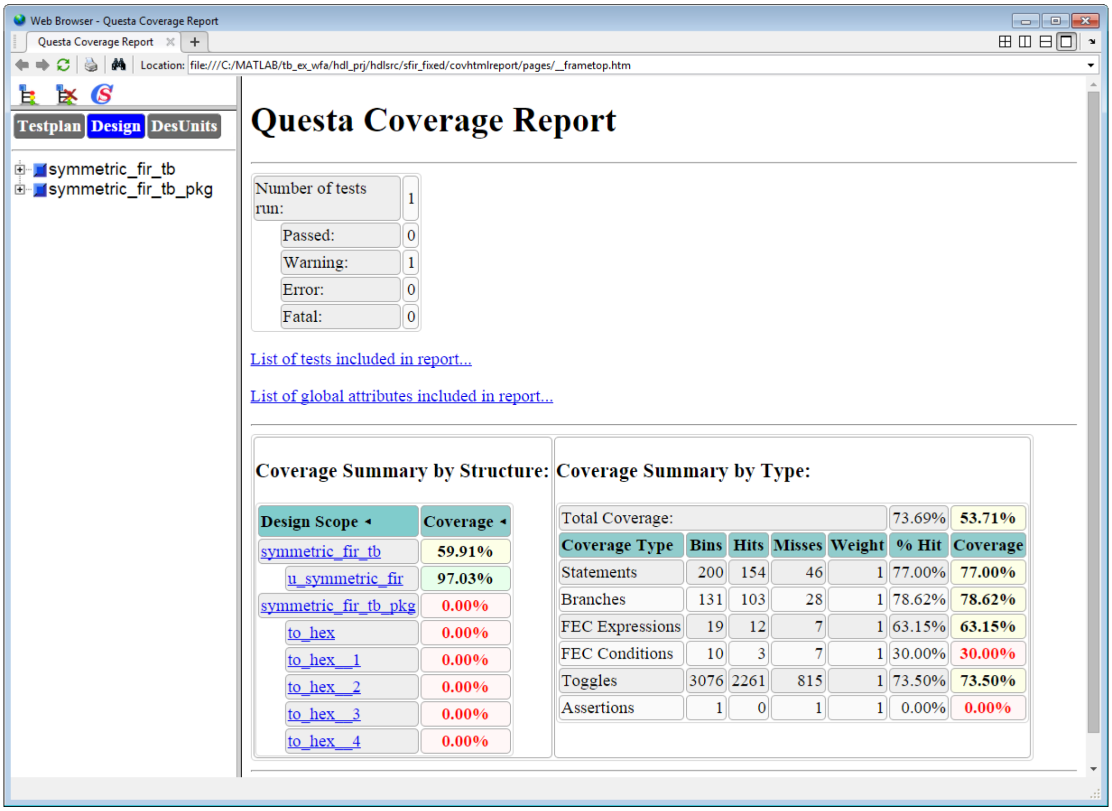
- If you selected **Cosimulation model**, then step 3.3, **Verify with HDL Cosimulation**, appears in the HDL Workflow Advisor. This step automatically runs the generated cosimulation model. The model compares the result of the HDL code running in your HDL simulator with the output of your Simulink subsystem.
- 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`.



Web Browser - Questa Coverage Report
Questa Coverage Report
Location: file:///C:/MATLAB/tb_ex_wfa/hdl_prj/hdlsrc/sfir_fixed/covhtmlreport/pages/__frametop.htm
Testplan
Design
DesUnits
symmetric_fir_tb
symmetric_fir_tb_pkg

Questa Coverage Report

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
symmetric_fir_tb	59.91%
u_symmetric_fir	97.03%
symmetric_fir_tb_pkg	0.00%
to_hex	0.00%
to_hex__1	0.00%
to_hex__2	0.00%
to_hex__3	0.00%
to_hex__4	0.00%

Coverage Summary by Type:

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Total Coverage:					73.69%	53.71%
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%

More About

- “HDL Code Generation Pane: Test Bench” on page 11-122
- “Choose a Test Bench for Generated HDL Code” on page 18-37

Generate HDL Code for FPGA Floating-Point Target Libraries

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 22-44.

In this section...
"Setup for FPGA Floating-Point Library Mapping" on page 22-26
"Map to an FPGA Floating-Point Library" on page 22-27
"View Code Generation Reports of Floating-Point Library Mapping" on page 22-29
"Analyze Results of Floating-Point Library Mapping" on page 22-31

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 `hdlsetuptoolpath`. For example, to set the path to the `Altera Quartus II` synthesis tool:

  ```
  hdlsetuptoolpath('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”.

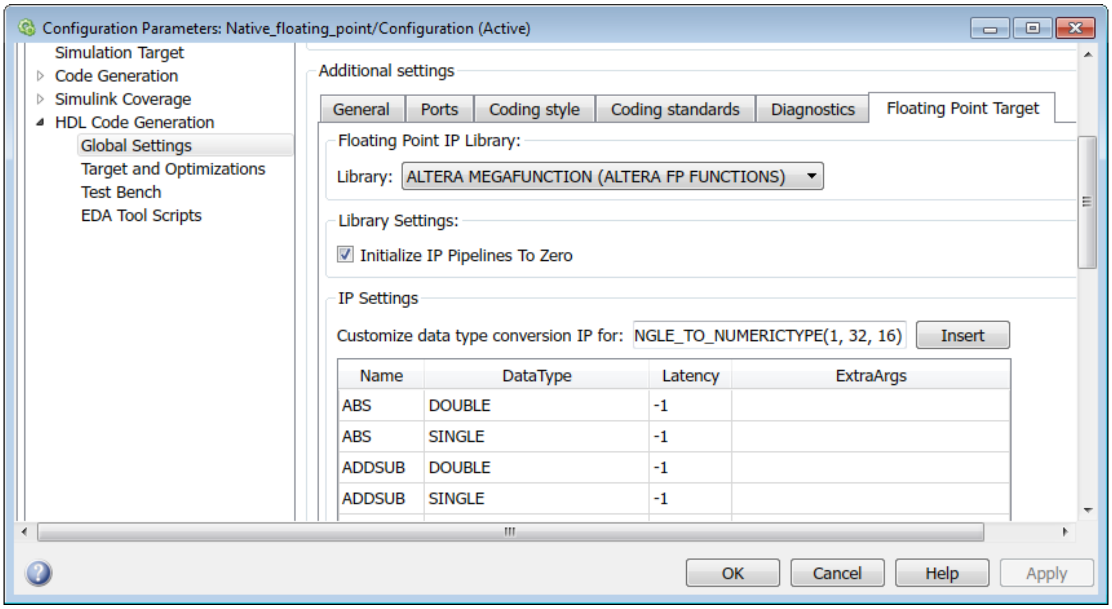
Map to an FPGA Floating-Point Library

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. Open the Configuration Parameters dialog box.
2. In the **HDL Code Generation** > **Global Settings** > **Floating Point Target** tab, select the floating-point IP library.


Configuration Parameters: Native_floating_point/Configuration (Active)

Additional settings

Floating Point IP Library:

Library: ALTERA MEGAFUNCTION (ALTERA FP FUNCTIONS)

Library Settings:

☑ Initialize IP Pipelines To Zero

IP Settings

Customize data type conversion IP for: NGLE_TO_NUMERICTYPE(1, 32, 16)

Name	DataType	Latency	ExtraArgs
ABS	DOUBLE	-1	
ABS	SINGLE	-1	
ADDSUB	DOUBLE	-1	
ADDSUB	SINGLE	-1	

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 22-35.

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 22-39.

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**. You can now generate HDL code from the Simulink model (see “HDL Code Generation from a Simulink Model”).

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('ALTERAFPFUNCTIONS');
```

```
hdlset_param('sfir_single', 'FloatingPointTargetConfiguration', fpconfig);
```

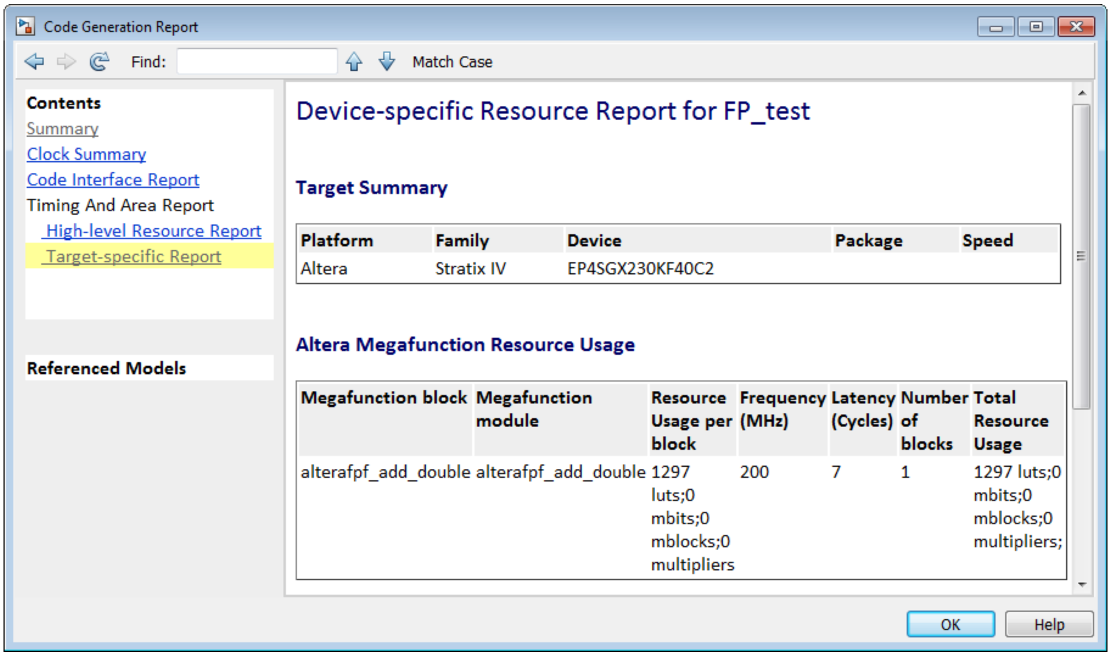
2. You can customize the IP settings based on the floating-point library that you specify. For more information, see "Customize Floating-Point IP Configuration" on page 22-34.
3. Use `makehdl` to generate HDL code from the subsystem.

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 16-2.

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**.



Device-specific Resource Report for FP_test

Target Summary

Platform	Family	Device	Package	Speed
Altera	Stratix IV	EP4SGX230KF40C2		

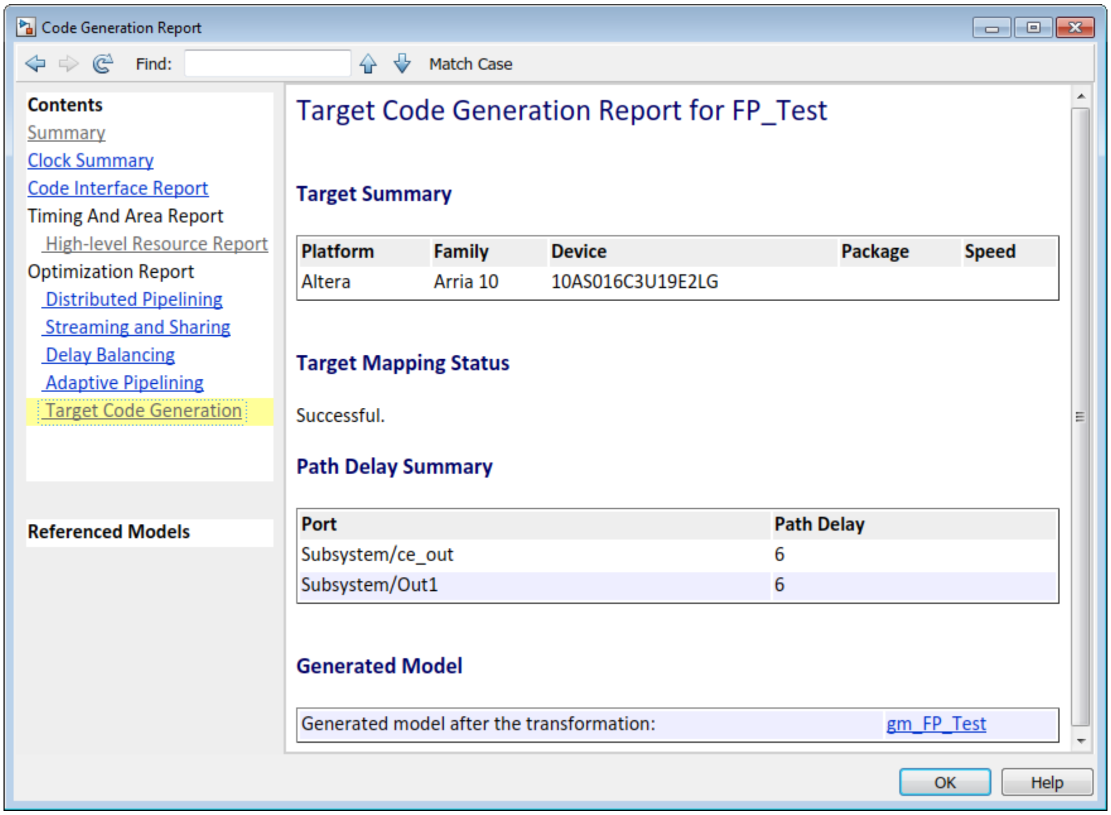
Altera Megafunction Resource Usage

Megafunction block	Megafunction module	Resource Usage per block	Frequency (MHz)	Latency (Cycles)	Number of blocks	Total Resource Usage
alterafpf_add_double	alterafpf_add_double	1297 luts;0 mbits;0 mblocks;0 multipliers	200	7	1	1297 luts;0 mbits;0 mblocks;0 multipliers;

To learn more about the Resource Utilization report, see “Resource Utilization Report” on page 16-4.

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.



Target Code Generation Report for FP_Test

Target Summary

Platform	Family	Device	Package	Speed
Altera	Arria 10	10AS016C3U19E2LG		

Target Mapping Status

Successful.

Path Delay Summary

Port	Path Delay
Subsystem/ce_out	6
Subsystem/Out1	6

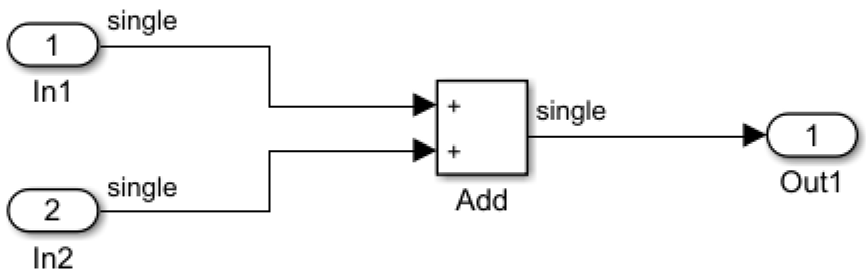
Generated Model

Generated model after the transformation:	gm_FP_Test

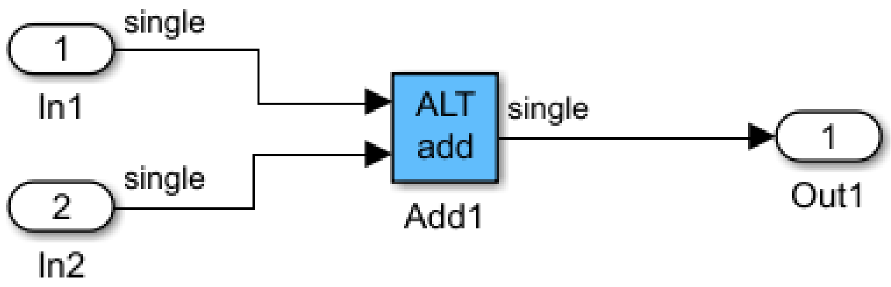
To learn more about the Optimization report, see “Optimization Report” on page 16-9.

Analyze Results of Floating-Point Library Mapping

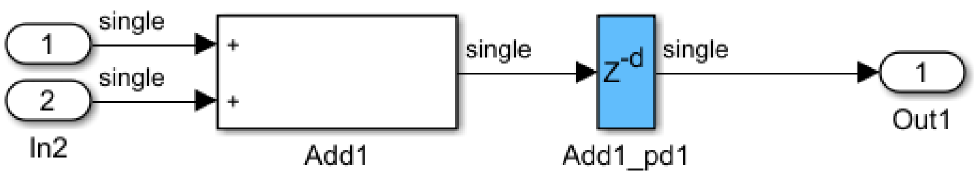
You can get the latency information of the floating-point target IP from the generated model after HDL code generation. For example, consider this add block in Simulink with inputs of double data type.



1 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.

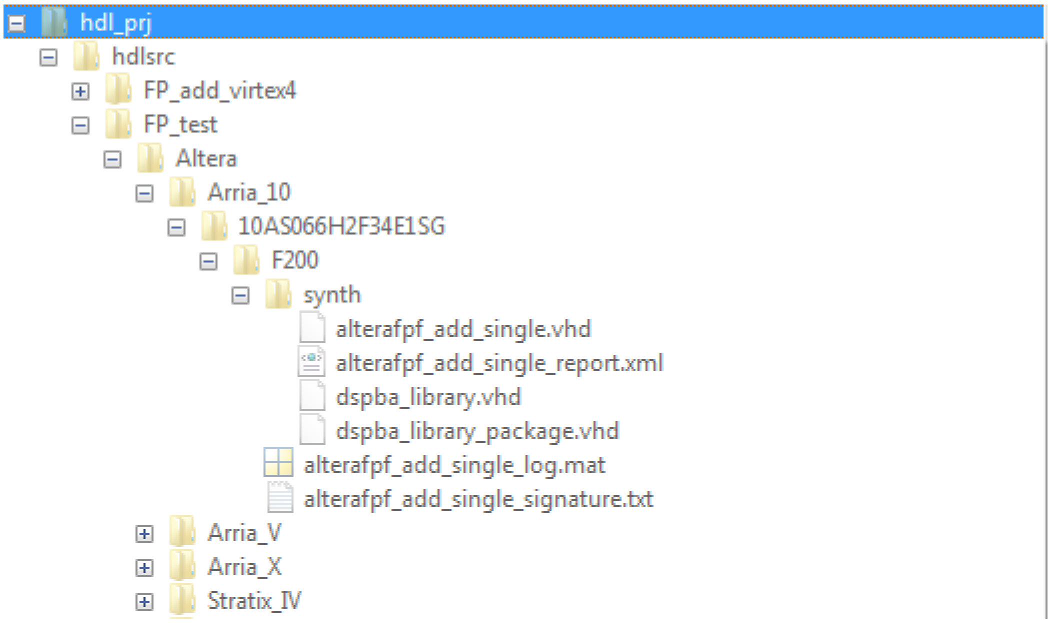


2 Double-click the **ALT add** block. The length of the delay block is the latency of the floating-point target IP.



To learn more about the generated model, see "Generated Model and Validation Model" on page 14-2.

To see your FPGA floating-point library mapping results, you can view the IP core files generated after HDL code generation.



```
hdl_prj
  hdlsrc
    FP_add_virtex4
    FP_test
      Altera
        Arria_10
          10AS066H2F34E1SG
            F200
              synth
                alterafpf_add_single.vhd
                alterafpf_add_single_report.xml
                dspba_library.vhd
                dspba_library_package.vhd
              alterafpf_add_single_log.mat
              alterafpf_add_single_signature.txt
        Arria_V
        Arria_X
        Stratix_IV
```

HDL Coder checks and reuses existing generated IP core files, taking less time when successively generating code for the same floating-point target IP.

Related Examples

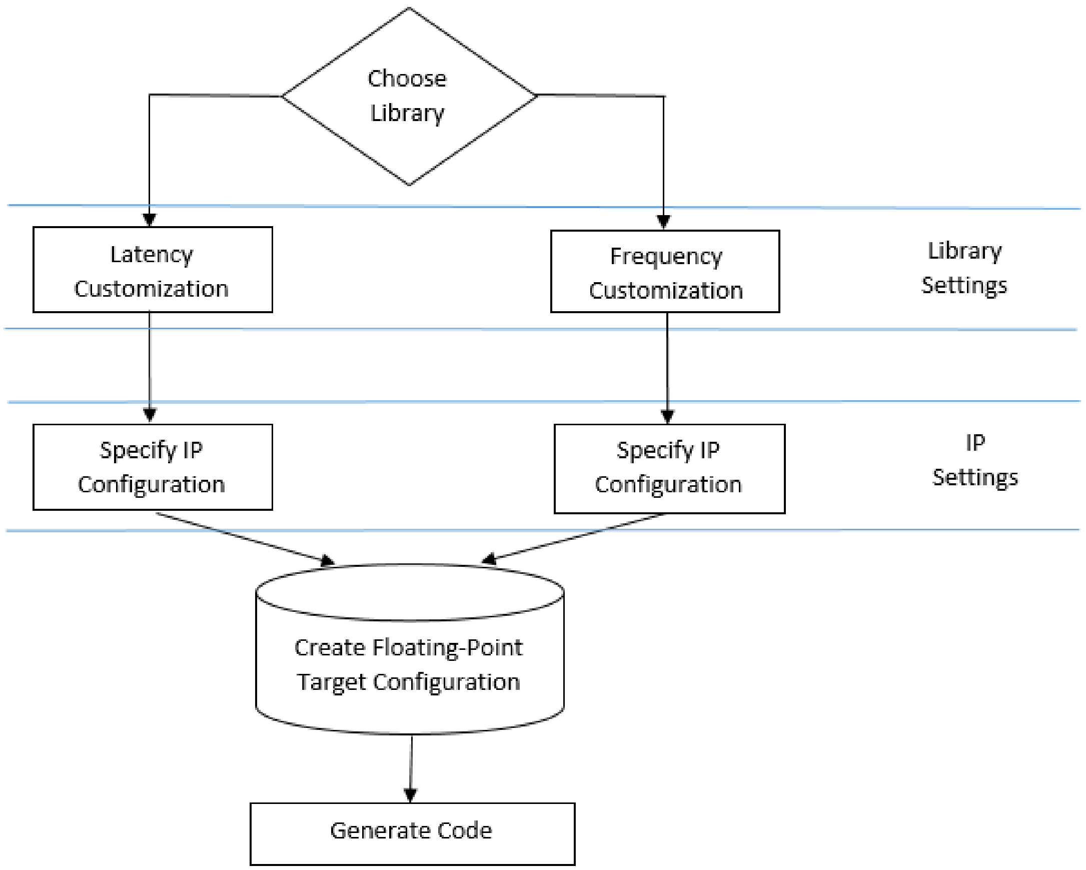
-

More About

- “Customize Floating-Point IP Configuration” on page 22-34
- “HDL Coder Support for FPGA Floating-Point Library Mapping” on page 22-44

Customize Floating-Point IP Configuration

When mapping your Simulink model to floating-point target libraries, you can create a floating-point target configuration with your own custom IP settings. To customize the IP settings, you can use an IP configuration table to choose from different combinations of IP names and data types. The table contains a list of IP types and additional columns that you can use to specify your own custom latency value and other IP settings.



The IP configuration depends on the library settings. The library settings are specific to the floating-point library that you choose. You can customize the IP latency by using the target frequency or the latency strategy setting.

In this section...

Customize the IP Latency with Target Frequency

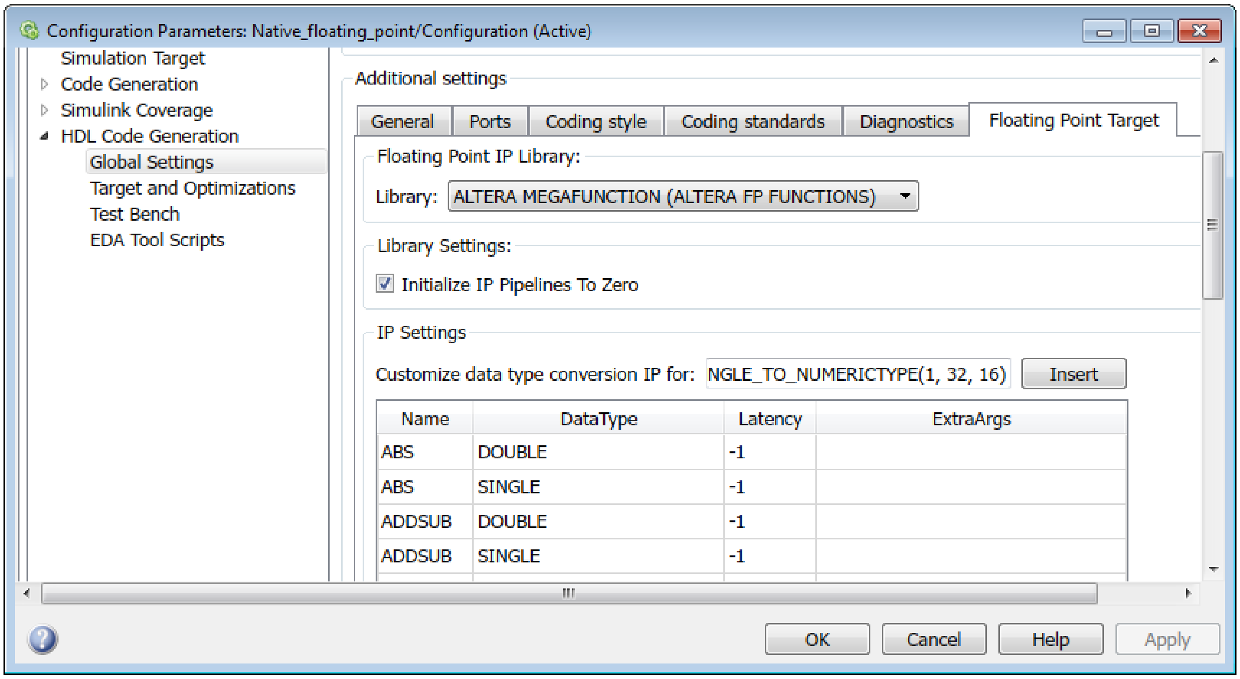
To specify the target frequency that you want the IP to achieve, use the `Altera Megafunctions (ALTERA FP Functions)` library. HDL Coder infers the latency of the IP based on the target frequency value. If you do not specify the target frequency, HDL Coder sets the target frequency to a default value of `200 MHz`.

You can customize the IP latency by using the **Target Frequency** setting in the Configuration Parameters dialog box or the `TargetFrequency` property from the command line.

From the Configuration Parameters Dialog Box

To customize the IP latency by using the target frequency setting:

1. Specify the library: In the Configuration Parameters dialog box, on the **HDL Code Generation** > **Global Settings** > **Floating Point Target** tab, for **Library**, select `Altera Megafunctions (ALTERA FP Functions)`.



Floating Point IP Library:

Library: ALTERA MEGAFUNCTION (ALTERA FP FUNCTIONS)

Library Settings:

☑ Initialize IP Pipelines To Zero

IP Settings

Customize data type conversion IP for: NGLE_TO_NUMERICTYPE(1, 32, 16)

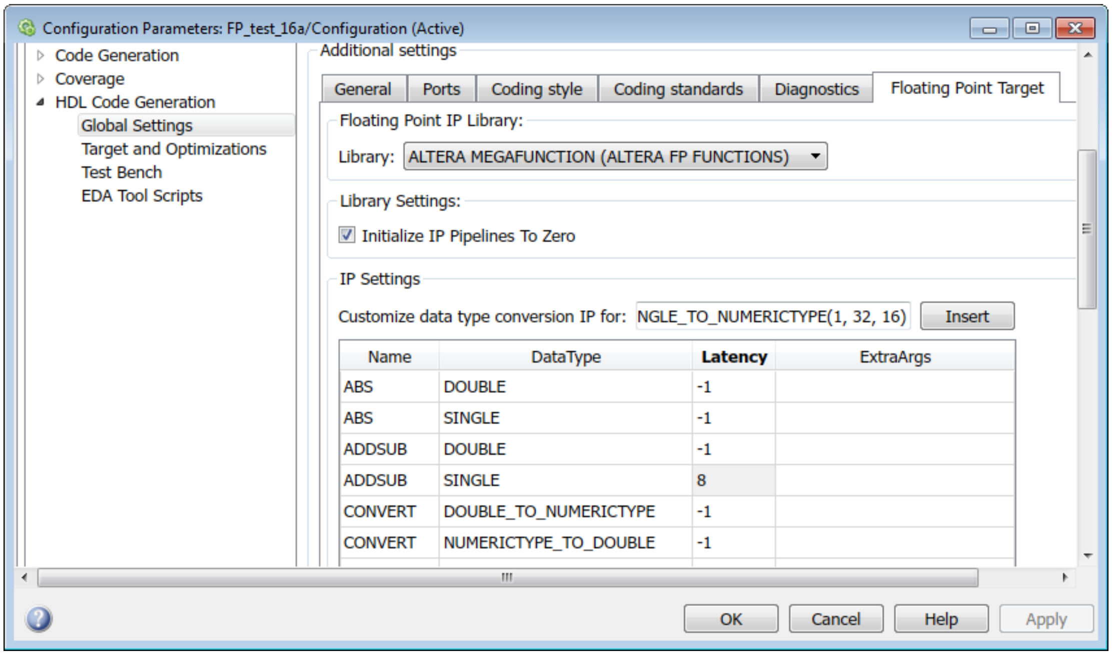
Name	DataType	Latency	ExtraArgs
ABS	DOUBLE	-1	
ABS	SINGLE	-1	
ADDSUB	DOUBLE	-1	
ADDSUB	SINGLE	-1	

2 Specify the target frequency: In the **Target and Optimizations** pane, for **Target Frequency (MHz)**, enter the target frequency that you want the floating-point IP to achieve. If you do not specify a target frequency, HDL Coder sets the target frequency to a default value of `200 MHz`.

3 Specify the library settings: By using the **Initialize IP Pipelines to Zero** option, you can specify whether to initialize pipeline registers in the IP to zero. To avoid potential numerical mismatches in the HDL simulation, it is recommended to leave the **Initialize IP Pipelines to Zero** option set to `true`.

4 Specify the IP settings: In the IP configuration table, you can optionally specify a custom latency and any additional settings specific to the IP.

- In the **Latency** column of the table, the default latency value of `–1` means that the IP inherits the latency value from the target frequency. If you specify a latency value, HDL Coder tries to map your Simulink model to the IP at a target frequency corresponding to that latency value.
- In the **Extra Args** column of the table, you can specify additional settings specific to the IP.


Configuration Parameters: FP_test_16a/Configuration (Active)

Floating Point IP Library:

Library: ALTERA MEGAFUNCTION (ALTERA FP FUNCTIONS)

Library Settings:

☑ Initialize IP Pipelines To Zero

IP Settings

Customize data type conversion IP for: NGLE_TO_NUMERICTYPE(1, 32, 16)

Name	DataType	**Latency**	ExtraArgs
ABS	DOUBLE	-1	
ABS	SINGLE	-1	
ADDSUB	DOUBLE	-1	
ADDSUB	SINGLE	8	
CONVERT	DOUBLE_TO_NUMERICTYPE	-1	
CONVERT	NUMERICTYPE_TO_DOUBLE	-1	

5 Generate code: Click **Apply**. You can now generate HDL code for this floating-point configuration from the Simulink model (see “HDL Code Generation from a Simulink Model”).

From the Command Line

To customize the IP latency from the command line:

1 Specify the library: Create a `hdlcoder.FloatingPointTargetConfig` object for the floating-point library by using the `hdlcoder.createFloatingPointTargetConfig` function. Then, use `hdlset_param` to save the configuration on the model.

For example, for an `sfir_single` model, to create a floating-point target configuration for the `Altera Megafunctions (ALTERA FP FUNCTIONS)` library with the default settings, enter:

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('ALTERAFPFUNCTIONS');
hdlset_param('sfir_single', 'FloatingPointTargetConfiguration', fpconfig);
```
To see the default settings for the floating-point IP, enter `fpconfig`.

```
fpconfig =

  FloatingPointTargetConfig with properties:

             Library: 'ALTERAFPFUNCTIONS'
     LibrarySettings: [1×1 fpconfig.FrequencyDrivenMode]
            IPConfig: [1×1 hdlcoder.FloatingPointTargetConfig.IPConfig]
```

2. Specify the target frequency: If you choose `ALTERA MEGAFUNCTION (ALTERA FP FUNCTIONS)` as the library, you can create a floating-point configuration with a custom target frequency. To specify the target frequency for the IP to achieve, use the `TargetFrequency` property. For example:

   ```
   hdlset_param('sfir_single', 'TargetFrequency', 300);
   ```

3. Specify the library settings: Specify whether you want to initialize the pipeline registers in the IP to zero. Use the `InitializeIPPipelinesToZero` property of the `fpconfig.LibrarySettings` function.

   For example, to set the `InitializeIPPipelinesToZero` property to false, enter:

   ```
   fpconfig.LibrarySettings.InitializeIPPipelinesToZero = false;
   ```
   To see the library settings that you have applied, enter `fpconfig.LibrarySettings`.

   ```
   ans =

     FrequencyDrivenMode with properties:

       InitializeIPPipelinesToZero: 0
   ```

   To avoid potential numerical mismatches in the HDL simulation, it is recommended to leave `InitializeIPPipelinesToZero` set to `true`.

4. Specify the IP settings: With the `IPConfig` method, use the `Latency` and `ExtraArgs` to customize the latency of the IP and specify any additional settings specific to the IP.

For example, when mapping to the **ADDSUB** IP with Xilinx LogiCORE libraries, to specify a custom latency of 8:

```
fpconfig.IPConfig.customize('ADDSUB', 'SINGLE', 'Latency', 8);
```
To see the IP settings that you have applied, enter `fpconfig.IPConfig`.

```
ans = 

       Name                 DataType              Latency    ExtraArgs
    __________    __________________________    _______    __________

    'ABS'         'DOUBLE'                      -1         ''
    'ABS'         'SINGLE'                      -1         ''
    'ADDSUB'      'DOUBLE'                      -1         ''
    'ADDSUB'      'SINGLE'                       8         ''
    'CONVERT'     'DOUBLE_TO_NUMERICTYPE'       -1         ''
    'CONVERT'     'NUMERICTYPE_TO_DOUBLE'       -1         ''
    'CONVERT'     'NUMERICTYPE_TO_SINGLE'       -1         ''
    'CONVERT'     'SINGLE_TO_NUMERICTYPE'       -1         ''
```

5 Generate HDL code: To generate code from the subsystem, use `makehdl`.

Customize the IP Latency with Latency Strategy

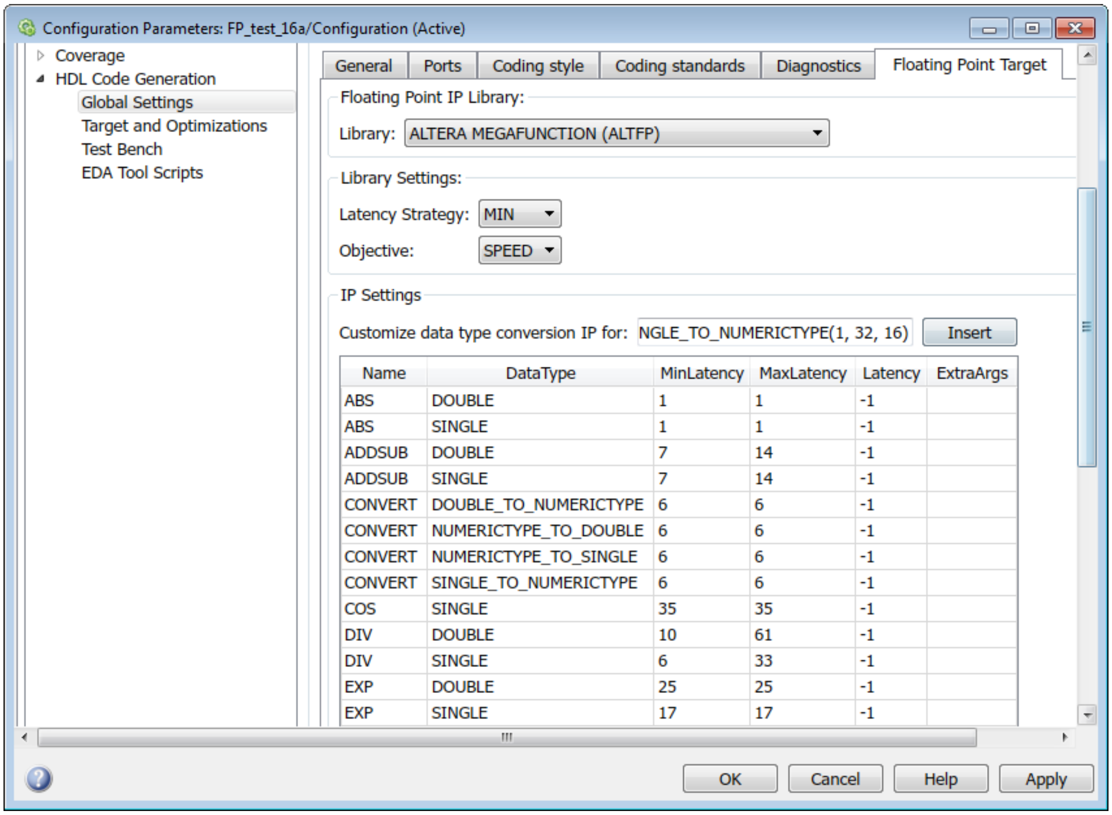
To customize the IP latency with the latency strategy setting, use the `ALTERA MEGAFUNCTION (ALTFP)` or `XILINX LOGICORE` libraries. Specify whether to map your Simulink model to maximum or minimum latency. HDL Coder infers the latency of the IP from the latency strategy setting.

You can customize the IP latency in the Configuration Parameters dialog box or from the command line.

From the Configuration Parameters Dialog Box

To customize the IP latency with the latency strategy setting:

1 Specify the library: In the Configuration Parameters dialog box, on the **HDL Code Generation** > **Global Settings** > **Floating Point Target** tab, for **Library**, select `ALTERA MEGAFUNCTION (ALTFP)` or `XILINX LOGICORE`.



Floating Point IP Library:

Library: ALTERA MEGAFUNCTION (ALTFP)

Library Settings:

Latency Strategy: MIN

Objective: SPEED

IP Settings

Customize data type conversion IP for: NGLE_TO_NUMERICTYPE(1, 32, 16)

Name	DataType	MinLatency	MaxLatency	Latency	ExtraArgs
ABS	DOUBLE	1	1	-1	
ABS	SINGLE	1	1	-1	
ADDSUB	DOUBLE	7	14	-1	
ADDSUB	SINGLE	7	14	-1	
CONVERT	DOUBLE_TO_NUMERICTYPE	6	6	-1	
CONVERT	NUMERICTYPE_TO_DOUBLE	6	6	-1	
CONVERT	NUMERICTYPE_TO_SINGLE	6	6	-1	
CONVERT	SINGLE_TO_NUMERICTYPE	6	6	-1	
COS	SINGLE	35	35	-1	
DIV	DOUBLE	10	61	-1	
DIV	SINGLE	6	33	-1	
EXP	DOUBLE	25	25	-1	
EXP	SINGLE	17	17	-1	

2 Specify the library settings: For **Latency Strategy**, specify whether to map your Simulink model to the minimum or maximum latency for the IP. For **Objective**, specify whether to optimize for speed or area.

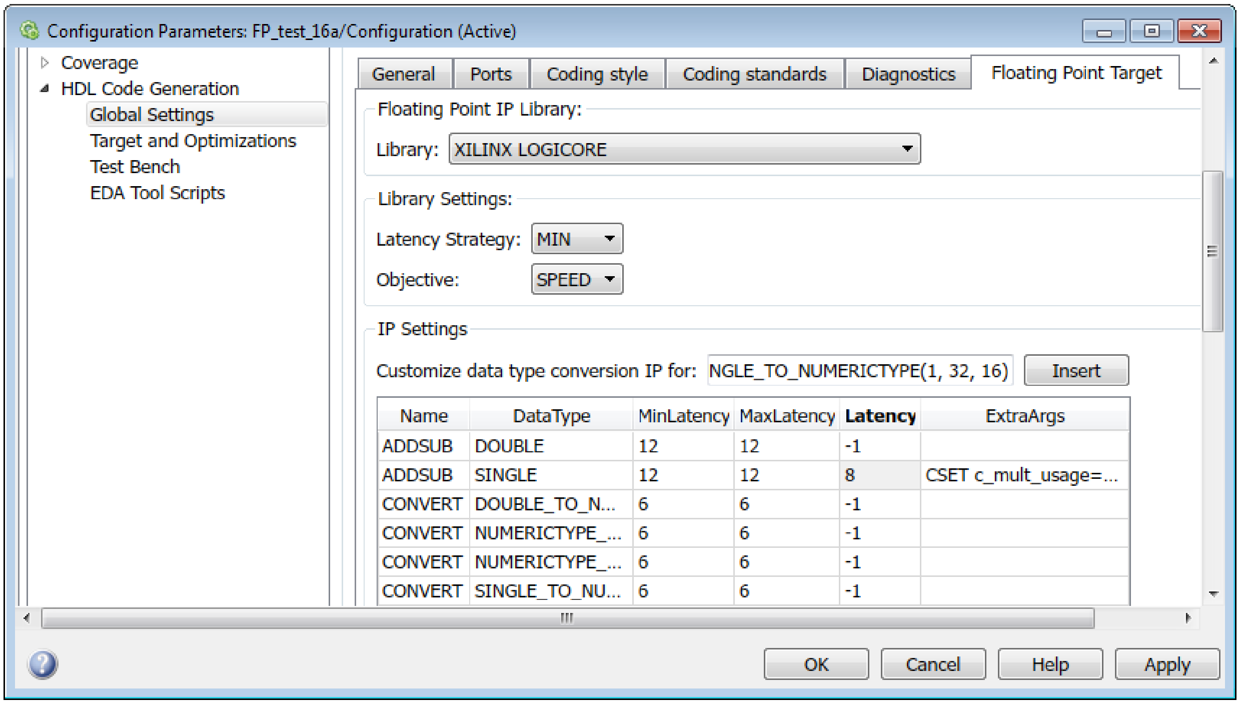
3 Specify the IP settings: An IP configuration table appears that contains the IP types, and their maximum and minimum latencies. In the table, you can optionally specify a custom latency and any additional settings specific to the IP.

- In the **Latency** column of the table, the default latency value of **–1** means that the IP inherits the latency value from the library settings. To customize the latency of the IP that your Simulink blocks map to, enter a value for the latency.

For example, when mapping to the **ADDSUB** IP with Xilinx LogiCORE, if you specify a latency of 8, the latency of the IP changes to 8 instead of the default value of 12.

- In the **Extra Args** column of the table, specify any additional settings specific to the IP.

  For example, when mapping to Xilinx LogiCORE IP, for **Extra Args**, you can specify the parameter **c_mult_usage** to control the DSP resources that you want to use. To learn more about the parameter usage and syntax, see the IP library documentation.



Floating Point IP Library:

Library: XILINX LOGICORE

Library Settings:

Latency Strategy: MIN

Objective: SPEED

IP Settings

Customize data type conversion IP for: NGLE_TO_NUMERICTYPE(1, 32, 16) Insert

Name	DataType	MinLatency	MaxLatency	**Latency**	ExtraArgs
ADDSUB	DOUBLE	12	12	-1	
ADDSUB	SINGLE	12	12	8	CSET c_mult_usage=...
CONVERT	DOUBLE_TO_N...	6	6	-1	
CONVERT	NUMERICTYPE_...	6	6	-1	
CONVERT	NUMERICTYPE_...	6	6	-1	
CONVERT	SINGLE_TO_NU...	6	6	-1	

4 Generate code: Click **Apply**. You can now generate HDL code for this floating-point configuration from the Simulink model (see “HDL Code Generation from a Simulink Model”).

From the Command Line

To customize the IP latency from the command line:

1. Specify the library: Create a `hdlcoder.FloatingPointTargetConfig` object for the floating-point library by using the `hdlcoder.createFloatingPointTargetConfig` function. Then, use `hdlset_param` to save the configuration on the model.

   For example, for an `sfir_single` model, to create a floating-point target configuration for the `ALTERA MEGAFUNCTION (ALTFP)` library with the default settings, enter:

   ```
   fpconfig = hdlcoder.createFloatingPointTargetConfig('ALTFP');
   hdlset_param('sfir_single', 'FloatingPointTargetConfiguration', fpconfig);
   ```
   By default, the library uses the minimum latency and speed objective for the floating-point IP.
2. Specify the library settings: Customize the library settings with the `Objective` and `LatencyStrategy` of the `fpconfig.LibrarySettings` function.

   For example, to customize the `ALTERA MEGAFUNCTION (ALTFP)` library to use the maximum latency and objective as area, enter:

   ```
   fpconfig.LibrarySettings.Objective = 'AREA';
   fpconfig.LibrarySettings.LatencyStrategy = 'MAX';
   ```
   To see the library settings that you have applied, enter `fpconfig.LibrarySettings`.

```
ans = 

  LatencyDrivenMode with properties:

    LatencyStrategy: 'MAX'
          Objective: 'AREA'

 fpconfig is a variable of type hdlcoder.FloatingPointTargetConfig.
```

3 Specify the IP settings: With the `IPConfig` method, use the `Latency` and `ExtraArgs` to customize the latency of the IP and specify any additional settings specific to the IP.

For example, when mapping to the **ADDSUB** IP with Xilinx LogiCORE libraries, to use a custom latency of 8 and to specify the DSP resource usage with the `cmultusage` parameter:

```
fpconfig.IPConfig.customize('ADDSUB', 'SINGLE', 'Latency', 8, 'ExtraArgs', 'CSET c_
```

To see the IP settings that you have applied, enter `fpconfig.IPConfig`.

```
ans =

     Name                DataType              MinLatency    MaxLatency    Latency
   _________    _________________________    __________    __________    _______

   'ADDSUB'     'DOUBLE'                      7             14            -1
   'ADDSUB'     'SINGLE'                      7             14             8
   'CONVERT'    'DOUBLE_TO_NUMERICTYPE'       6              6            -1
   'CONVERT'    'NUMERICTYPE_TO_DOUBLE'       6              6            -1
   'CONVERT'    'NUMERICTYPE_TO_SINGLE'       6              6            -1
```

4 Generate HDL code: To generate code from the subsystem, use `makehdl`.

Related Examples

•

More About

- “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 22-26
- “HDL Coder Support for FPGA Floating-Point Library Mapping” on page 22-44

HDL Coder Support for FPGA Floating-Point Library Mapping

In this section...

In the HDL Coder block library, a subset of Simulink blocks support floating-point library mapping. The subset includes:

- Blocks that perform basic math operations such as addition, multiplication, and complex trigonometric sine and cosine functions. These blocks map to one or more floating-point IP units on the target FPGA device.
- Discrete blocks, blocks that perform signal routing, and blocks that perform math operations such as matrix concatenation. These blocks need not map to a floating-point IP unit on the target FPGA device.

Supported Blocks That Map to FPGA Floating-Point Target IP

The following table summarizes the Simulink blocks that can map to FPGA floating-point IP cores.

When mapping to floating-point IP cores, some blocks have mode restrictions.

Note: Some blocks do not map to a floating-point IP core in the third-party hardware. For example, the Abs block maps to an Altera target IP core but not to a Xilinx target IP core.

Block	Altera Megafunction IP (ALTFP and ALTERA FP Functions)	Xilinx LogiCORE IP	Remarks and Limitations
Abs	✓		—
Add	✓	✓	—
Bias	✓	✓	—

Block	Altera Megafunction IP (ALTFP and ALTERA FP Functions)	Xilinx LogiCORE IP	Remarks and Limitations
Compare To Constant	✓	✓	—
Compare To Zero	✓	✓	—
Data Type Conversion	✓	✓	• Conversions between single and double data types are not supported. • **Integer rounding mode** attribute in the Block Parameters dialog box must be set to `Nearest`.
Decrement Real World	✓	✓	—
Discrete FIR Filter	✓	✓	—
Discrete Transfer Fcn	✓	✓	—
Discrete-Time Integrator	✓	✓	—
Divide	✓	✓	—
Dot Product	✓	✓	
Gain	✓	✓	—
Math Function	✓		• Set the **Function** attribute in Block Parameters dialog box to either `reciprocal`, `log` or `exp`.
MinMax	✓	✓	—
Multiply-Add	✓	✓	—

Block	Altera Megafunction IP (ALTFP and ALTERA FP Functions)	Xilinx LogiCORE IP	Remarks and Limitations
Product	✓	✓	• Product block with more than two inputs is not supported.
Product of Elements	✓	✓	• The **Architecture** in HDL Block Properties must be set to `Tree`.
Reciprocal Sqrt	✓		—
Relational Operator	✓	✓	—
Sqrt	✓	✓	—
Subtract	✓	✓	—
Sum	✓	✓	• Sum block with `-` ports is not supported. • The block cannot have more than two inputs.
Sum of Elements	✓	✓	• The **Architecture** in HDL Block Properties must be set to `Tree`.

Block	Altera Megafunction IP (ALTFP and ALTERA FP Functions)	Xilinx LogiCORE IP	Remarks and Limitations
Trigonometric Function	✓		• Only single data types are supported for floating-point library mapping. • In the Block Parameters dialog box, **Function** must be set to either `sin` or `cos` and **Approximation method** must be set to `None`. • If you are using Altera Quartus 10.1 or 11.0, turn on the `AlteraBackwardIncompatibleSinCosPipeline` global property using `hdlset_param`.
Unary Minus	✓	✓	—

Supported Blocks That Do Not Need to Map to FPGA Floating-Point Target IP

Following are the Simulink blocks that generate HDL code but need not map to an FPGA floating-point IP core.

- Bus Assignment
- Bus Creator
- Bus Selector
- Constant
- Delay

- Demux
- Deserializer1D
- Downsample
- From
- Goto
- Index Vector
- Matrix Concatenate
- Memory
- Model Info
- Multiport Switch
- Mux
- Rate Transition
- Reshape
- Serializer1D
- Subsystem
- Switch block with control input other than `u2 ~= 0`.
- Unit Delay
- Upsample
- Vector Concatenate
- Zero-Order Hold

Limitations for FPGA Floating-Point Library Mapping

- If your synthesis tool is Xilinx Vivado, you cannot use FPGA floating-point library mapping.
- Complex data types are not supported.
- The streaming optimization is not supported with floating-point library mapping.
- The resource sharing optimization is not supported with Unary Minus and Abs blocks.
- For IP Core Generation, FPGA Turnkey, and Simulink Real-Time FPGA I/O workflows, your DUT ports cannot use floating-point data types.

Related Examples

-

More About

- “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 22-26
- “Customize Floating-Point IP Configuration” on page 22-34

FPGA Synthesis and Analysis

In this section...

You can perform FPGA synthesis and analysis with the HDL Workflow Advisor. The HDL Workflow Advisor is a tool that supports and integrates the stages of the FPGA design process starting from checking the model for HDL code generation till running FPGA synthesis on the target synthesis tool. The tasks in the **FPGA Synthesis and Analysis** folder let you run third-party FPGA synthesis and analysis tools without leaving the HDL Workflow Advisor environment. Tasks in this category include:

- Creation of FPGA synthesis projects for supported FPGA synthesis tools
- Launching supported FPGA synthesis tools to perform synthesis, mapping, and place/ route tasks
- Annotation of your original model with critical path information obtained from the synthesis tools

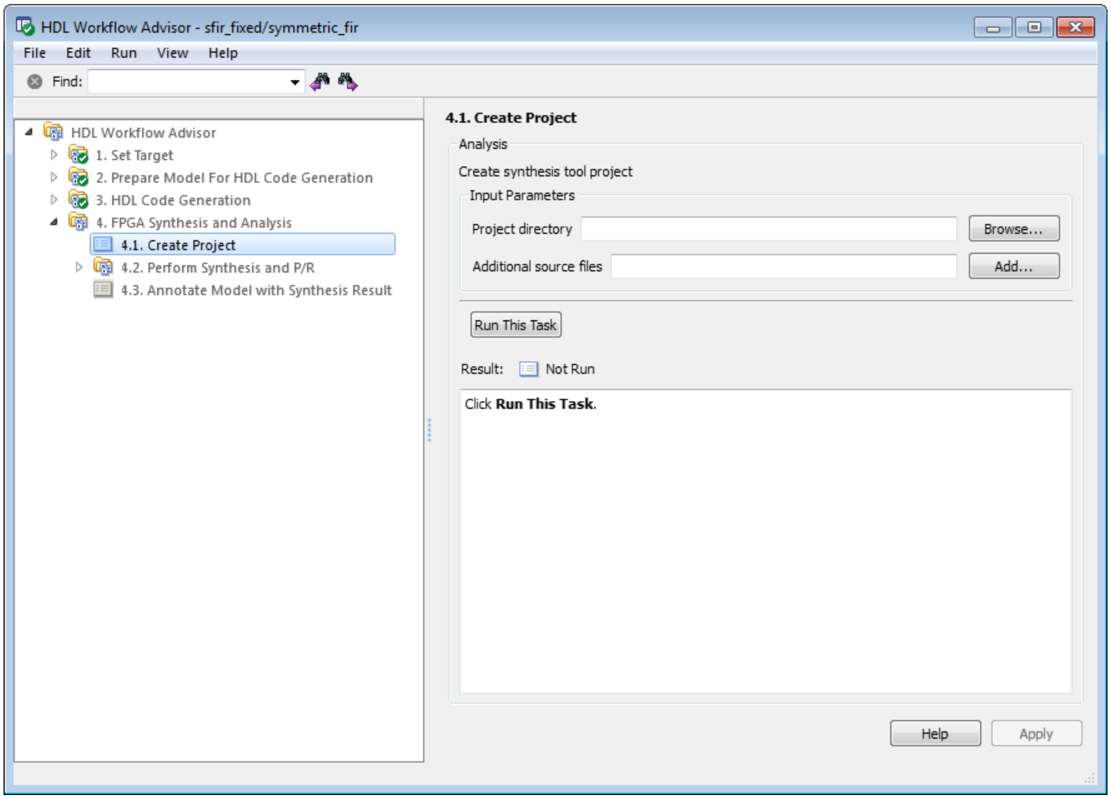
Note: A supported synthesis tool must be installed, and the synthesis tool executable must be on the system path to perform the tasks in the **FPGA Synthesis and Analysis** folder. See "Third-Party Synthesis Tools and Version Support" for more information.

Creating a Synthesis Project

The **Create Project** task does the following:

- Realizes a synthesis project for the tool from the previously generated HDL code
- Creates a link to the project files in the **Result** subpane
- (Optional) Launches the synthesis tool and opens the synthesis project

The following figure shows the **Create Project** task in an enabled state, after HDL code generation.


HDL Workflow Advisor - sfir_fixed/symmetric_fir
File
Edit
Run
View
Help
Find:
HDL Workflow Advisor
1. Set Target
2. Prepare Model For HDL Code Generation
3. HDL Code Generation
4. FPGA Synthesis and Analysis
4.1. Create Project
4.2. Perform Synthesis and P/R
4.3. Annotate Model with Synthesis Result
Browse...
Add...
Run This Task
Help
Apply

4.1. Create Project

Analysis

Create synthesis tool project

Input Parameters

Project directory

Additional source files

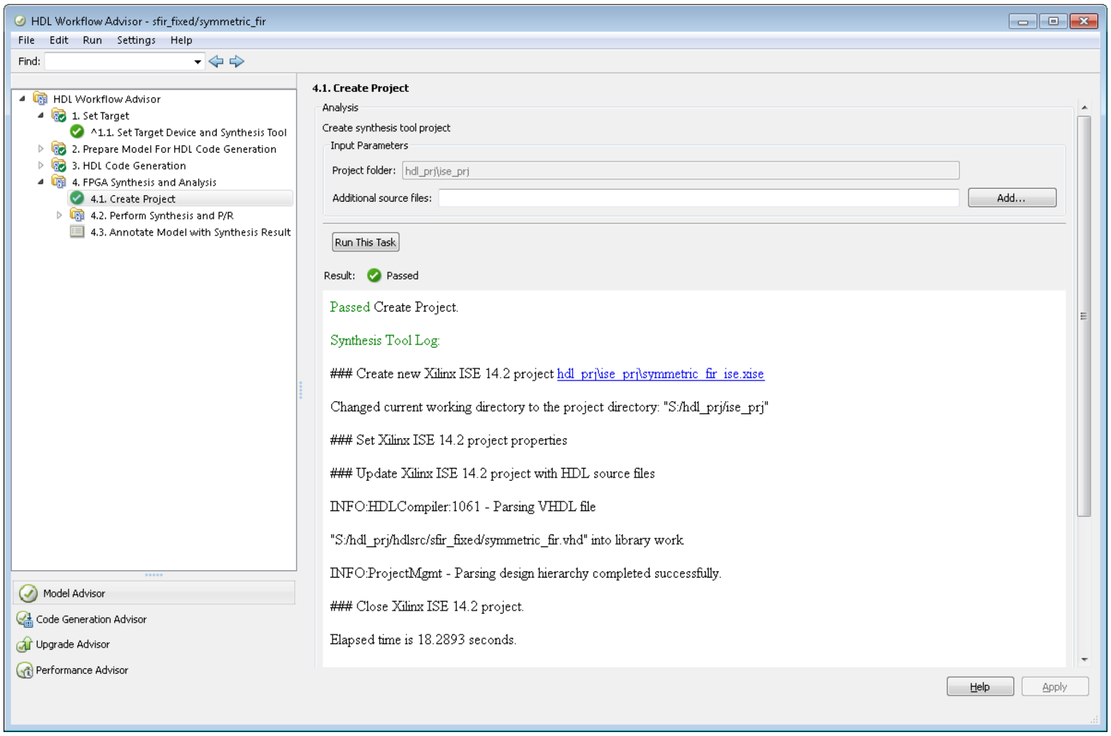
Result: Not Run

Click **Run This Task**.

The **Create Project** task parameters are:

- **Project directory**: The HDL Workflow Advisor writes the project files to a subfolder of the `hdlsrc` folder. You can enter the path to an alternative folder, or click the **Browse** button to navigate to the desired folder.
- **Additional source files**: To include HDL files (or other synthesis files, such as UCF or SDC files) that the code does not generate in your synthesis project, enter the full path to the desired files. Click the **Add** button to locate each file.

The following figure shows the HDL Workflow Advisor after passing the **Create Project** task. If you want to view the synthesis project, click the hyperlink in the **Result** subpane. This link launches the synthesis tool and opens the synthesis project.


HDL Workflow Advisor - sfir_fixed/symmetric_fir
File Edit Run Settings Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis Tool
2. Prepare Model For HDL Code Generation
3. HDL Code Generation
4. FPGA Synthesis and Analysis
4.1. Create Project
4.2. Perform Synthesis and P/R
4.3. Annotate Model with Synthesis Result
Model Advisor
Code Generation Advisor
Upgrade Advisor
Performance Advisor
4.1. Create Project
Analysis
Create synthesis tool project
Input Parameters
Project folder: hdl_prj\ise_prj
Additional source files:
Add...
Run This Task
Result: Passed
Help
Apply

```
Passed Create Project.

Synthesis Tool Log:

### Create new Xilinx ISE 14.2 project hdl_prj\ise_prj\symmetric_fir_ise.xise

Changed current working directory to the project directory: "S:/hdl_prj/ise_prj"

### Set Xilinx ISE 14.2 project properties

### Update Xilinx ISE 14.2 project with HDL source files

INFO:HDLCompiler:1061 - Parsing VHDL file

"S:/hdl_prj/hdlsrc/sfir_fixed/symmetric_fir.vhd" into library work

INFO:ProjectMgmt - Parsing design hierarchy completed successfully.

### Close Xilinx ISE 14.2 project.

Elapsed time is 18.2893 seconds.
```

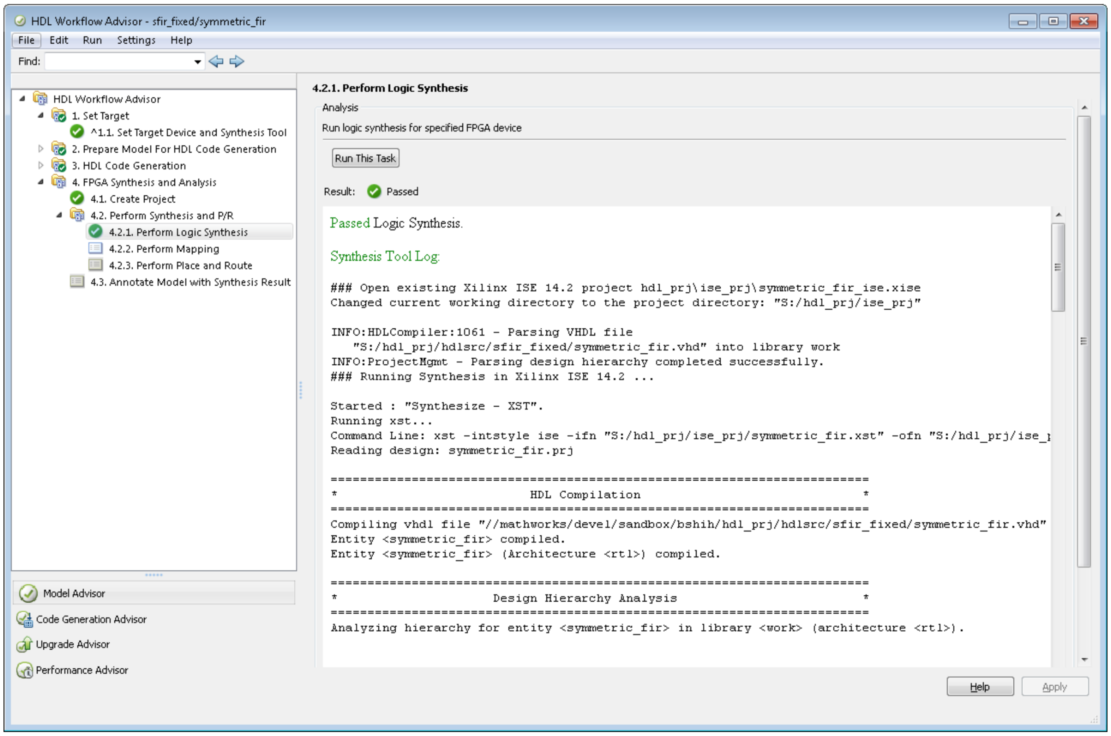
Performing Synthesis, Mapping, and Place and Route

Performing Logic Synthesis

The **Perform Logic Synthesis** task does the following:

- Launches the synthesis tool in the background.
- Opens the previously generated synthesis project, compiles HDL code, synthesizes the design and emits netlists and related files.
- Displays a synthesis log in the **Result** subpane.

The **Perform Logic Synthesis** task does not have input parameters. The following figure shows the HDL Workflow Advisor after passing the **Perform Logic Synthesis** task.


HDL Workflow Advisor - sfir_fixed/symmetric_fir
File Edit Run Settings Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis Tool
2. Prepare Model For HDL Code Generation
3. HDL Code Generation
4. FPGA Synthesis and Analysis
4.1. Create Project
4.2. Perform Synthesis and P/R
4.2.1. Perform Logic Synthesis
4.2.2. Perform Mapping
4.2.3. Perform Place and Route
4.3. Annotate Model with Synthesis Result
Model Advisor
Code Generation Advisor
Upgrade Advisor
Performance Advisor
4.2.1. Perform Logic Synthesis
Analysis
Run This Task
Help
Apply

Run logic synthesis for specified FPGA device

Result: Passed

Passed Logic Synthesis.

Synthesis Tool Log:

```
### Open existing Xilinx ISE 14.2 project hdl_prj\ise_prj\symmetric_fir_ise.xise
Changed current working directory to the project directory: "S:/hdl_prj/ise_prj"

INFO:HDLCompiler:1061 - Parsing VHDL file
   "S:/hdl_prj/hdlsrc/sfir_fixed/symmetric_fir.vhd" into library work
INFO:ProjectMgmt - Parsing design hierarchy completed successfully.
### Running Synthesis in Xilinx ISE 14.2 ...

Started : "Synthesize - XST".
Running xst...
Command Line: xst -intstyle ise -ifn "S:/hdl_prj/ise_prj/symmetric_fir.xst" -ofn "S:/hdl_prj/ise_
Reading design: symmetric_fir.prj

=========================================================================
*                          HDL Compilation                              *
=========================================================================
Compiling vhdl file "//mathworks/devel/sandbox/bshih/hdl_prj/hdlsrc/sfir_fixed/symmetric_fir.vhd"
Entity <symmetric_fir> compiled.
Entity <symmetric_fir> (Architecture <rtl>) compiled.

=========================================================================
*                     Design Hierarchy Analysis                         *
=========================================================================
Analyzing hierarchy for entity <symmetric_fir> in library <work> (architecture <rtl>).
```

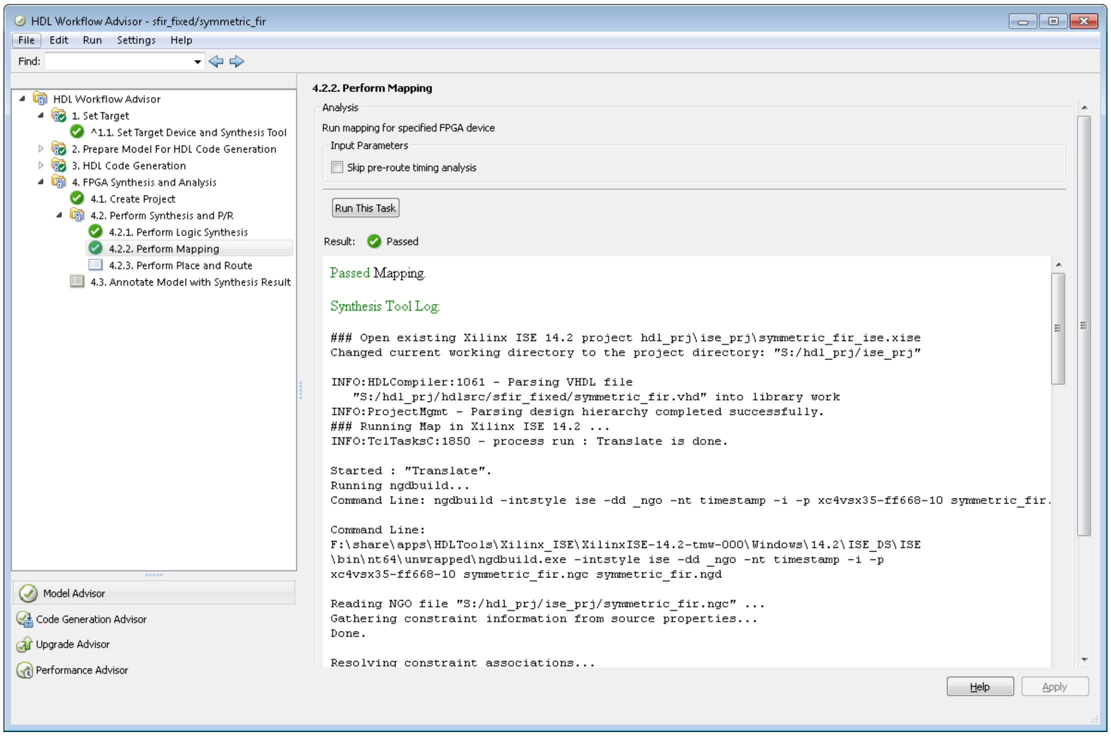
Performing Mapping

The **Perform Mapping** task does the following:

- Launches the synthesis tool in the background.
- Runs a mapping process that maps the synthesized logic design to the target FPGA.
- Emits a circuit description file for use in the place and route phase.
- Displays a log in the **Result** subpane.

If your tool does not support early timing estimation, you can enable **Skip pre-route timing analysis**. When this option is enabled, the **Annotate Model with Synthesis Result** task sets **Critical path source** to **post-route**.

The following figure shows the HDL Workflow Advisor after passing the **Perform Mapping** task.


HDL Workflow Advisor - sfir_fixed/symmetric_fir
File
Edit
Run
Settings
Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis Tool
2. Prepare Model For HDL Code Generation
3. HDL Code Generation
4. FPGA Synthesis and Analysis
4.1. Create Project
4.2. Perform Synthesis and P/R
4.2.1. Perform Logic Synthesis
4.2.2. Perform Mapping
4.2.3. Perform Place and Route
4.3. Annotate Model with Synthesis Result
Model Advisor
Code Generation Advisor
Upgrade Advisor
Performance Advisor
Run This Task
Help
Apply

4.2.2. Perform Mapping

Analysis

Run mapping for specified FPGA device

Input Parameters

☐ Skip pre-route timing analysis

Result: ✓ Passed

Passed Mapping.

Synthesis Tool Log:

```
### Open existing Xilinx ISE 14.2 project hdl_prj\ise_prj\symmetric_fir_ise.xise
Changed current working directory to the project directory: "S:/hdl_prj/ise_prj"

INFO:HDLCompiler:1061 - Parsing VHDL file
   "S:/hdl_prj/hdlsrc/sfir_fixed/symmetric_fir.vhd" into library work
INFO:ProjectMgmt - Parsing design hierarchy completed successfully.
### Running Map in Xilinx ISE 14.2 ...
INFO:TclTasksC:1850 - process run : Translate is done.

Started : "Translate".
Running ngdbuild...
Command Line: ngdbuild -intstyle ise -dd _ngo -nt timestamp -i -p xc4vsx35-ff668-10 symmetric_fir.

Command Line:
F:\share\apps\HDLTools\Xilinx_ISE\XilinxISE-14.2-tmw-000\Windows\14.2\ISE_DS\ISE
\bin\nt64\unwrapped\ngdbuild.exe -intstyle ise -dd _ngo -nt timestamp -i -p
xc4vsx35-ff668-10 symmetric_fir.ngc symmetric_fir.ngd

Reading NGO file "S:/hdl_prj/ise_prj/symmetric_fir.ngc" ...
Gathering constraint information from source properties...
Done.

Resolving constraint associations...
```

Performing Place and Route

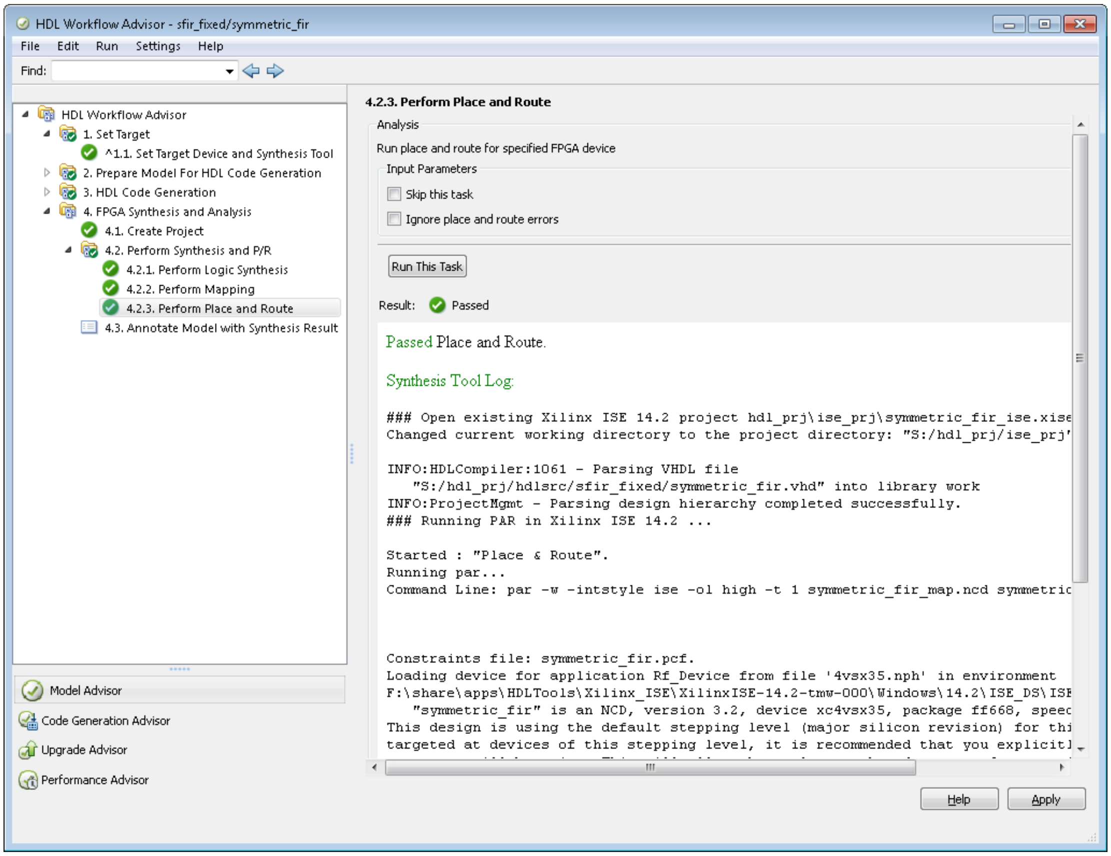
The **Perform Place and Route** task does the following:

- Launches the synthesis tool in the background.
- Runs a place and route process using the circuit description produced by the mapping process, and emits a circuit description suitable for programming an FPGA.
- Emits pre- and post-routing timing information for use in critical path analysis and back annotation of your source model.
- Displays a log in the **Result** subpane.

Unlike other tasks in the HDL Workflow Advisor hierarchy, **Perform Place and Route** is optional. If you select **Skip this task** in the right-hand pane, the HDL Workflow Advisor executes the workflow, but omits the **Perform Place and Route** task, marking it **Passed**. Select **Skip this task** if you prefer to do place and route work manually.

If the **Perform Place and Route** task fails, you can select **Ignore place and route errors** to continue to the **Annotate Model with Synthesis Result** task. This allows you to use post-mapping timing results to find critical paths in your model even if place and route fails.

The following figure shows the HDL Workflow Advisor after passing the **Perform Place and Route** task.


HDL Workflow Advisor - sfir_fixed/symmetric_fir
File Edit Run Settings Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis Tool
2. Prepare Model For HDL Code Generation
3. HDL Code Generation
4. FPGA Synthesis and Analysis
4.1. Create Project
4.2. Perform Synthesis and P/R
4.2.1. Perform Logic Synthesis
4.2.2. Perform Mapping
4.2.3. Perform Place and Route
4.3. Annotate Model with Synthesis Result
Model Advisor
Code Generation Advisor
Upgrade Advisor
Performance Advisor
Run This Task
Help
Apply

4.2.3. Perform Place and Route

Analysis

Run place and route for specified FPGA device

Input Parameters

☐ Skip this task

☐ Ignore place and route errors

Result: ✓ Passed

Passed Place and Route.

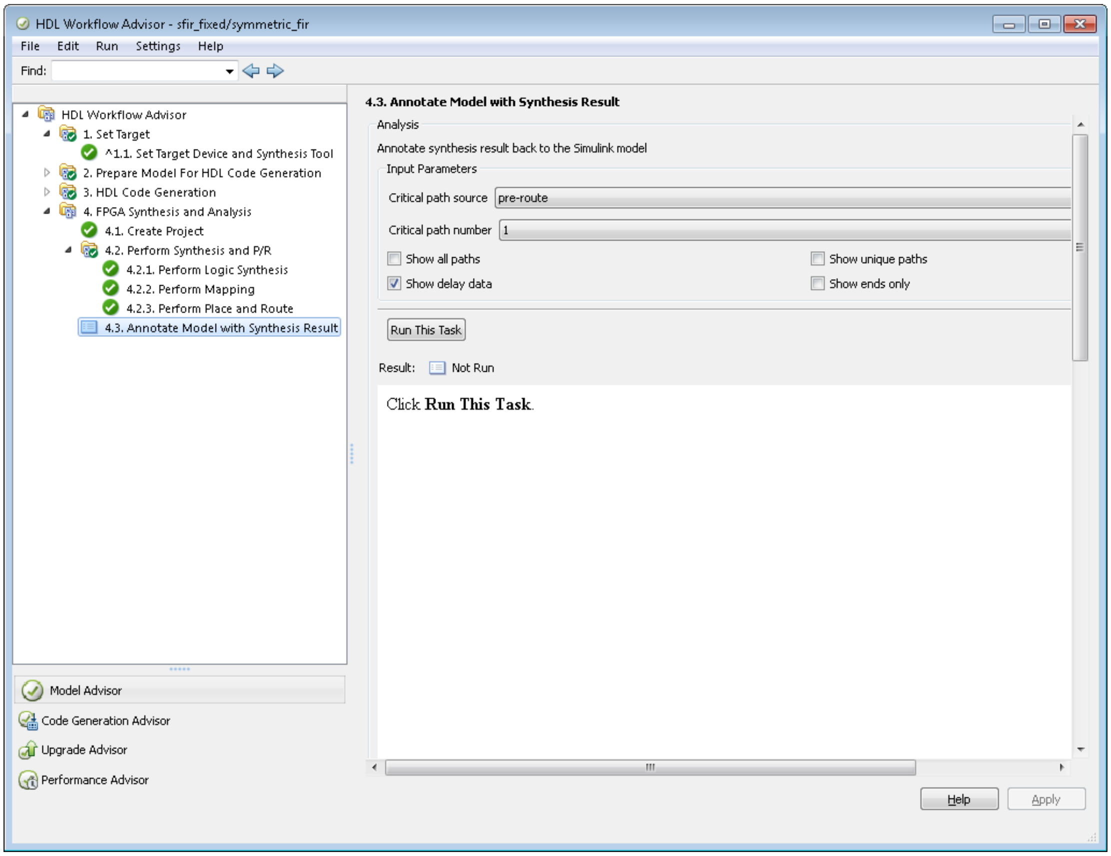
Synthesis Tool Log:

```
### Open existing Xilinx ISE 14.2 project hdl_prj\ise_prj\symmetric_fir_ise.xise
Changed current working directory to the project directory: "S:/hdl_prj/ise_prj'

INFO:HDLCompiler:1061 - Parsing VHDL file
   "S:/hdl_prj/hdlsrc/sfir_fixed/symmetric_fir.vhd" into library work
INFO:ProjectMgmt - Parsing design hierarchy completed successfully.
### Running PAR in Xilinx ISE 14.2 ...

Started : "Place & Route".
Running par...
Command Line: par -w -intstyle ise -ol high -t 1 symmetric_fir_map.ncd symmetric



Constraints file: symmetric_fir.pcf.
Loading device for application Rf_Device from file '4vsx35.nph' in environment
F:\share\apps\HDLTools\Xilinx_ISE\XilinxISE-14.2-tmw-000\Windows\14.2\ISE_DS\ISE
   "symmetric_fir" is an NCD, version 3.2, device xc4vsx35, package ff668, speec
This design is using the default stepping level (major silicon revision) for thi
targeted at devices of this stepping level, it is recommended that you explicitl
```

Annotating Your Model with Critical Path Information

The **Annotate Model with Synthesis Result** task helps you identify critical paths in your model. In this task, you can analyze pre- or post-routing timing information from the **Perform Place and Route** task and visually highlight one or more critical paths in your model.

The following figure shows the **Annotate Model with Synthesis Result** task in an enabled state.


HDL Workflow Advisor - sfir_fixed/symmetric_fir
File Edit Run Settings Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis Tool
2. Prepare Model For HDL Code Generation
3. HDL Code Generation
4. FPGA Synthesis and Analysis
4.1. Create Project
4.2. Perform Synthesis and P/R
4.2.1. Perform Logic Synthesis
4.2.2. Perform Mapping
4.2.3. Perform Place and Route
4.3. Annotate Model with Synthesis Result
Model Advisor
Code Generation Advisor
Upgrade Advisor
Performance Advisor
Run This Task
Help
Apply

4.3. Annotate Model with Synthesis Result

Analysis

Annotate synthesis result back to the Simulink model

Input Parameters

Critical path source: pre-route

Critical path number: 1

- [ ] Show all paths
- [ ] Show unique paths
- [x] Show delay data
- [ ] Show ends only

Result: Not Run

Click **Run This Task**.

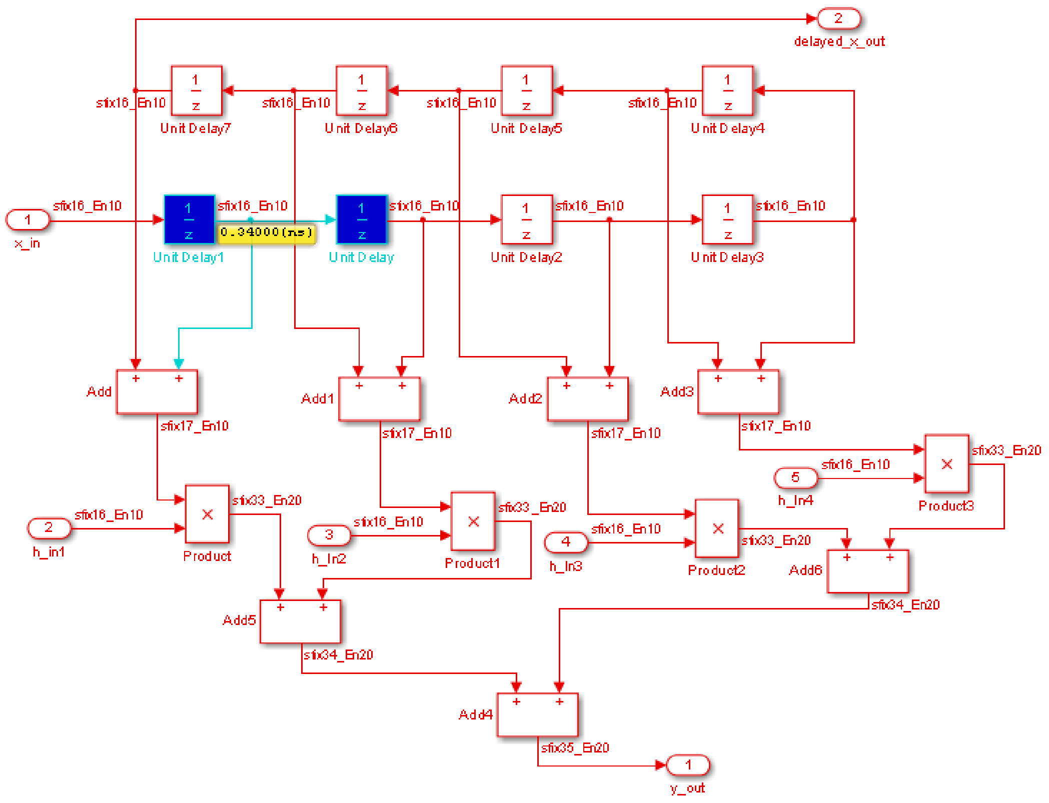
The task parameters are:

- **Critical path source**: Select **pre-route** or **post-route**. The default is **pre-route**.

  Note that the **pre-route** option is unavailable when **Skip pre-route timing analysis** is enabled in the **Perform Mapping** task.

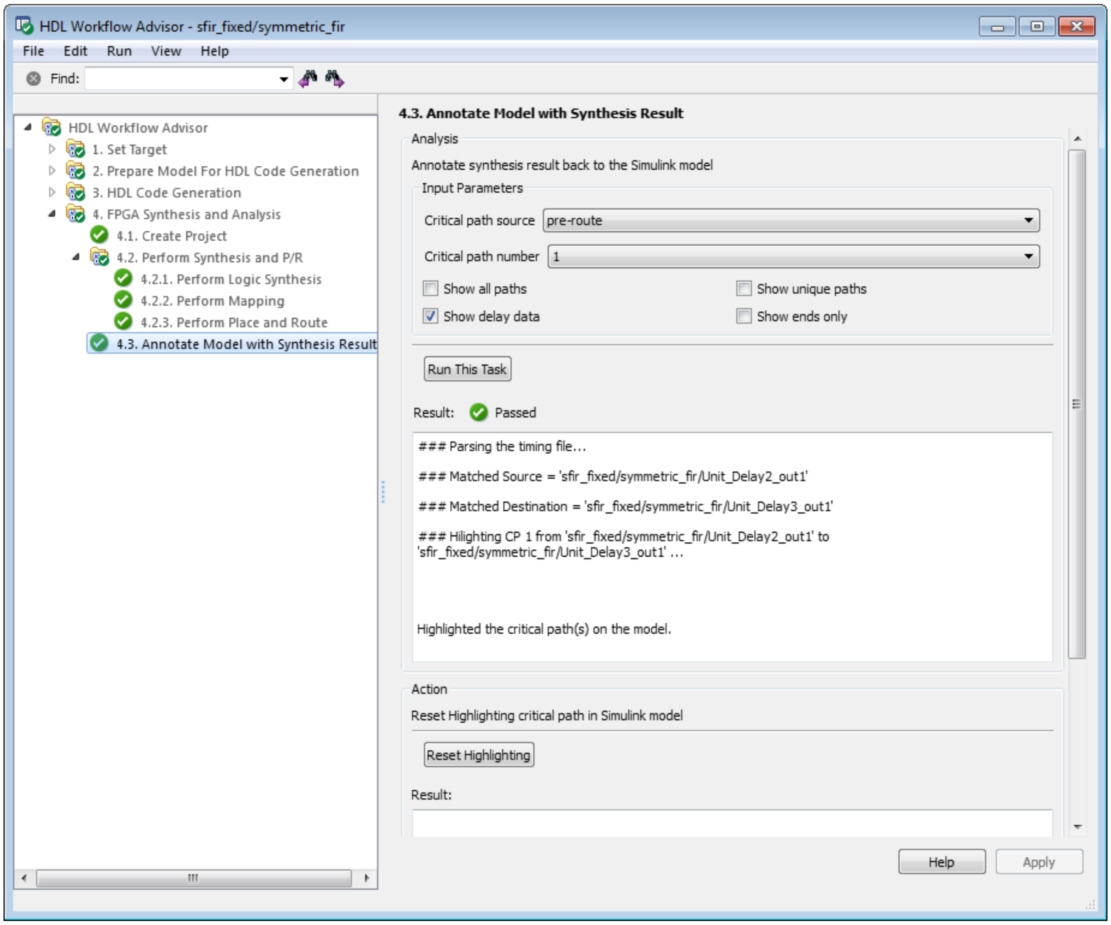
- **Critical path number**: You can annotate up to 3 critical paths. Select the number of paths you want to annotate. The default is `1`.
- **Show all paths**: Show critical paths, including duplicate paths. The default is `off`.
- **Show unique paths**: Show only the first instance of a path that is duplicated. The default is `off`.
- **Show delay data**: Annotate the cumulative timing delay on each path. The default is `on`.
- **Show ends only**: Show the endpoints of each path, but omit the connecting signal lines. The default is `off`.

When the **Annotate Model with Synthesis Result** task runs to completion, HDL Coder displays the DUT with critical path information highlighted. The following figure shows a subsystem after critical path annotation. Using default options, the annotation includes the endpoints, signal lines, and delay data.



After the **Annotate Model with Synthesis Result** task runs to completion, the HDL Workflow Advisor enables the **Reset Highlighting** button in the **Action** subpane. When you click this button, the HDL Workflow Advisor:

- Clears critical path annotations from the model.
- Resets the **Annotate Model with Synthesis Result** task.


HDL Workflow Advisor - sfir_fixed/symmetric_fir
File Edit Run View Help
Find:
HDL Workflow Advisor
1. Set Target
2. Prepare Model For HDL Code Generation
3. HDL Code Generation
4. FPGA Synthesis and Analysis
4.1. Create Project
4.2. Perform Synthesis and P/R
4.2.1. Perform Logic Synthesis
4.2.2. Perform Mapping
4.2.3. Perform Place and Route
4.3. Annotate Model with Synthesis Result
Run This Task
Reset Highlighting
Help
Apply

4.3. Annotate Model with Synthesis Result

Analysis

Annotate synthesis result back to the Simulink model

Input Parameters

Critical path source: pre-route

Critical path number: 1

Show all paths ☐ Show unique paths ☐

Show delay data ☑ Show ends only ☐

Result: Passed

```
### Parsing the timing file...

### Matched Source = 'sfir_fixed/symmetric_fir/Unit_Delay2_out1'

### Matched Destination = 'sfir_fixed/symmetric_fir/Unit_Delay3_out1'

### Hilighting CP 1 from 'sfir_fixed/symmetric_fir/Unit_Delay2_out1' to
'sfir_fixed/symmetric_fir/Unit_Delay3_out1' ...


Highlighted the critical path(s) on the model.
```

Action

Reset Highlighting critical path in Simulink model

Result:

Related Examples

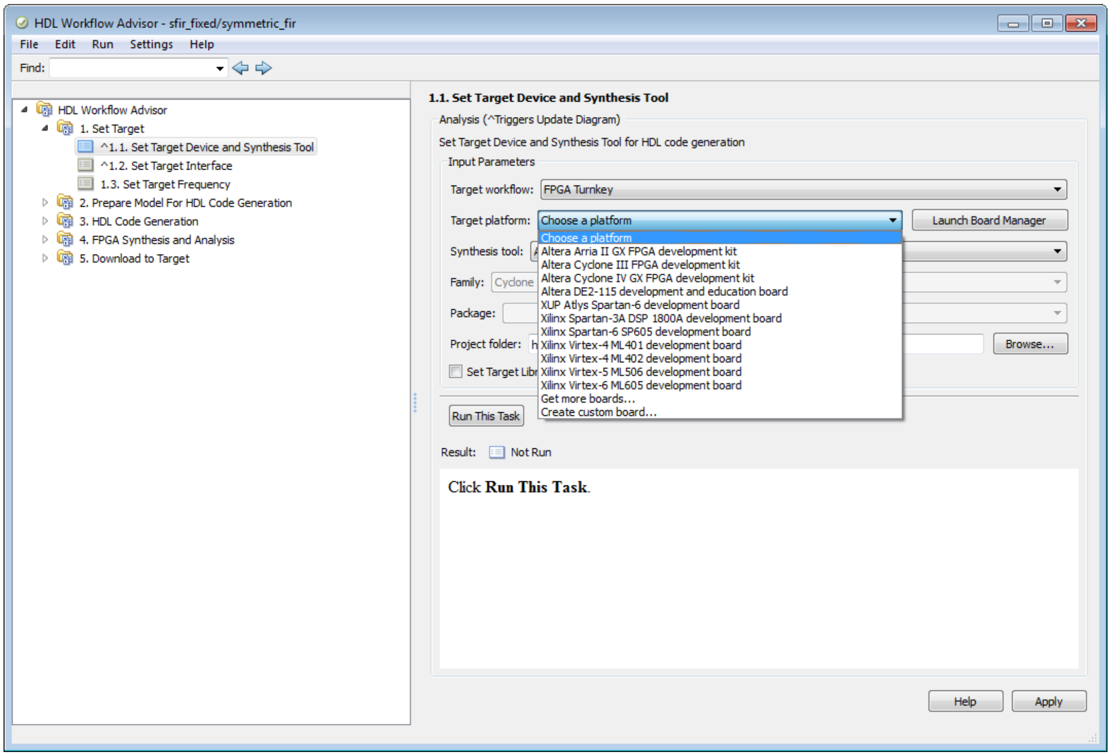
- “Open and Run Tasks in the HDL Workflow Advisor” on page 22-2
- “Save and Restore HDL Workflow Advisor State” on page 22-5
- “Fix a Workflow Advisor Warning or Failure” on page 22-9
- “View and Save HDL Workflow Advisor Reports” on page 22-12
- “HDL Workflow Advisor Tasks” on page 25-2

Automated Workflows for Specific Targets and Tools

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 and other information obtained during synthesis
- Complete automated workflows for selected FPGA development target devices and Simulink Real-Time, including FPGA-in-the-loop simulation

The HDL Workflow Advisor helps you perform complete automated workflows for a number of target devices. For the **Target workflow** you select, the **Target platform** menu lists the supported target devices.


HDL Workflow Advisor - sfir_fixed/symmetric_fir
File
Edit
Run
Settings
Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis Tool
^1.2. Set Target Interface
1.3. Set Target Frequency
2. Prepare Model For HDL Code Generation
3. HDL Code Generation
4. FPGA Synthesis and Analysis
5. Download to Target
Choose a platform
Altera Arria II GX FPGA development kit
Altera Cyclone III FPGA development kit
Altera Cyclone IV GX FPGA development kit
Altera DE2-115 development and education board
XUP Atlys Spartan-6 development board
Xilinx Spartan-3A DSP 1800A development board
Xilinx Spartan-6 SP605 development board
Xilinx Virtex-4 ML401 development board
Xilinx Virtex-4 ML402 development board
Xilinx Virtex-5 ML506 development board
Xilinx Virtex-6 ML605 development board
Get more boards...
Create custom board...
Launch Board Manager
Browse...
Run This Task
Help
Apply

1.1. Set Target Device and Synthesis Tool

Analysis (^Triggers Update Diagram)

Set Target Device and Synthesis Tool for HDL code generation

Input Parameters

Target workflow: FPGA Turnkey

Target platform: Choose a platform

Synthesis tool:

Family: Cyclone

Package:

Project folder: h

Set Target Libr

Result: Not Run

Click **Run This Task**.

After you select the desired target device and configure its I/O interface, you can let the HDL Workflow Advisor perform the subsequent model checking, HDL code generation, and FPGA synthesis and analysis tasks, without your intervention. For information on automated workflows for specific types of targets, see:

- “FPGA Programming and Configuration” on page 27-33
- “Program Standalone Xilinx FPGA Development Board from Simulink” (HDL Coder Support Package for Xilinx FPGA Boards)
- “Program Standalone Altera FPGA Development Board from Simulink” (HDL Coder Support Package for Altera FPGA Boards)
- “Custom IP Core Generation” on page 26-2
- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 27-2

Related Examples

- “Open and Run Tasks in the HDL Workflow Advisor” on page 22-2
- “Save and Restore HDL Workflow Advisor State” on page 22-5
- “Fix a Workflow Advisor Warning or Failure” on page 22-9
- “View and Save HDL Workflow Advisor Reports” on page 22-12
- “HDL Workflow Advisor Tasks” on page 25-2

Synthesis Objective to Tcl Command Mapping

In this section...
"Altera Quartus II" on page 22-63
"Xilinx Vivado 2014.4" on page 22-63
"Xilinx ISE 14.7 with PlanAhead" on page 22-64

When you specify a synthesis objective in the HDL Workflow Advisor **Synthesis objective** field, or in the HDL Workflow CLI workflow `hdlcoder.Objective`, the HDL Coder software generates Tcl commands that are specific to your synthesis tool.

Altera Quartus II

Synthesis objective	Tcl Commands
Area Optimized	`set_global_assignment -name OPTIMIZATION_TECHNIQUE "Area"` `set_global_assignment -name FITTER_EFFORT "Standard Fit"`
Compile Optimized	`set_global_assignment -name OPTIMIZATION_TECHNIQUE "Balanced"` `set_global_assignment -name FITTER_EFFORT "Fast Fit"`
Speed Optimized	`set_global_assignment -name OPTIMIZATION_TECHNIQUE "Speed"` `set_global_assignment -name FITTER_EFFORT "Standard Fit"`

Xilinx Vivado 2014.4

If your tool version is different, the Tcl commands are slightly different.

Synthesis objective	Tcl Commands
Area Optimized	`set_property strategy {Vivado Synthesis Defaults} [get_runs synth_1]` `set_property strategy "Area_Explore" [get_runs impl_1]`

Synthesis objective	Tcl Commands
Compile Optimized	`set_property strategy "Flow_RuntimeOptimized" [get_runs synth1]` `set_property strategy "Flow_Quick" [get_runs impl_1]`
Speed Optimized	`set_property strategy {Vivado Synthesis Defaults} [get_runs synth_1]` `set_property strategy "Performance_Explore" [get_runs impl_1]`

Xilinx ISE 14.7 with PlanAhead

If your tool version is different, the Tcl commands are slightly different.

Synthesis objective	Tcl Commands
Area Optimized	`set_property strategy "AreaReduction" [get_runs synth_1]` `set_property strategy "MapCoverArea" [get_runs impl_1]`
Compile Optimized	`set_property strategy "{XST Defaults}" [get_runs synth_1]` `set_property strategy "{ISE Defaults}" [get_runs impl_1]`
Speed Optimized	`set_property strategy "TimingWithIOBPacking" [get_runs synth_1]` `set_property strategy "MapTiming" [get_runs impl_1]`

Run HDL Workflow with a Script

In this section...

To run the HDL workflow as a command-line script, configure and run the HDL Workflow Advisor with your Simulink design, then export a script. The script uses HDL Workflow CLI commands to perform the same tasks as the HDL Workflow Advisor, including FPGA bitstream or synthesis project generation.

You can export an HDL workflow script for these target workflows:

- `Generic ASIC/FPGA`
- `FPGA Turnkey`
- `IP Core Generation`

To update an existing script, import it into the HDL Workflow Advisor, modify the tasks, and export the updated script. Alternatively, you can manually edit the script.

Export an HDL Workflow Script

1. In the HDL Workflow Advisor, configure and run all the tasks.
2. Select **File** > **Export to Script**.
3. 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.

Enable or Disable Tasks in HDL Workflow Script

To disable all workflow tasks, update the workflow configuration object with the `hdlcoder.WorkflowConfig.clearAllTasks` method.

To reenable all workflow tasks, update the workflow configuration object with the `hdlcoder.WorkflowConfig.setAllTasks` method.

Run a Single Workflow Task

To run a single workflow task without rerunning other workflow tasks:

1. Disable all tasks in the workflow configuration object by running the `hdlcoder.WorkflowConfig.clearAllTasks` method.
2. In the workflow configuration object, enable the task that you want to run.

For example, if you previously ran an HDL workflow script and generated a bitstream, you can program your target hardware without rerunning the other workflow tasks. To run the target device programming task for an `hdlcoder.WorkflowConfig` workflow configuration object, `hCfg` and Simulink design, `model/DUTname`:

1. Run the `clearAllTasks` method.

   ```
   hCfg.clearAllTasks;
   ```
2. Enable the target device programming task.

   ```
   hCfg.RunTaskProgramTargetDevice = true;
   ```
3. Run the workflow.

   ```
   hdlcoder.runWorkflow('model/DUTname', hCfg);
   ```

Import an HDL Workflow Script

1. In the HDL Workflow Advisor, select **File** > **Import from Script**.
2. In the Import Workflow Configuration dialog box, select the script file and click **Open**.

   The HDL Workflow Advisor updates the tasks with the imported script settings.

Generic ASIC/FPGA Workflow Script Example

This example shows how to configure and run an exported HDL workflow script.

This script is a generic ASIC/FPGA workflow script that targets a Xilinx Virtex 7 device. It uses the Xilinx Vivado synthesis tool.

Open and view your exported HDL workflow script.

```
% Export Workflow Configuration Script
% Generated with MATLAB 8.6 (R2015b) at 11:33:25 on 08/07/2015
% Parameter Values:
%     Filename  : 'S:\generic_workflow_example.m'
%     Overwrite : true
%     Comments  : true
%     Headers   : true
%     DUT       : 'sfir_fixed/symmetric_fir1'

%% Load the Model
load_system('sfir_fixed');

%% Model HDL Parameters
% Set Model HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir1');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Virtex7');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7vx485t');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'ffg1761');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-2');
hdlset_param('sfir_fixed', 'TargetDirectory', 'hdl_prj\hdlsrc');


%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado','TargetWorkflow', ...
      'Generic ASIC/FPGA');

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndTestbench = true;
hWC.RunTaskVerifyWithHDLCosimulation = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskRunSynthesis = true;
hWC.RunTaskRunImplementation = false;
hWC.RunTaskAnnotateModelWithSynthesisResult = true;

% Set Properties related to Generate RTL Code And Testbench Task
hWC.GenerateRTLCode = true;
hWC.GenerateRTLTestbench = false;
hWC.GenerateCosimulationModel = false;
hWC.CosimulationModelForUseWith = 'Mentor Graphics ModelSim';
hWC.GenerateValidationModel = false;

% Set Properties related to Create Project Task
hWC.Objective = hdlcoder.Objective.None;
hWC.AdditionalProjectCreationTclFiles = '';

% Set Properties related to Run Synthesis Task
hWC.SkipPreRouteTimingAnalysis = false;

% Set Properties related to Run Implementation Task
hWC.IgnorePlaceAndRouteErrors = false;

% Set Properties related to Annotate Model With Synthesis Result Task
hWC.CriticalPathSource = 'pre-route';
hWC.CriticalPathNumber = 1;
hWC.ShowAllPaths = false;
hWC.ShowDelayData = true;
hWC.ShowUniquePaths = false;
hWC.ShowEndsOnly = false;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('sfir_fixed/symmetric_fir1', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `generic_workflow_example.m`, at the command line, enter:

`generic_workflow_example.m`

FPGA Turnkey Workflow Script Example

This example shows how to configure and run an exported HDL workflow script.

This script is an FPGA Turnkey workflow script that targets a Xilinx Virtex 5 development board. It uses the Xilinx ISE synthesis tool.

Open and view your exported HDL workflow script.

```
% Export Workflow Configuration Script
% Generated with MATLAB 8.6 (R2015b) at 14:24:32 on 08/07/2015
% Parameter Values:
%     Filename  : 'S:\turnkey_workflow_example.m'
%     Overwrite : true
%     Comments  : true
%     Headers   : true
%     DUT       : 'hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA'

%% Load the Model
load_system('hdlcoderUARTServoControllerExample');

%% Model HDL Parameters
% Set Model HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'HDLSubsystem', 'hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisTool', 'Xilinx ISE');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisToolChipFamily', 'Virtex5');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisToolDeviceName', 'xc5vsx50t');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisToolPackageName', 'ff1136');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisToolSpeedValue', '-1');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'TargetPlatform', 'Xilinx Virtex-5 ML506 development board');
hdlset_param('hdlcoderUARTServoControllerExample', 'Workflow', 'FPGA Turnkey');

% Set Inport HDL parameters
```

```
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/uart_rxd', ...
    'IOInterface', 'RS-232 Serial Port Rx');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/uart_rxd', ...
    'IOInterfaceMapping', '[0]');

% Set Outport HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/uart_txd', ...
    'IOInterface', 'RS-232 Serial Port Tx');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/uart_txd', ...
    'IOInterfaceMapping', '[0]');

% Set Outport HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/version', ...
    'IOInterface', 'LEDs General Purpose [0:7]');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/version', ...
    'IOInterfaceMapping', '[0:3]');

% Set Outport HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/pwm_output', ...
    'IOInterface', 'Expansion Headers J6 Pin 2-64 [0:31]');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/pwm_output', ...
    'IOInterfaceMapping', '[0]');

% Set Outport HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/servo_debug1', ...
    'IOInterface', 'Expansion Headers J6 Pin 2-64 [0:31]');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/servo_debug1', ...
    'IOInterfaceMapping', '[1]');

% Set Outport HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/servo_debug2', ...
    'IOInterface', 'Expansion Headers J6 Pin 2-64 [0:31]');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/servo_debug2', ...
    'IOInterfaceMapping', '[2]');


%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx ISE', ...
    'TargetWorkflow','FPGA Turnkey');

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';
```

```
% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndTestbench = true;
hWC.RunTaskVerifyWithHDLCosimulation = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskPerformLogicSynthesis = true;
hWC.RunTaskPerformMapping = true;
hWC.RunTaskPerformPlaceAndRoute = true;
hWC.RunTaskGenerateProgrammingFile = true;
hWC.RunTaskProgramTargetDevice = false;

% Set Properties related to Create Project Task
hWC.Objective = hdlcoder.Objective.None;
hWC.AdditionalProjectCreationTclFiles = '';

% Set Properties related to Perform Mapping Task
hWC.SkipPreRouteTimingAnalysis = true;

% Set Properties related to Perform Place and Route Task
hWC.IgnorePlaceAndRouteErrors = false;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `turnkey_workflow_example.m`, at the command line, enter:

```
turnkey_workflow_example.m
```

IP Core Generation Workflow Script Example

This example shows how to configure and run an exported HDL workflow script.

This script is an IP core generation workflow script that targets the Altera Cyclone V SoC development kit. It uses the Altera Quartus II synthesis tool.

Open and view your exported HDL workflow script.

```
% Export Workflow Configuration Script
% Generated with MATLAB 8.6 (R2015b) at 14:42:16 on 08/07/2015
% Parameter Values:
%     Filename  : 'S:\ip_core_gen_workflow_example.m'
%     Overwrite : true
%     Comments  : true
%     Headers   : true
%     DUT       : 'hdlcoder_led_blinking/led_counter'

%% Load the Model
load_system('hdlcoder_led_blinking');

%% Model HDL Parameters
% Set Model HDL parameters
hdlset_param('hdlcoder_led_blinking', ...
    'HDLSubsystem', 'hdlcoder_led_blinking/led_counter');
hdlset_param('hdlcoder_led_blinking', 'OptimizationReport', 'on');
hdlset_param('hdlcoder_led_blinking', ...
    'ReferenceDesign', 'Default system (Qsys 14.0)');
hdlset_param('hdlcoder_led_blinking', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_led_blinking', 'ResourceReport', 'on');
hdlset_param('hdlcoder_led_blinking', 'SynthesisTool', 'Altera QUARTUS II');
hdlset_param('hdlcoder_led_blinking', 'SynthesisToolChipFamily', 'Cyclone V');
hdlset_param('hdlcoder_led_blinking', 'SynthesisToolDeviceName', '5CSXFC6D6F31C6');
hdlset_param('hdlcoder_led_blinking', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('hdlcoder_led_blinking', ...
    'TargetPlatform', 'Altera Cyclone V SoC development kit - Rev.D');
hdlset_param('hdlcoder_led_blinking', 'Traceability', 'on');
hdlset_param('hdlcoder_led_blinking', 'Workflow', 'IP Core Generation');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter', ...
    'ProcessorFPGASynchronization', 'Free running');

% Set Inport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_frequency', ...
    'IOInterface', 'AXI4');
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_frequency', ...
    'IOInterfaceMapping', 'x"100"');

% Set Inport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_direction', ...
    'IOInterface', 'AXI4');
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_direction', ...
    'IOInterfaceMapping', 'x"104"');

% Set Outport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/LED', 'IOInterface', 'External Port');

% Set Outport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/Read_back', 'IOInterface', 'AXI4');
hdlset_param('hdlcoder_led_blinking/led_counter/Read_back', ...
    'IOInterfaceMapping', 'x"108"');


%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Altera QUARTUS II', ...
    'TargetWorkflow','IP Core Generation');

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndIPCore = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskGenerateSoftwareInterfaceModel = false;
hWC.RunTaskBuildFPGABitstream = true;
hWC.RunTaskProgramTargetDevice = false;

% Set Properties related to Generate RTL Code And IP Core Task
hWC.IPCoreRepository = '';
hWC.GenerateIPCoreReport = true;

% Set Properties related to Create Project Task
hWC.Objective = hdlcoder.Objective.AreaOptimized;

% Set Properties related to Generate Software Interface Model Task
hWC.OperatingSystem = '';
hWC.AddLinuxDeviceDriver = false;

% Set Properties related to Build FPGA Bitstream Task
hWC.RunExternalBuild = true;
hWC.TclFileForSynthesisBuild = hdlcoder.BuildOption.Default;

% Validate the Workflow Configuration Object
```

```
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('hdlcoder_led_blinking/led_counter', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `ip_core_workflow_example.m`, at the command line, enter:

```
ip_core_gen_workflow_example.m
```

See Also

Functions
`hdlcoder.runWorkflow` | `hdlcoder.WorkflowConfig.clearAllTasks` | `hdlcoder.WorkflowConfig.setAllTasks`

Classes
hdlcoder.WorkflowConfig

HDL Test Bench

Test Bench Generation

You can generate a HDL Testbench for a subsystem or model reference that you specify in your Simulink model. The coder generates an HDL test bench by running a Simulink simulation to capture input vectors and expected output data for your DUT.

In this section...

How Test Bench Generation Works

HDL Coder writes the DUT stimulus and reference data from your MATLAB or Simulink simulation to data files (`.dat`).

During HDL simulation, the HDL test bench reads the saved stimulus from the `.dat` files. The test bench compares the actual DUT output with the expected output, which is also saved in .dat files. After you generate code, the message window displays links to the test bench data files.

Reference data is delayed by one clock cycle in the waveform viewer compared to default test bench generation due to the delay in reading data from files.

Test Bench Data Files

The coder saves stimulus and reference data for each DUT input and output in a separate test bench data file (`.dat`), with the following exceptions:

- Two files are generated for the real and imaginary parts of complex data.
- Constant DUT input data is written to the test bench as constants.

Vector input or output data is saved as a single file.

Test Bench Data Type Limitations

If you have double, single, or enumeration data types at the DUT inputs and outputs, the simulation data is generated as constants in the test bench code, instead of writing the simulation data to files.

Use Constants Instead of File I/O

You can generate test bench stimulus and reference data as constants in the test bench code instead of using file I/O. Simulating a long running test bench that uses constants requires more memory than a test bench that uses file I/O.

If your DUT inputs or outputs use data types that are not supported for file I/O, test bench generation automatically generates data as constants. For details, see “Test Bench Data Type Limitations” on page 23-3.

Using the HDL Workflow Advisor

To generate a test bench that uses constants:

1. In the **HDL Code Generation** > **Set Code Generation Options** > **Set Testbench Options** task, clear **Use file I/O to read/write test bench data** and click **Apply**.
2. In the **HDL Code Generation** > **Generate RTL Code and Testbench** task, select **Generate RTL testbench** and click **Apply**.

Using the Command Line

To generate a test bench that uses constants, use the `UseFileIOInTestBench` parameter with `makehdltb`.

For example, to generate a Verilog test bench by using constants for a DUT subsystem, `sfir_fixed/symmetric_fir`, enter:

```
makehdltb('sfir_fixed/symmetric_fir','TargetLanguage','Verilog',...
          'UseFileIOInTestBench','off');
```

See Also

makehdltb

More About

- “Test Bench Block Restrictions” on page 23-5

- “Choose a Test Bench for Generated HDL Code” on page 18-37

Test Bench Block Restrictions

Blocks that belong to the blocksets and toolboxes in the following list should not be directly connected to the DUT. Instead, place them in a subsystem, and connect the subsystem to the DUT. This restriction applies to all blocks in the following products:

- RF Blockset™
- Simscape™ Driveline™
- SimEvents®
- Simscape Multibody™
- Simscape Power Systems™
- Simscape

FPGA Board Customization

FPGA Board Customization

In this section...
"Feature Description" on page 24-2
"Custom Board Management" on page 24-2
"FPGA Board Requirements" on page 24-3

Feature Description

Both HDL Coder and HDL Verifier software include a set of predefined FPGA boards you can use with the Turnkey or FPGA-in-the-loop (FIL) workflows. You can view the lists of these supported boards in the HDL Workflow Advisor or in the FIL wizard. With the FPGA Board Manager, you can add additional boards to use either of these workflows. To add a board, you need the relevant information from the board specification documentation.

The FPGA Board Manager is the hub for accessing wizards and dialog boxes that take you through the steps necessary to create a custom board configuration. You can also access options for:

- Importing a custom board
- Copying a board definition file for further modification
- Verifying a new board

Custom Board Management

You manage FPGA custom boards through the following user interfaces:

- "FPGA Board Manager" on page 24-21: portal to adding, importing, deleting, and otherwise managing board definition files.
- "New FPGA Board Wizard" on page 24-25: This wizard guides you through creating a custom board definition file with information you obtain from the board specification documentation.
- "FPGA Board Editor" on page 24-38: user interface for viewing or editing board information.

To begin, review the “FPGA Board Requirements” on page 24-3 and then follow the steps described in “Create Custom FPGA Board Definition” on page 24-7.

FPGA Board Requirements

- “FPGA Device” on page 24-3
- “FPGA Design Software” on page 24-3
- “General Hardware Requirements” on page 24-3
- “Ethernet Connection Requirements for FPGA-in-the-Loop” on page 24-4
- “JTAG Connection Requirements for FPGA-in-the-Loop” on page 24-6

FPGA Device

Select one of the following links to view a current list of supported FPGA device families:

- For use with FPGA-in-the-loop (FIL), see “Supported FPGA Device Families for Board Customization” (HDL Verifier).
- For use with FPGA Turnkey, see “Supported FPGA Device Families for Board Customization”.

FPGA Design Software

Altera Quartus® II or Xilinx ISE is required. See product documentation for HDL Coder or HDL Verifier for the specific software versions required.

The following MathWorks® tools are required to use FIL or FPGA Turnkey.

Workflow	Required Tools
FPGA-in-the-loop	• HDL Verifier • Fixed-Point Designer
FPGA Turnkey	• HDL Coder • Simulink • Fixed-Point Designer

General Hardware Requirements

To use an FPGA development board, make sure that you have the following FPGA resources:

- **Clock**: An external clock connected to the FPGA is required. The clock can be differential or single-ended. The accepted clock frequency is from 5 MHz to 300 MHz. When used with FIL, there are additional requirements to the clock frequency (see "Ethernet Connection Requirements for FPGA-in-the-Loop" on page 24-4).
- **Reset**: An external reset signal connected to the FPGA is optional. When supplied, this signal functions as the global reset to the FPGA design.
- **JTAG download cable**: A JTAG download cable that connects host computer and FPGA board is required for the FPGA programming. The FPGA must be programmable using Xilinx iMPACT or Altera Quartus II.

Ethernet Connection Requirements for FPGA-in-the-Loop

- "Supported Ethernet PHY Device" on page 24-4
- "Ethernet PHY Interface" on page 24-5
- "Special Timing Considerations for RGMII" on page 24-5
- "Special Clock Frequency Requirement for GMII/RGMII/SGMII Interface" on page 24-5

Supported Ethernet PHY Device

On the FPGA board, the Ethernet MAC is implemented in FPGA. An Ethernet PHY chip is required to be on the FPGA board to connect the physical medium to the Media ACcess (MAC) layer in the FPGA.

Note: When programming the FPGA, HDL Verifier assumes that there is only one download cable connected to the Host computer. It also assumes that the FPGA programming software automatically recognizes the cable. If not, use FPGA programming software to program your FPGA with the correct options.

The FIL feature is tested with the following Ethernet PHY chips and may not work with other Ethernet PHY devices.

Ethernet PHY Chip	Test
Marvell® Alaska 88E1111	For GMII, RGMII, SGMII, and 100 Base-T MII interfaces
National Semiconductor DP83848C	For 100 Base-T MII interface only

Ethernet PHY Interface

The Ethernet PHY chip must be connected to the FPGA using one of the following interfaces:

Interface	Note
Gigabit Media Independent Interface (GMII)	Only 1000 Mbits/s speed is supported using this interface.
Reduced Gigabit Media Independent Interface (RGMII)	Only 1000 Mbits/s speed is supported using this interface.
Serial Gigabit Media Independent Interface (SGMII)	Only 1000 Mbits/s speed is supported using this interface.
Media Independent Interface (MII)	Only 100 Mbits/s speed is supported using this interface.

Note: For GMII, the TXCLK (clock signal for 10/100 Mbits signal) signal is not required because only 1000 Mbits/s speed is supported.

In addition to the standard GMII/RGMII/SGMII/MII interface signals, FPGA-in-the-loop also requires an Ethernet PHY chip reset signal (ETH_RESET_n). This active-low reset signal performs the PHY hardware reset by FPGA. It is active-low.

Special Timing Considerations for RGMII

When the RGMII interface is used, the MAC on the FPGA assumes that the data are aligned with the edges of reference clock as specified in the original RGMII v1.3 standard. In this case, PC board designs provide additional trace delay for clock signals.

The RGMII v2.0 standard allows the transmitter to integrate this delay so that PC board delay is not required. Marvell Alaska 88E1111 has internal registers to add internal delays to RX and TX clocks. The internal delays are not added by default, which means that you must use the MDIO module to configure Marvell 88E1111 to add internal delays. For more information on the MDIO module, see "FIL I/O" on page 24-30.

Special Clock Frequency Requirement for GMII/RGMII/SGMII Interface

When GMII/RGMII/SGMII interfaces are used, the FPGA requires an exact 125 MHz clock to drive the 1000 Mbits/s communication. This clock is derived from the user supplied external clock using the clock module or PLL.

Not all external clock frequencies can derive an exact 125 MHz clock frequency. The acceptable clock frequencies vary depending on the FPGA device family. The recommended clock frequencies are 50, 100, 125, and 200 MHz.

JTAG Connection Requirements for FPGA-in-the-Loop

Vendor	Supported Devices	Required Hardware	Required Software
Altera	The FPGA board must be using an FPGA device in the supported Altera FPGA families.	• USB Blaster I or USB Blaster II download cable	• USB Blaster I or II driver • For Windows® operating systems: Quartus Prime executable directory must be on system path. • For Linux® operating systems: versions below Quartus II 13.1 are not supported. Quartus II 14.1 is not supported. Only 64-bit Quartus is supported. Quartus library directory must be on `LD_LIBRARY_PATH` *before* starting MATLAB. Prepend the Linux distribution library path before the Quartus library on `LD_LIBRARY_PATH`. For example, `/lib/x86_64-linux-gnu:$QUARTUS_PATH`.
Xilinx	The board must be using one of the following supported Xilinx FPGAs: Artix®-7, Virtex-7, Kintex®-7 or Zynq 7000.	• Digilent® download cable. If your board has a standard Xilinx 14 pin JTAG connector, you can obtain the HS2 cable from Digilent.	• For Windows operating systems: Xilinx Vivado executable directory must be on system path. • For Linux operating systems: Digilent Adept2

Create Custom FPGA Board Definition

1 Be ready with the following:

   a Board specification document. Any format you are comfortable with is fine. However, if you have it in an electronic version, you can search for the information as it is required.

   b If you plan to validate (test) your board definition file, set up FPGA design software tools:

      For validation, you must have Xilinx or Altera on your path. Use the `hdlsetuptoolpath` function to configure the tool for use with MATLAB.

2 Open the FPGA Board Manager by typing `fpgaBoardManager` in the MATLAB command window. Alternatively, if you are using the HDL Workflow Advisor, you can click **Launch Board Manager** at Step 1.1.

3 Open the New FPGA Board wizard by clicking **Create New Board**. For a description of all the tasks you can perform with the FPGA Board Manager, see "FPGA Board Manager" on page 24-21.

4 The wizard guides you through entering all board information. At each page, fill in the required fields. For assistance in entering board information, see "New FPGA Board Wizard" on page 24-25.

5 Save the board definition file. This step is the last and is automatically instigated when you click **Finish** in the New FPGA Board wizard. See "Save Board Definition File" on page 24-17.

Your custom board definition now appears in the list of available FPGA Boards in the FPGA Board Manager. If you are using HDL Workflow Advisor, it also shows in the **Target platform** list.

Follow the example "Create Xilinx KC705 Evaluation Board Definition File" on page 24-8 for a demonstration of adding a custom FPGA board with the New FPGA Board Manager.

Create Xilinx KC705 Evaluation Board Definition File

In this section...

Overview

For FPGA-in-the-loop, you can use your own qualified FPGA board, even if it is not in the pre-registered FPGA board list supplied by MathWorks. Using the New FPGA Board wizard, you can create a board definition file that describes your custom FPGA board.

In this example, you can follow the workflow of creating a board definition file for the Xilinx KC705 evaluation board to use with FIL simulation.

What You Need to Know Before Starting

- Check the board specification so that you have the following information ready:
  - FPGA interface to the Ethernet PHY chip
  - Clock pins names and numbers
  - Reset pins names and numbers

  In this example, the required information is supplied to you. In general, you can find this type of information in the board specification file. This example uses the KC705 Evaluation Board for the Kintex-7 FPGA User Guide, published by Xilinx.
- For validation, you must have Xilinx or Altera on your path. Use the `hdlsetuptoolpath` function to configure the tool for use with MATLAB.

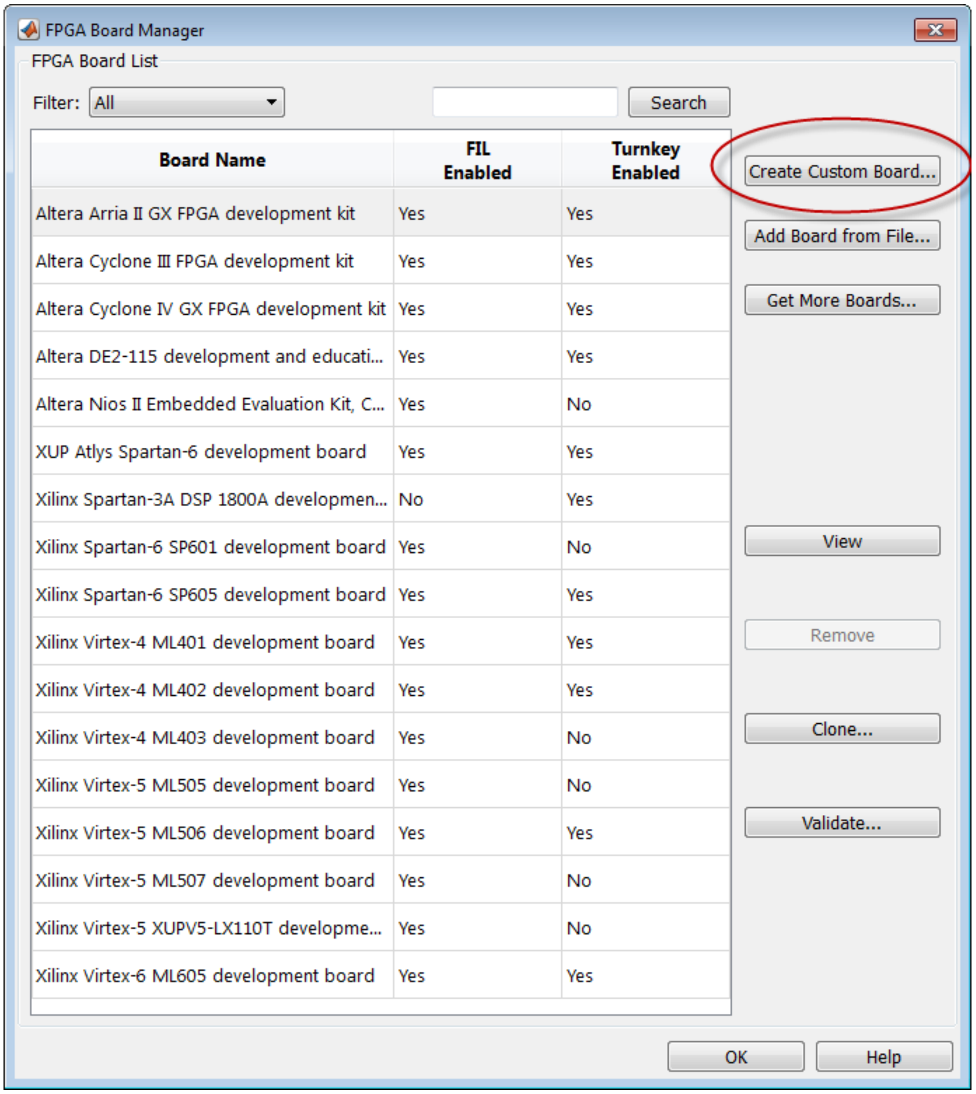
- To verify programming the FPGA board after you add its definition file, attach the custom board to your computer. However, having the board connected is not necessary for creating the board definition file.

Start New FPGA Board Wizard

1 Start the FPGA Board Manager by entering the following command at the MATLAB prompt:

```
>>fpgaBoardManager
```

2 Click **Create Custom Board** to open the New FPGA Board wizard.

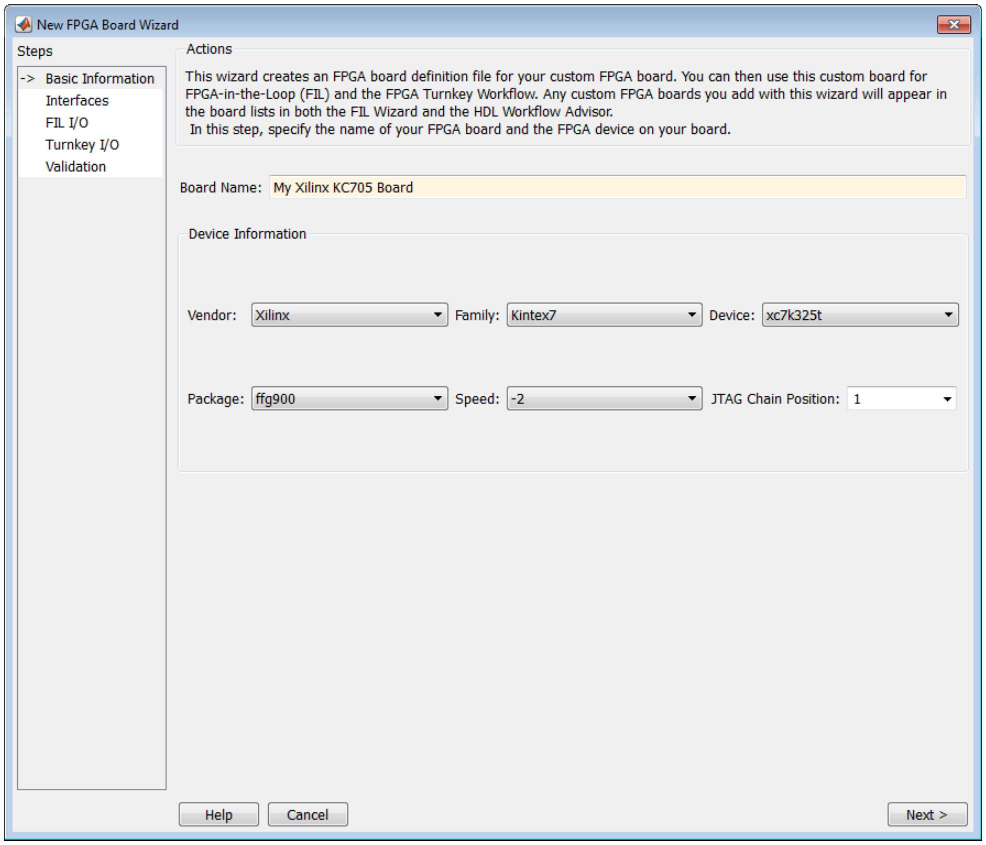


Board Name	FIL Enabled	Turnkey Enabled
Altera Arria II GX FPGA development kit	Yes	Yes
Altera Cyclone III FPGA development kit	Yes	Yes
Altera Cyclone IV GX FPGA development kit	Yes	Yes
Altera DE2-115 development and educati...	Yes	Yes
Altera Nios II Embedded Evaluation Kit, C...	Yes	No
XUP Atlys Spartan-6 development board	Yes	Yes
Xilinx Spartan-3A DSP 1800A developmen...	No	Yes
Xilinx Spartan-6 SP601 development board	Yes	No
Xilinx Spartan-6 SP605 development board	Yes	Yes
Xilinx Virtex-4 ML401 development board	Yes	Yes
Xilinx Virtex-4 ML402 development board	Yes	Yes
Xilinx Virtex-4 ML403 development board	Yes	No
Xilinx Virtex-5 ML505 development board	Yes	No
Xilinx Virtex-5 ML506 development board	Yes	Yes
Xilinx Virtex-5 ML507 development board	Yes	No
Xilinx Virtex-5 XUPV5-LX110T developme...	Yes	No
Xilinx Virtex-6 ML605 development board	Yes	Yes

Provide Basic Board Information

1 In the Basic Information pane, enter the following information:

- **Board Name:** Enter "My Xilinx KC705 Board"
- **Vendor:** Select `Xilinx`
- **Family:** Select `Kintex7`
- **Device:** Select `xc7k325t`
- **Package:** Select `ffg900`
- **Speed:** Select `-2`
- **JTAG Chain Position:** Select `1`



This wizard creates an FPGA board definition file for your custom FPGA board. You can then use this custom board for FPGA-in-the-Loop (FIL) and the FPGA Turnkey Workflow. Any custom FPGA boards you add with this wizard will appear in the board lists in both the FIL Wizard and the HDL Workflow Advisor.
In this step, specify the name of your FPGA board and the FPGA device on your board.

Board Name: My Xilinx KC705 Board

Device Information

Vendor: Xilinx
Family: Kintex7
Device: xc7k325t
Package: ffg900
Speed: -2
JTAG Chain Position: 1

The information you just entered can be found in the KC705 Evaluation Board for the Kintex-7 FPGA User Guide.

2 Click **Next**.

Specify FPGA Interface Information

1 In the Interfaces pane, perform the following tasks.

   a Select **FIL Interface**. This option is required for using your board with FPGA-in-the-loop.

   b Select **GMII** in the PHY Interface Type. This option indicates that the onboard FPGA is connected to the Ethernet PHY chip via a GMII interface.

   c Leave the **User-defined I/O** option in the FPGA Turnkey Interface section cleared. FPGA Turnkey workflow is not the focus of this example.

   d **Clock Frequency:** Enter 200. This Xilinx KC705 board has multiple clock sources. The 200 MHz clock is one of the recommended clock frequencies for use with Ethernet interface (50, 100, 125, and 200 MHz).

   e **Clock Type:** Select `Differential`.

   f **Clock_P Pin Number:** Enter AD12.

   g **Clock_N Pin Number:** Enter AD11.

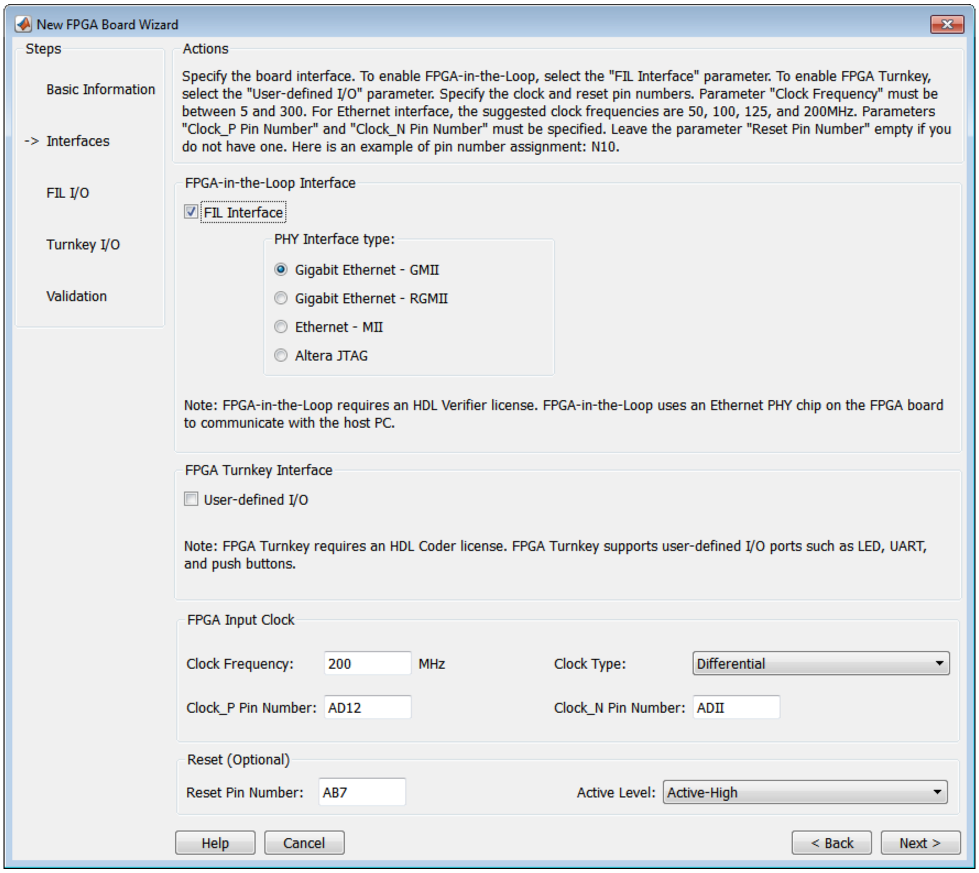
   h **Clock IO Standard** — Leave blank.

   i **Reset Pin Number:** Enter AB7. This value supplies a global reset to the FPGA.

   j **Active Level:** Select `Active-High`.

   k **Reset IO Standard** — Leave blank.

You can obtain all necessary information from the board design specification.



Actions

Specify the board interface. To enable FPGA-in-the-Loop, select the "FIL Interface" parameter. To enable FPGA Turnkey, select the "User-defined I/O" parameter. Specify the clock and reset pin numbers. Parameter "Clock Frequency" must be between 5 and 300. For Ethernet interface, the suggested clock frequencies are 50, 100, 125, and 200MHz. Parameters "Clock_P Pin Number" and "Clock_N Pin Number" must be specified. Leave the parameter "Reset Pin Number" empty if you do not have one. Here is an example of pin number assignment: N10.

FPGA-in-the-Loop Interface

✓ FIL Interface

PHY Interface type:

● Gigabit Ethernet - GMII
○ Gigabit Ethernet - RGMII
○ Ethernet - MII
○ Altera JTAG

Note: FPGA-in-the-Loop requires an HDL Verifier license. FPGA-in-the-Loop uses an Ethernet PHY chip on the FPGA board to communicate with the host PC.

FPGA Turnkey Interface

☐ User-defined I/O

Note: FPGA Turnkey requires an HDL Coder license. FPGA Turnkey supports user-defined I/O ports such as LED, UART, and push buttons.

FPGA Input Clock

Clock Frequency: 200 MHz   Clock Type: Differential

Clock_P Pin Number: AD12   Clock_N Pin Number: AD11

Reset (Optional)

Reset Pin Number: AB7   Active Level: Active-High

2 Click **Next**.

Enter FPGA Pin Numbers

1 In the FILI/O pane, enter the numbers for each FPGA pin. This information is required.

Pin numbers for RXD and TXD signals are entered from the least significant digit (LSD) to the most significant digit (MSB), separated by a comma.

For signal name...	Enter FPGA pin number...
ETH_COL	W19
ETH_CRS	R30
ETH_GTXCLK	K30
ETH_MDC	R23
ETH_MDIO	J21
ETH_RESET_n	L20
ETH_RXCLK	U27
ETH_RXD	U30,U25,T25,U28,R19,T27,T26,T28
ETH_RXDV	R28
ETH_RXER	V26
ETH_TXD	N27,N25,M29,L28,J26,K26,L30,J28
ETH_TXEN	M27
ETH_TXER	N29

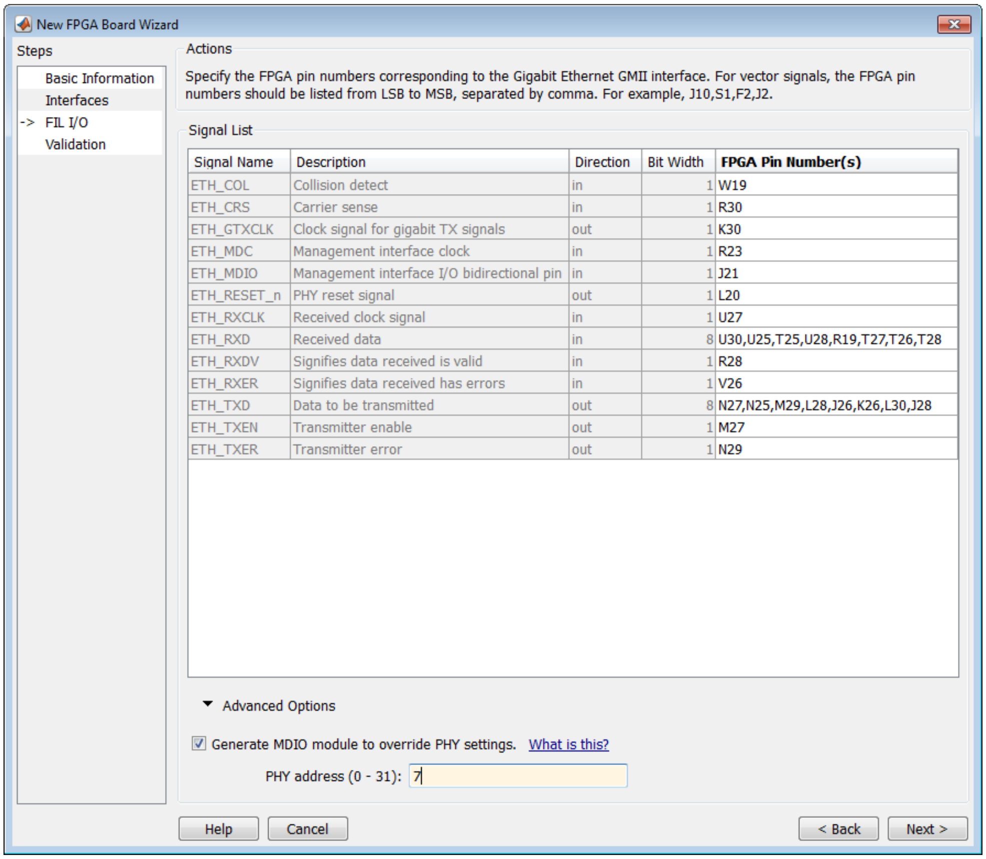
2 Click Advanced Options to expand the section.

3 Check the **Generate MDIO module to override PHY settings** option.

This option is selected for the following reasons:

- There are jumpers on the Xilinx KC705 board that configure the Ethernet PHY device to MII, GMII, RGMII, or SGMII mode. Since this example uses the GMII interfaces, the FPGA board does not work if the PHY devices are set to the wrong mode. When the **Generate MDIO module to override PHY settings** option is selected, the FPGA uses the Management Data Input/Output (MDIO) bus to override the jumper settings and configure the PHY chip to the correct GMII mode.
- This option currently only applies to Marvell Alaska PHY device 88E1111 and this KC705 board is using the Marvel device.

4 **PHY address (0 – 31):** Enter 7.



Specify the FPGA pin numbers corresponding to the Gigabit Ethernet GMII interface. For vector signals, the FPGA pin numbers should be listed from LSB to MSB, separated by comma. For example, J10,S1,F2,J2.

Signal List

Signal Name	Description	Direction	Bit Width	FPGA Pin Number(s)
ETH_COL	Collision detect	in	1	W19
ETH_CRS	Carrier sense	in	1	R30
ETH_GTXCLK	Clock signal for gigabit TX signals	out	1	K30
ETH_MDC	Management interface clock	in	1	R23
ETH_MDIO	Management interface I/O bidirectional pin	in	1	J21
ETH_RESET_n	PHY reset signal	out	1	L20
ETH_RXCLK	Received clock signal	in	1	U27
ETH_RXD	Received data	in	8	U30,U25,T25,U28,R19,T27,T26,T28
ETH_RXDV	Signifies data received is valid	in	1	R28
ETH_RXER	Signifies data received has errors	in	1	V26
ETH_TXD	Data to be transmitted	out	8	N27,N25,M29,L28,J26,K26,L30,J28
ETH_TXEN	Transmitter enable	out	1	M27
ETH_TXER	Transmitter error	out	1	N29

Advanced Options

[✓] Generate MDIO module to override PHY settings. What is this?

PHY address (0 - 31): 7

5 Click **Next**.

Run Optional Validation Tests

This step provides a validation test for you to verify if the entered information is correct by performing FPGA-in-the-loop cosimulation. You need Xilinx ISE 13.4 or higher versions installed on the same computer. This step is optional and you can skip it, if you prefer.

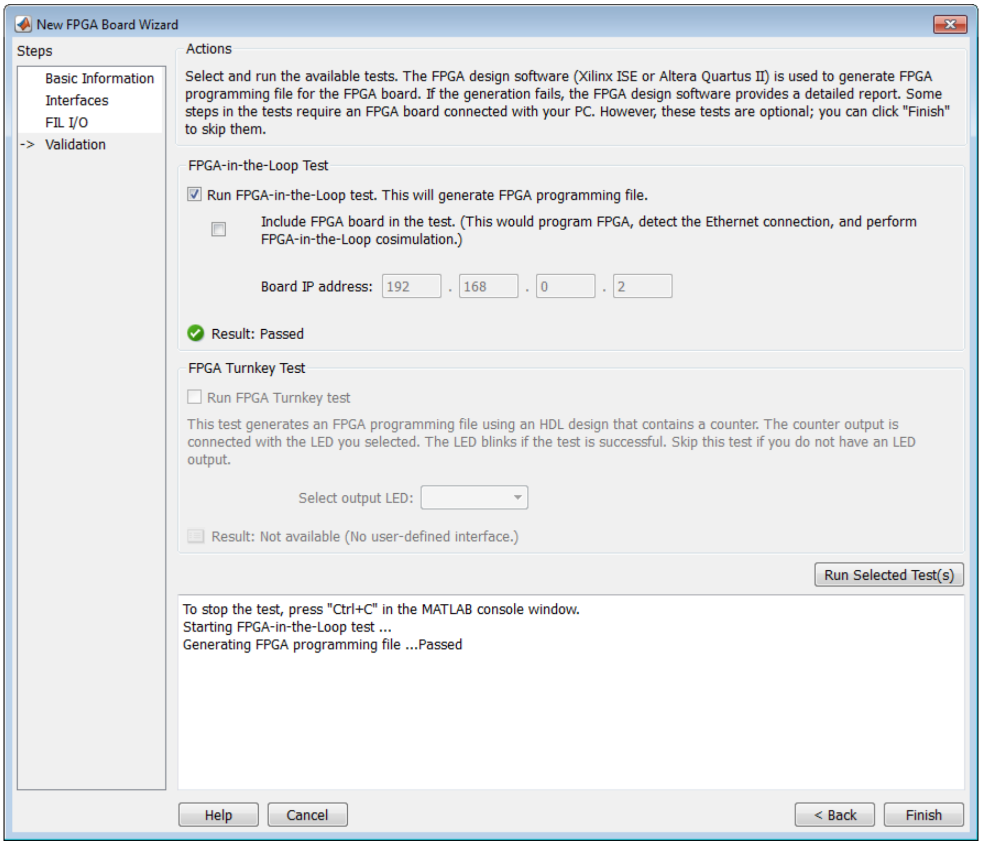
Note: For validation, you must have Xilinx or Altera on your path. Use the `hdlsetuptoolpath` function to configure the tool for use with MATLAB.

To run this test, perform the following actions.

1 Check the **Run FPGA-in-the-Loop test** option.

2 If you have the board attached, check the **Include FPGA board in the test** option. You need to supply the IP address of the FPGA Board. This example assumes that the Xilinx KC705 board is attached to your host computer and it has an IP address of 192.168.0.2.

3 Click **Run Selected Test(s)**. The tests take about 10 minutes to complete.



Select and run the available tests. The FPGA design software (Xilinx ISE or Altera Quartus II) is used to generate FPGA programming file for the FPGA board. If the generation fails, the FPGA design software provides a detailed report. Some steps in the tests require an FPGA board connected with your PC. However, these tests are optional; you can click "Finish" to skip them.

☑ Run FPGA-in-the-Loop test. This will generate FPGA programming file.

☐ Include FPGA board in the test. (This would program FPGA, detect the Ethernet connection, and perform FPGA-in-the-Loop cosimulation.)

Board IP address: 192 . 168 . 0 . 2

Result: Passed

☐ Run FPGA Turnkey test

This test generates an FPGA programming file using an HDL design that contains a counter. The counter output is connected with the LED you selected. The LED blinks if the test is successful. Skip this test if you do not have an LED output.

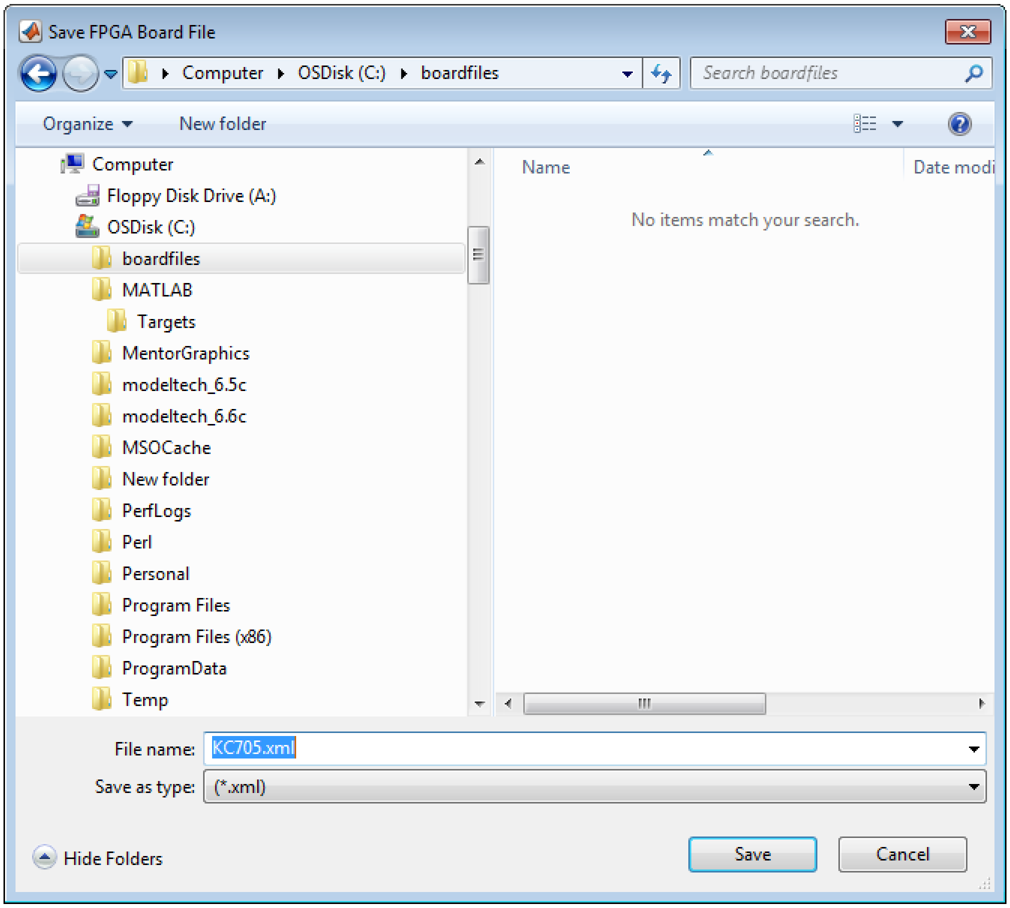
Select output LED:

Result: Not available (No user-defined interface.)

```
To stop the test, press "Ctrl+C" in the MATLAB console window.
Starting FPGA-in-the-Loop test ...
Generating FPGA programming file ...Passed
```

Save Board Definition File

1 Click **Finish** to exit the New FPGA Board wizard. A **Save As** dialog box pops up and asks for the location of the FPGA board definition file. For this example, save as `C:\boardfiles\KC705.xml`.



No items match your search.

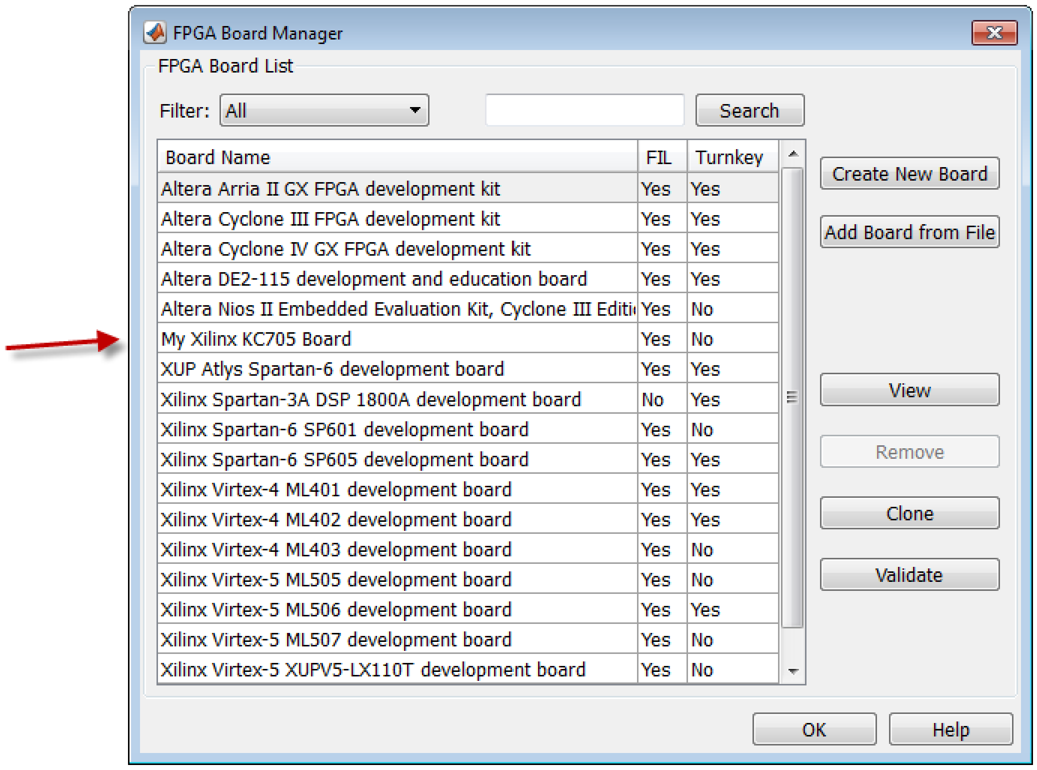
File name: KC705.xml

Save as type: (*.xml)

2 Click **Save** to save the file and exit.

Use New FPGA Board

1 After you save the board definition file, you are returned to the FPGA Board Manager. In the FPGA Board List, you can now see the new board you defined.



Board Name	FIL	Turnkey
Altera Arria II GX FPGA development kit	Yes	Yes
Altera Cyclone III FPGA development kit	Yes	Yes
Altera Cyclone IV GX FPGA development kit	Yes	Yes
Altera DE2-115 development and education board	Yes	Yes
Altera Nios II Embedded Evaluation Kit, Cyclone III Editi	Yes	No
My Xilinx KC705 Board	Yes	No
XUP Atlys Spartan-6 development board	Yes	Yes
Xilinx Spartan-3A DSP 1800A development board	No	Yes
Xilinx Spartan-6 SP601 development board	Yes	No
Xilinx Spartan-6 SP605 development board	Yes	Yes
Xilinx Virtex-4 ML401 development board	Yes	Yes
Xilinx Virtex-4 ML402 development board	Yes	Yes
Xilinx Virtex-4 ML403 development board	Yes	No
Xilinx Virtex-5 ML505 development board	Yes	No
Xilinx Virtex-5 ML506 development board	Yes	Yes
Xilinx Virtex-5 ML507 development board	Yes	No
Xilinx Virtex-5 XUPV5-LX110T development board	Yes	No

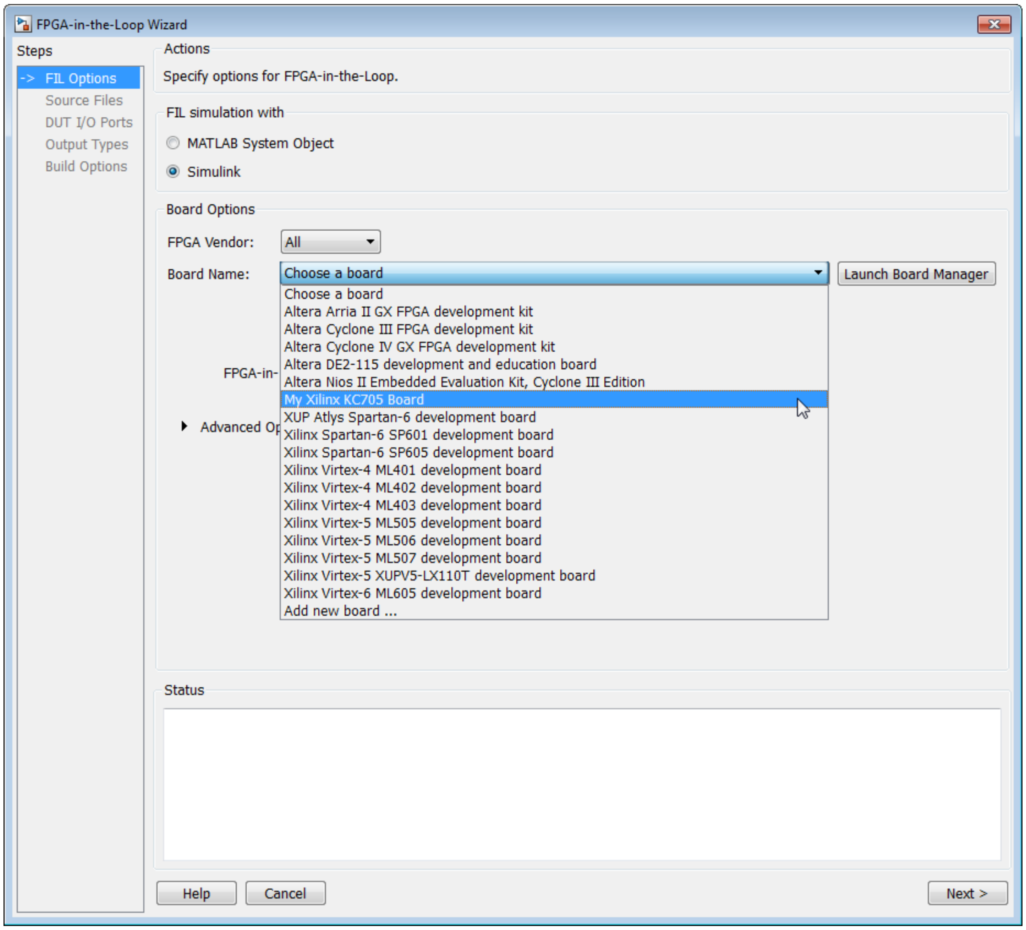
Click **OK** to close the FPGA Board Manager.

2 You can view the new board in the board list from either the FIL wizard or the HDL Workflow Advisor.

a Start the FIL wizard from the MATLAB prompt.

```
>>filWizard
```

The Xilinx KC705 board appears in the board list and you can select it for FPGA-in-the-loop simulation.


FPGA-in-the-Loop Wizard

Steps
- -> FIL Options
- Source Files
- DUT I/O Ports
- Output Types
- Build Options

Actions

Specify options for FPGA-in-the-Loop.

FIL simulation with
- MATLAB System Object
- [✓ Simulink]

Board Options

FPGA Vendor: All

Board Name: Choose a board

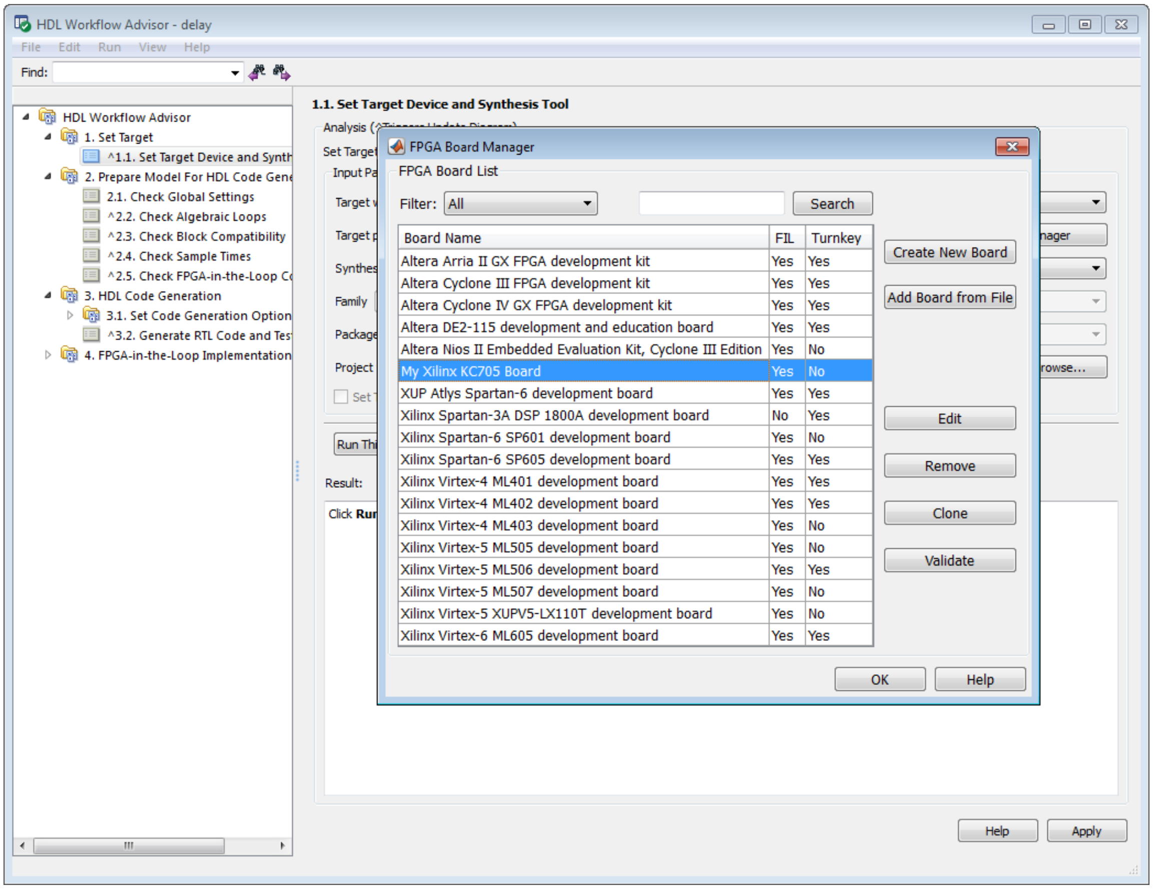
Launch Board Manager

- Choose a board
- Altera Arria II GX FPGA development kit
- Altera Cyclone III FPGA development kit
- Altera Cyclone IV GX FPGA development kit
- Altera DE2-115 development and education board
- Altera Nios II Embedded Evaluation Kit, Cyclone III Edition
- My Xilinx KC705 Board
- XUP Atlys Spartan-6 development board
- Xilinx Spartan-6 SP601 development board
- Xilinx Spartan-6 SP605 development board
- Xilinx Virtex-4 ML401 development board
- Xilinx Virtex-4 ML402 development board
- Xilinx Virtex-4 ML403 development board
- Xilinx Virtex-5 ML505 development board
- Xilinx Virtex-5 ML506 development board
- Xilinx Virtex-5 ML507 development board
- Xilinx Virtex-5 XUPV5-LX110T development board
- Xilinx Virtex-6 ML605 development board
- Add new board ...

b Start HDL Workflow Advisor.

In step 1.1, select `FPGA-in-the-Loop` and click **Launch Board Manager**.

The Xilinx KC705 board appears in the board list and you can select it for FPGA-in-the-loop simulation.


HDL Workflow Advisor - delay
File Edit Run View Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synth
2. Prepare Model For HDL Code Gene
2.1. Check Global Settings
^2.2. Check Algebraic Loops
^2.3. Check Block Compatibility
^2.4. Check Sample Times
^2.5. Check FPGA-in-the-Loop C
3. HDL Code Generation
3.1. Set Code Generation Option
^3.2. Generate RTL Code and Tes
4. FPGA-in-the-Loop Implementation
1.1. Set Target Device and Synthesis Tool
FPGA Board Manager
FPGA Board List
Filter: All
Search
Create New Board
Add Board from File
Edit
Remove
Clone
Validate
OK
Help
Apply

Board Name	FIL	Turnkey
Altera Arria II GX FPGA development kit	Yes	Yes
Altera Cyclone III FPGA development kit	Yes	Yes
Altera Cyclone IV GX FPGA development kit	Yes	Yes
Altera DE2-115 development and education board	Yes	Yes
Altera Nios II Embedded Evaluation Kit, Cyclone III Edition	Yes	No
My Xilinx KC705 Board	Yes	No
XUP Atlys Spartan-6 development board	Yes	Yes
Xilinx Spartan-3A DSP 1800A development board	No	Yes
Xilinx Spartan-6 SP601 development board	Yes	No
Xilinx Spartan-6 SP605 development board	Yes	Yes
Xilinx Virtex-4 ML401 development board	Yes	Yes
Xilinx Virtex-4 ML402 development board	Yes	Yes
Xilinx Virtex-4 ML403 development board	Yes	No
Xilinx Virtex-5 ML505 development board	Yes	No
Xilinx Virtex-5 ML506 development board	Yes	Yes
Xilinx Virtex-5 ML507 development board	Yes	No
Xilinx Virtex-5 XUPV5-LX110T development board	Yes	No
Xilinx Virtex-6 ML605 development board	Yes	Yes

FPGA Board Manager

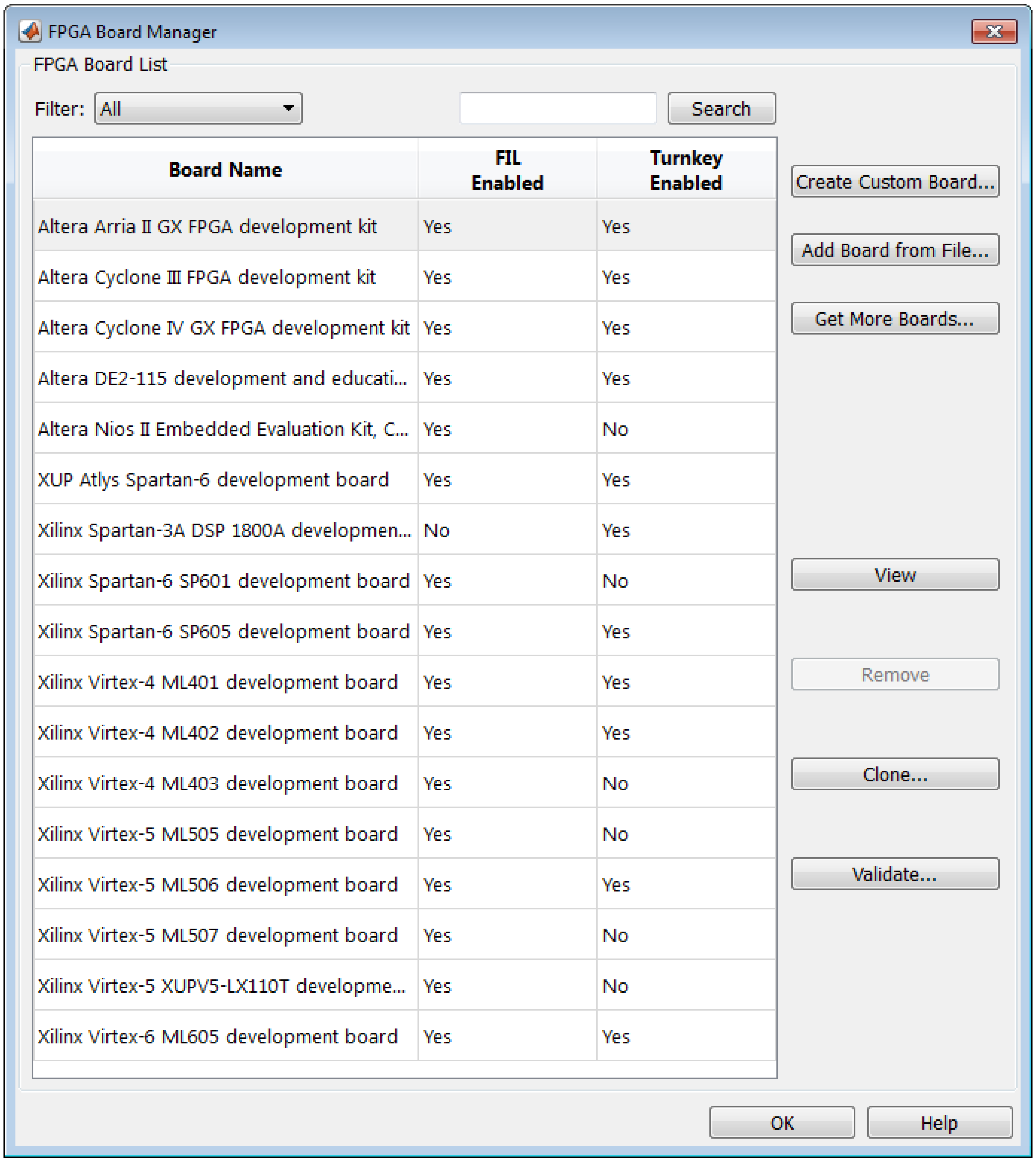
In this section...

Introduction

The FPGA Board Manager is the portal to managing custom FPGA boards. You can create a board definition file or edit an existing one. You can even import a custom board from an existing board definition file.

You start the FPGA Board Manager by one of the following methods:

- By typing `fpgaBoardManager` in the MATLAB command window
- From the FIL wizard by clicking **Launch Board Manager** on the first page
- From the HDL Workflow Advisor (when using HDL Coder) at Step 1.1



Board Name	FIL Enabled	Turnkey Enabled
Altera Arria II GX FPGA development kit	Yes	Yes
Altera Cyclone III FPGA development kit	Yes	Yes
Altera Cyclone IV GX FPGA development kit	Yes	Yes
Altera DE2-115 development and educati...	Yes	Yes
Altera Nios II Embedded Evaluation Kit, C...	Yes	No
XUP Atlys Spartan-6 development board	Yes	Yes
Xilinx Spartan-3A DSP 1800A developmen...	No	Yes
Xilinx Spartan-6 SP601 development board	Yes	No
Xilinx Spartan-6 SP605 development board	Yes	Yes
Xilinx Virtex-4 ML401 development board	Yes	Yes
Xilinx Virtex-4 ML402 development board	Yes	Yes
Xilinx Virtex-4 ML403 development board	Yes	No
Xilinx Virtex-5 ML505 development board	Yes	No
Xilinx Virtex-5 ML506 development board	Yes	Yes
Xilinx Virtex-5 ML507 development board	Yes	No
Xilinx Virtex-5 XUPV5-LX110T developme...	Yes	No
Xilinx Virtex-6 ML605 development board	Yes	Yes

Filter

Choose one of the following views:

- All boards
- Only those boards that were preinstalled with HDL Verifier or HDL Coder
- Only custom boards

Search

Find a specific board in the list or those boards that fully or partially match your search string.

FIL Enabled/Turnkey Enabled

These columns indicate whether the specified board is supported for FIL or Turnkey operations.

Create Custom Board

Start New FPGA Board wizard. See “New FPGA Board Wizard” on page 24-25. You can find the process for creating a board definition file in “Create Custom FPGA Board Definition” on page 24-7.

Add Board from File

Import a board definition file (.xml).

Get More Boards

Download FPGA board support packages for use with FIL

1. Click **Get more boards**.
2. Follow the prompts in the Support Package Installer to download an FPGA board support package.
3. When the download is complete, you can see the new boards in the board list in the FPGA Board Manager.

Offline Support Package Installation You can install an FPGA board support package without an internet connection. See “Install Support Package Offline” (HDL Verifier).

View/Edit

View board configurations and modify the information. You can view a read-only file but not edit it. See “FPGA Board Editor” on page 24-38.

Remove

Remove custom board from the list. This action does not delete the board definition XML file.

Clone

Makes a copy of an existing custom board for further modification.

Validate

Runs the validation tests for FIL See “Run Optional Validation Tests” on page 24-15.

New FPGA Board Wizard

Using the New FPGA Board wizard, you can enter all the required information to add a board to the FPGA board list. This list applies to both FIL and Turnkey workflows. Review "FPGA Board Requirements" on page 24-3 before adding an FPGA board to make sure that it is compatible with the workflow for which you want to use it.

Several buttons in the New FPGA Board wizard help with navigation:

- **Back**: Go to a previous page to review or edit data already entered.
- **Next**: Go to next page when all requirements of current page have been satisfied.
- **Help**: Open Doc Center, and display this topic.
- **Cancel**: Exit New FPGA Board wizard. You can exit with or without saving the information from your session.

Adding Boards Once for Multiple Users To add new boards globally, follow these instructions. To access a board added globally, all users must be using the same MATLAB installation.

1. Create the following folder:

   matlabroot`/toolbox/shared/eda/board/boardfiles`
2. Copy the board description XML file to the `boardfiles` folder.
3. After copying the XML file, restart MATLAB. The new board appears in the FPGA board list for either or both the FIL and Turnkey workflows.

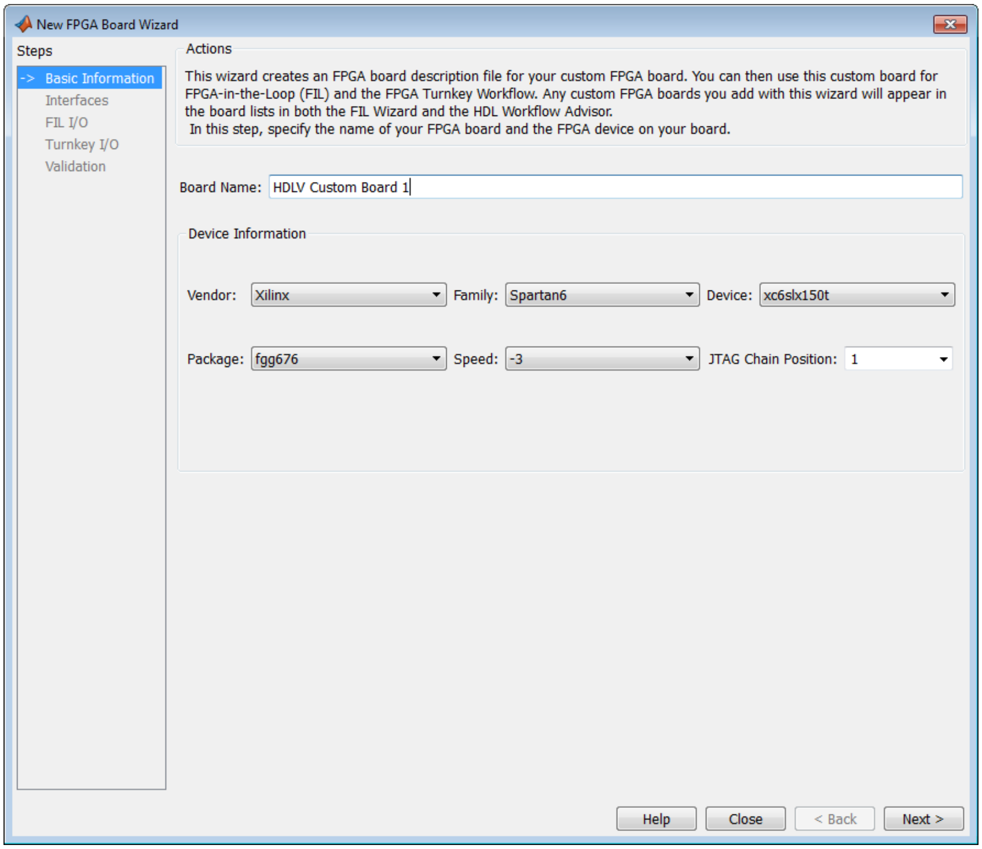
All boards under this folder show-up in the FPGA board list automatically for users with the same MATLAB installation. You do not need to use FPGA Board Manager to add these boards again.

The workflow for adding an FPGA board contains these steps:

In this section...
"Basic Information" on page 24-26
"Interfaces" on page 24-27
"FIL I/O" on page 24-30

In this section...
"Turnkey I/O" on page 24-33
"Validation" on page 24-36
"Finish" on page 24-37

Basic Information



This wizard creates an FPGA board description file for your custom FPGA board. You can then use this custom board for FPGA-in-the-Loop (FIL) and the FPGA Turnkey Workflow. Any custom FPGA boards you add with this wizard will appear in the board lists in both the FIL Wizard and the HDL Workflow Advisor.
In this step, specify the name of your FPGA board and the FPGA device on your board.

Board Name: HDLV Custom Board 1

Vendor: Xilinx Family: Spartan6 Device: xc6slx150t

Package: fgg676 Speed: -3 JTAG Chain Position: 1

Board Name: Enter a unique board name.

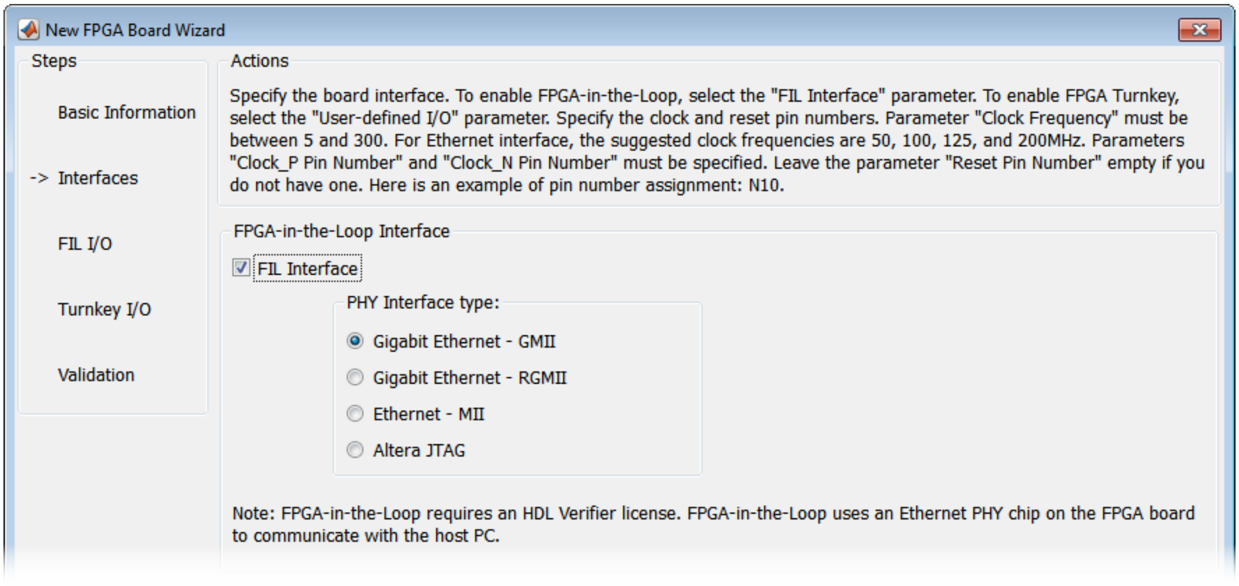
Device Information:

- **Vendor**: `Xilinx` or `Altera`

- **Family**: Family depends on the specified vendor. See the board specification file for applicable settings.
- **Device**: Use the board specification file to select the correct device.
- For Xilinx boards only:
  - **Package**: Use the board specification file to select the correct package.
  - **Speed**: Use the board specification file to select the correct speed.
  - **JTAG Chain Position**: Value indicates the starting position for JTAG chain. Consult the board specification file for this information.

Interfaces

FIL Interface for Altera Boards



Specify the board interface. To enable FPGA-in-the-Loop, select the "FIL Interface" parameter. To enable FPGA Turnkey, select the "User-defined I/O" parameter. Specify the clock and reset pin numbers. Parameter "Clock Frequency" must be between 5 and 300. For Ethernet interface, the suggested clock frequencies are 50, 100, 125, and 200MHz. Parameters "Clock_P Pin Number" and "Clock_N Pin Number" must be specified. Leave the parameter "Reset Pin Number" empty if you do not have one. Here is an example of pin number assignment: N10.

FPGA-in-the-Loop Interface

[✓] FIL Interface

PHY Interface type:

● Gigabit Ethernet - GMII
○ Gigabit Ethernet - RGMII
○ Ethernet - MII
○ Altera JTAG

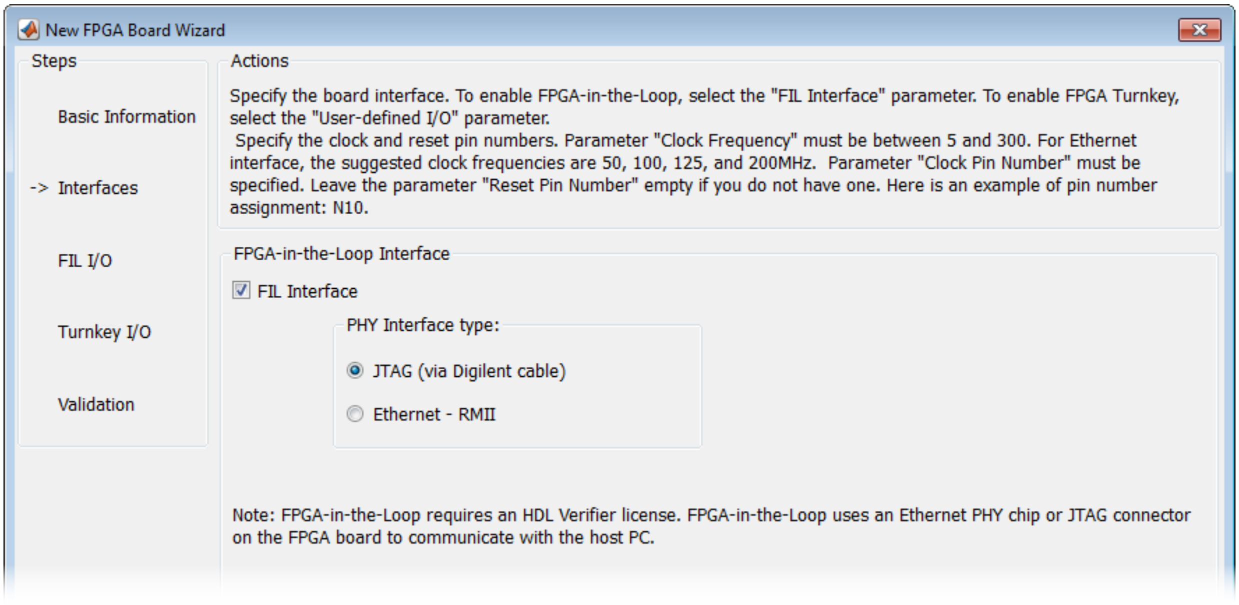
Note: FPGA-in-the-Loop requires an HDL Verifier license. FPGA-in-the-Loop uses an Ethernet PHY chip on the FPGA board to communicate with the host PC.

1 **FPGA-in-the-Loop**: To use this board with FIL, select **FIL Interface**.
2 Select one of the following **PHY Interface types**:

- **Gigabit Ethernet — GMII**
- **Gigabit Ethernet — RGMII**
- **Gigabit Ethernet — SGMII** (the SGMII option appears if you select a board from the Stratix® V or Stratix IV device families)
- **Ethernet — MII**
- **Altera JTAG** (Altera boards only)

Note: Not all interfaces are available for all boards. Availability depends on the board you selected in Basic Information.

FIL Interface for Xilinx Boards


New FPGA Board Wizard

Steps
- Basic Information
- -> Interfaces
- FIL I/O
- Turnkey I/O
- Validation

Actions

Specify the board interface. To enable FPGA-in-the-Loop, select the "FIL Interface" parameter. To enable FPGA Turnkey, select the "User-defined I/O" parameter.
Specify the clock and reset pin numbers. Parameter "Clock Frequency" must be between 5 and 300. For Ethernet interface, the suggested clock frequencies are 50, 100, 125, and 200MHz. Parameter "Clock Pin Number" must be specified. Leave the parameter "Reset Pin Number" empty if you do not have one. Here is an example of pin number assignment: N10.

FPGA-in-the-Loop Interface

☑ FIL Interface

PHY Interface type:
- ● JTAG (via Digilent cable)
- ○ Ethernet - RMII

Note: FPGA-in-the-Loop requires an HDL Verifier license. FPGA-in-the-Loop uses an Ethernet PHY chip or JTAG connector on the FPGA board to communicate with the host PC.

1 **FPGA-in-the-Loop Interface**: To use this board with FIL, select **FIL Interface**.
2 Select one of the following **PHY Interface types**:

- **JTAG (via Digilent cable)** (Xilinx boards only)
- **Ethernet — RMII**

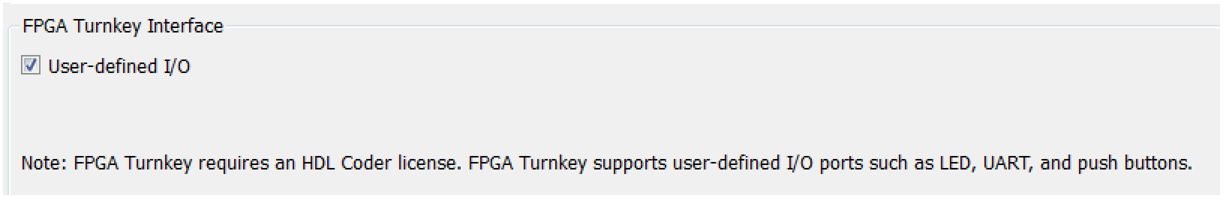
Note: Not all interfaces are available for all boards. Availability depends on the board you selected in Basic Information.

For more information on how to set up the JTAG connection for Xilinx boards, see “JTAG with Digilent Cable Setup” on page 24-40.

Limitations

When you simulate your FPGA design through a Digilent JTAG cable, you cannot use any other debugging feature that requires access to the JTAG; for example, the Vivado Logic Analyzer.

FPGA Turnkey Interface

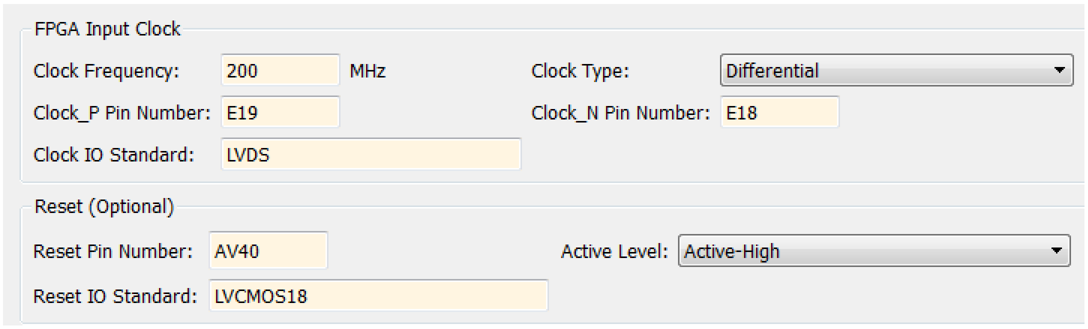

FPGA Turnkey Interface

☑ User-defined I/O

Note: FPGA Turnkey requires an HDL Coder license. FPGA Turnkey supports user-defined I/O ports such as LED, UART, and push buttons.

FPGA Turnkey Interface: If you want to use with board with the HDL Coder FPGA Turnkey workflow, select **User-defined I/O**.

FPGA Input Clock and Reset


FPGA Input Clock

Clock Frequency: 200 MHz
Clock Type: Differential
Clock_P Pin Number: E19
Clock_N Pin Number: E18
Clock IO Standard: LVDS

Reset (Optional)

Reset Pin Number: AV40
Active Level: Active-High
Reset IO Standard: LVCMOS18

1 **FPGA Input Clock** — Clock details are required for both workflows. You can find all necessary information in the board specification file.

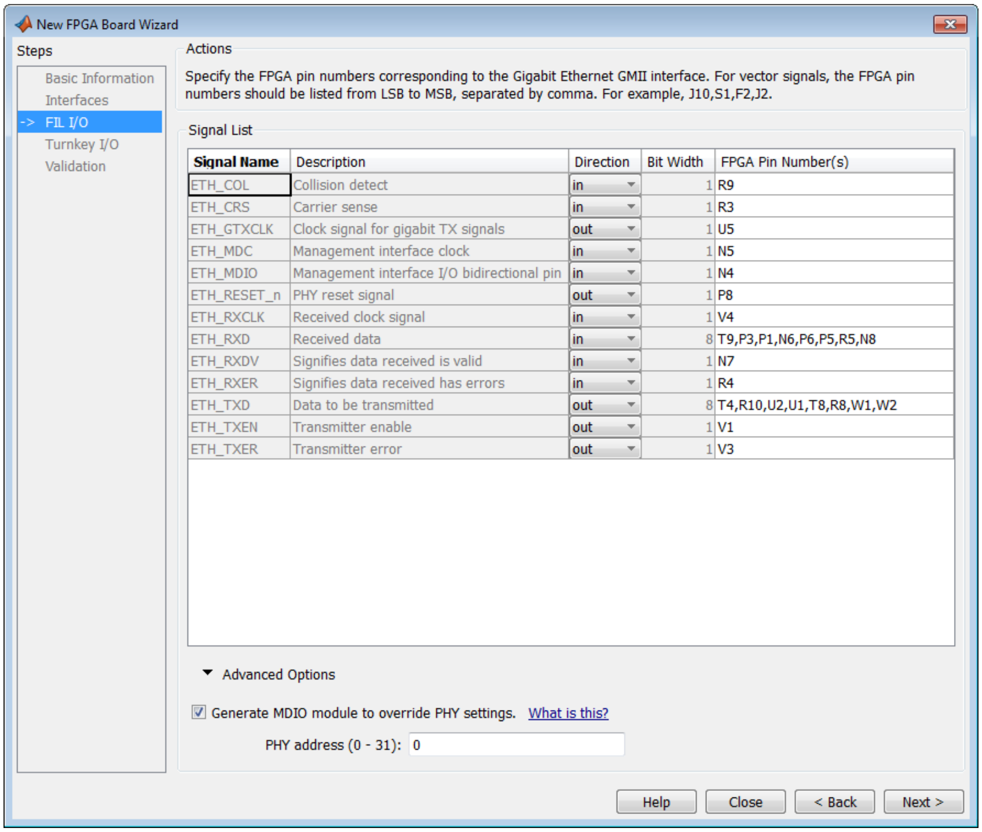
   - **Clock Frequency** — Must be from 5 through 300. For an Ethernet interface, the suggested clock frequencies are 50, 100, 125, and 200 MHz.
   - **Clock Type** — `Single_Ended` or `Differential`.
   - **Clock Pin Number** (Single_Ended) — Must be specified. Example: `N10`.
   - **Clock_P Pin Number** (Differential) — Must be specified. Example: `E19`.
   - **Clock_N Pin Number** (Differential) — Must be specified. Example: `E18`.
   - **Clock IO Standard** — The programmable I/O Standard to use to configure input, output, or bi-directional ports. For example, `LVDS`.

2 **Reset (Optional)** — If you want to indicate a reset, find the pin number and active level in the board specification file, and enter that information.

   - **Reset Pin Number** — Leave empty if you do not have one.
   - **Active Level** — `Active-Low` or `Active-High`.
   - **Reset IO Standard** — The programmable I/O Standard to use to configure input, output, or bi-directional ports. For example, `LVCMOS33`.

FIL I/O

When you select an Ethernet connection to your board, you must specify pins for the Ethernet signals on the FPGA.



Specify the FPGA pin numbers corresponding to the Gigabit Ethernet GMII interface. For vector signals, the FPGA pin numbers should be listed from LSB to MSB, separated by comma. For example, J10,S1,F2,J2.

Signal List

Signal Name	Description	Direction	Bit Width	FPGA Pin Number(s)
ETH_COL	Collision detect	in	1	R9
ETH_CRS	Carrier sense	in	1	R3
ETH_GTXCLK	Clock signal for gigabit TX signals	out	1	U5
ETH_MDC	Management interface clock	in	1	N5
ETH_MDIO	Management interface I/O bidirectional pin	in	1	N4
ETH_RESET_n	PHY reset signal	out	1	P8
ETH_RXCLK	Received clock signal	in	1	V4
ETH_RXD	Received data	in	8	T9,P3,P1,N6,P6,P5,R5,N8
ETH_RXDV	Signifies data received is valid	in	1	N7
ETH_RXER	Signifies data received has errors	in	1	R4
ETH_TXD	Data to be transmitted	out	8	T4,R10,U2,U1,T8,R8,W1,W2
ETH_TXEN	Transmitter enable	out	1	V1
ETH_TXER	Transmitter error	out	1	V3

Advanced Options

[✓] Generate MDIO module to override PHY settings. What is this?

PHY address (0 - 31): 0

Signal List: Provide all the FPGA pin numbers for the specified signals. You can find this information in the board specification file. For vector signals, list all pin numbers on the same line, separated by commas.

Note: If your PHY chip does not have the optional TX_ER pin, tie ETH_TXER to one of the unused pins on the FPGA.

Generate MDIO module to override PHY settings: See the next section on FPGA Board Management Data Input/Output Bus (MDIO) to determine when to use this feature. If you do select this option, enter the PHY address.

What Is the Management Data Input/Output Bus?

Management Data Input/Output (MDIO) is a serial bus, defined in the IEEE® 802.3 standard, that connects MAC devices and Ethernet PHY devices. The FPGA MAC uses the MDIO bus to set control registers in the Ethernet PHY device on the board.

Currently only the Marvell 88E1111 PHY chip is supported by this MDIO module implementation. Do not select this check box if you are not using Marvell 88E1111.

The generated MDIO module is used to perform the following operations:

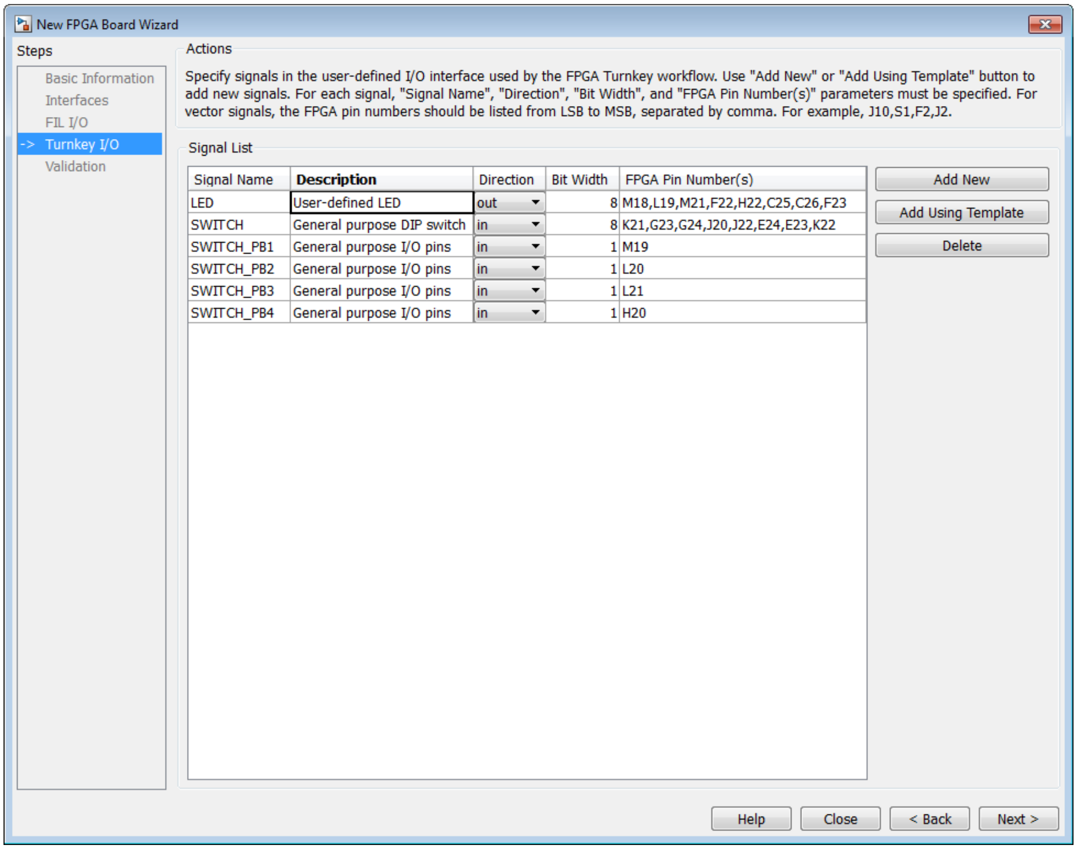
- **GMII mode**: The PHY device can start up using other modes, such as RGMII/SGMII. The generated MDIO module sets the PHY chip in GMII mode.
- **RGMII mode**: The PHY device can start up using other modes, such as GMII/SGMII. The generated MDIO module sets the PHY device in RGMII mode. In addition, the module sets the PHY chip to add internal delay for RX and TX clocks.
- **SGMII mode**: The PHY device can start up using other modes, such as RGMII/GMII. The generated MDIO module sets the PHY chip in SGMII mode.
- **MII mode**: The generated MDIO module sets the PHY device in GMII compatible mode. The module also sets the autonegotiation register to remove the 1000 Base-T capability advertisement. This reset ensures that the autonegotiation process does not select 1000 Mbits/s speed, which is not supported in MII mode.

When To Select MDIO: Select the **Generate MDIO module to override PHY settings** option when both the following conditions are met:

- The onboard Ethernet PHY device is Marvell 88E1111.
- The PHY device startup settings are not compatible with the FPGA MAC. The MDIO modules for different PHY modes must override these settings, as previously described.

Specifying the PHY Address: The PHY address is a 5-bit integer. The value is determined by the CONFIG[0] and CONFIG[1] pin on Marvell 88E1111 PHY device. See the board manual for this value.

Turnkey I/O



Specify signals in the user-defined I/O interface used by the FPGA Turnkey workflow. Use "Add New" or "Add Using Template" button to add new signals. For each signal, "Signal Name", "Direction", "Bit Width", and "FPGA Pin Number(s)" parameters must be specified. For vector signals, the FPGA pin numbers should be listed from LSB to MSB, separated by comma. For example, J10,S1,F2,J2.

Signal List

Signal Name	Description	Direction	Bit Width	FPGA Pin Number(s)
LED	User-defined LED	out	8	M18,L19,M21,F22,H22,C25,C26,F23
SWITCH	General purpose DIP switch	in	8	K21,G23,G24,J20,J22,E24,E23,K22
SWITCH_PB1	General purpose I/O pins	in	1	M19
SWITCH_PB2	General purpose I/O pins	in	1	L20
SWITCH_PB3	General purpose I/O pins	in	1	L21
SWITCH_PB4	General purpose I/O pins	in	1	H20

Note: Provide FIL I/O for an Ethernet connection only. Define at least one output port for the Turnkey I/O interface.

Signal List: Provide all the FPGA pin numbers for the specified signals. You can find this information in the board specification file. For vector signals, list all pin numbers on the same line, separated by commas. The number of pin numbers must match the bit width of the corresponding signal.

Add New: You are prompted to enter all entries in the signal list manually.

Add Using Template: The wizard prepopulates a new signal entry for UART, LED, GPIO, or DIP Switch signals with the following:

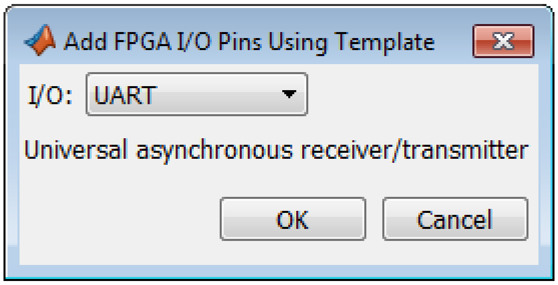
- A generic signal name
- Description
- Direction
- Bit width

You can change the values in any of these prepopulated fields.

Delete: Delete the selected signal from list.

The following example demonstrates using the **Add Using Template** feature.

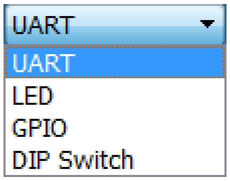
1 In the Turnkey I/O dialog box, click **Add Using Template**.
2 You can now view the template dialog box.



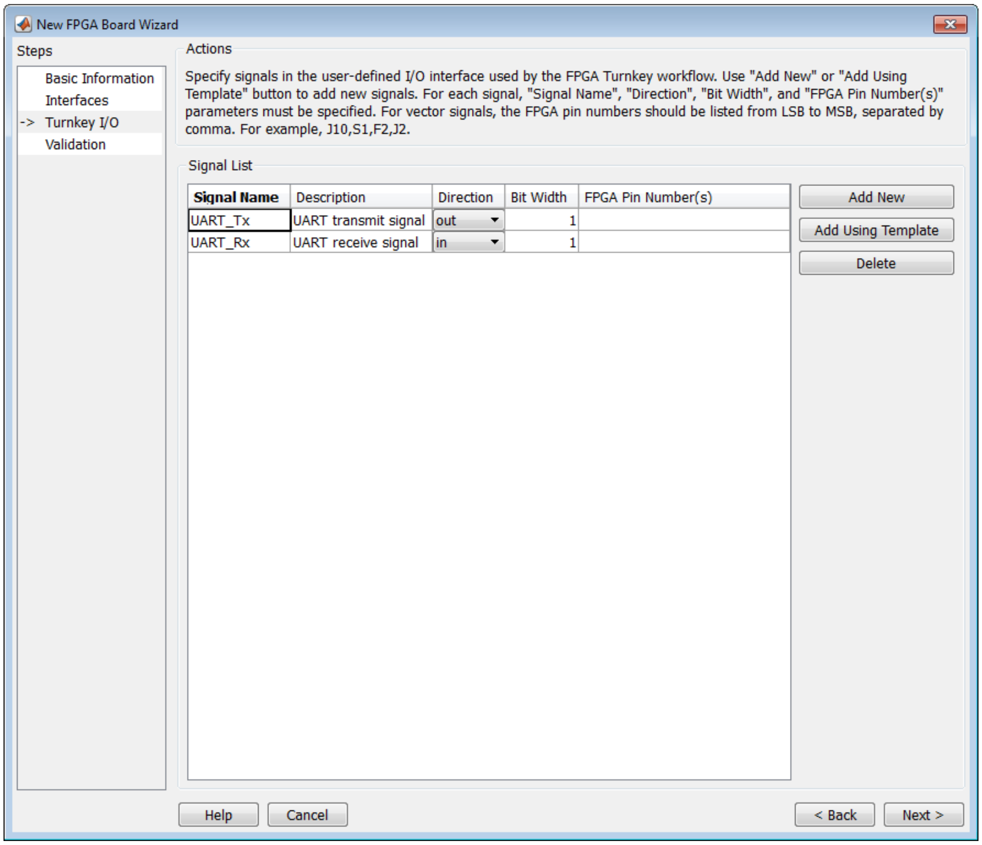
I/O: UART

Universal asynchronous receiver/transmitter

3 Pull down the I/O list and select from the following options:



4 Click **OK**.
5 The wizard adds the specified signal (or signals) to the I/O list.



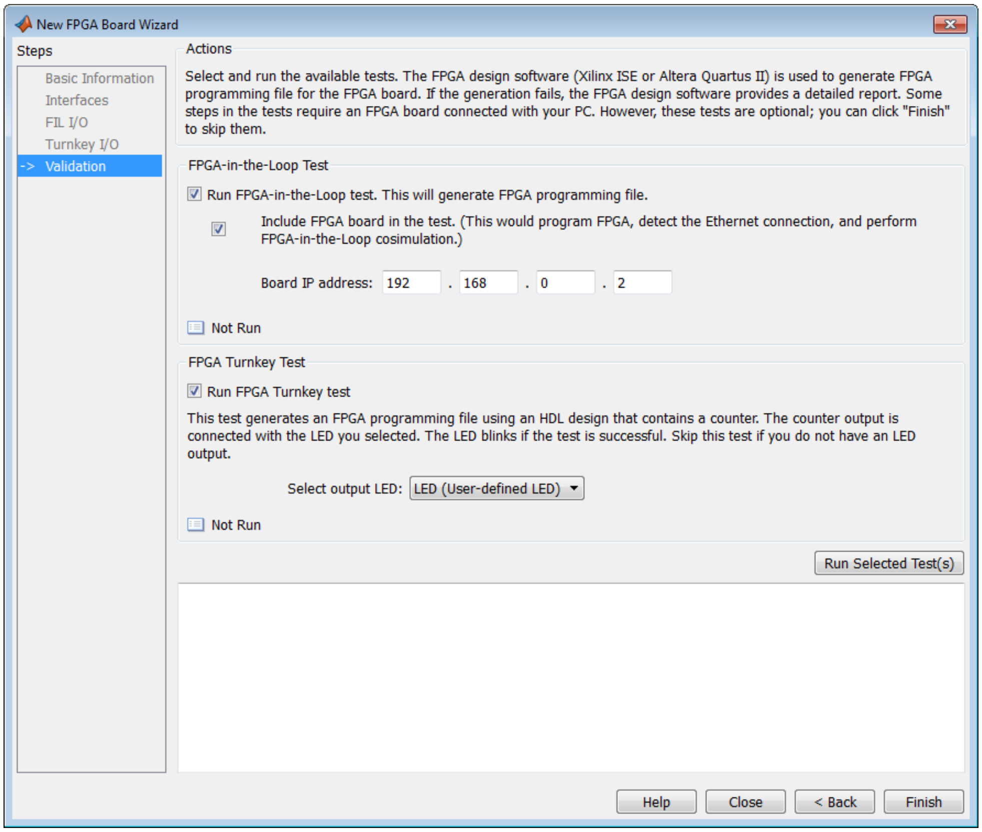
Actions

Specify signals in the user-defined I/O interface used by the FPGA Turnkey workflow. Use "Add New" or "Add Using Template" button to add new signals. For each signal, "Signal Name", "Direction", "Bit Width", and "FPGA Pin Number(s)" parameters must be specified. For vector signals, the FPGA pin numbers should be listed from LSB to MSB, separated by comma. For example, J10,S1,F2,J2.

Signal List

Signal Name	Description	Direction	Bit Width	FPGA Pin Number(s)
UART_Tx	UART transmit signal	out	1	
UART_Rx	UART receive signal	in	1	

Validation



Actions

Select and run the available tests. The FPGA design software (Xilinx ISE or Altera Quartus II) is used to generate FPGA programming file for the FPGA board. If the generation fails, the FPGA design software provides a detailed report. Some steps in the tests require an FPGA board connected with your PC. However, these tests are optional; you can click "Finish" to skip them.

FPGA-in-the-Loop Test

☑ Run FPGA-in-the-Loop test. This will generate FPGA programming file.

☑ Include FPGA board in the test. (This would program FPGA, detect the Ethernet connection, and perform FPGA-in-the-Loop cosimulation.)

Board IP address: 192 . 168 . 0 . 2

Not Run

FPGA Turnkey Test

☑ Run FPGA Turnkey test

This test generates an FPGA programming file using an HDL design that contains a counter. The counter output is connected with the LED you selected. The LED blinks if the test is successful. Skip this test if you do not have an LED output.

Select output LED: LED (User-defined LED)

Not Run

FPGA-in-the-Loop Test

- **Run FPGA-in-the-Loop test**: Select to generate an FPGA programming file.
  - **Include FPGA board in the test**: (Optional) This selection program the FPGA with the generated programming file, detects the Ethernet connection (if selected), and performs FPGA-in-the-loop simulation.
  - **Board IP address**: (Ethernet connection only) Use this option for setting the board IP address if it is not the default IP address (192.168.0.2).

    If necessary, change the computer IP address to a different subnet from 192.168.0.x when you set up the network adapter. If the default board IP address

192.168.0.2 is in use by another device, change the Board IP address according to the following guidelines:

- The subnet address, typically the first 3 bytes of board IP address, must be the same as the host IP address.
- The last byte of the board IP address must be different from the host IP address.
- The board IP address must not conflict with the IP addresses of other computers.

  For example, if the host IP address is 192.168.8.2, then you can use 192.168.8.3, if available.

FPGA Turnkey Test

- **Run FPGA Turnkey test**: Select to generate an FPGA programming file using an HDL design that contains a counter. You must have a board attached.
- **Select output LED**: The counter's output is connected with the LED you select. Skip this test if you do not have an LED output.

Finish

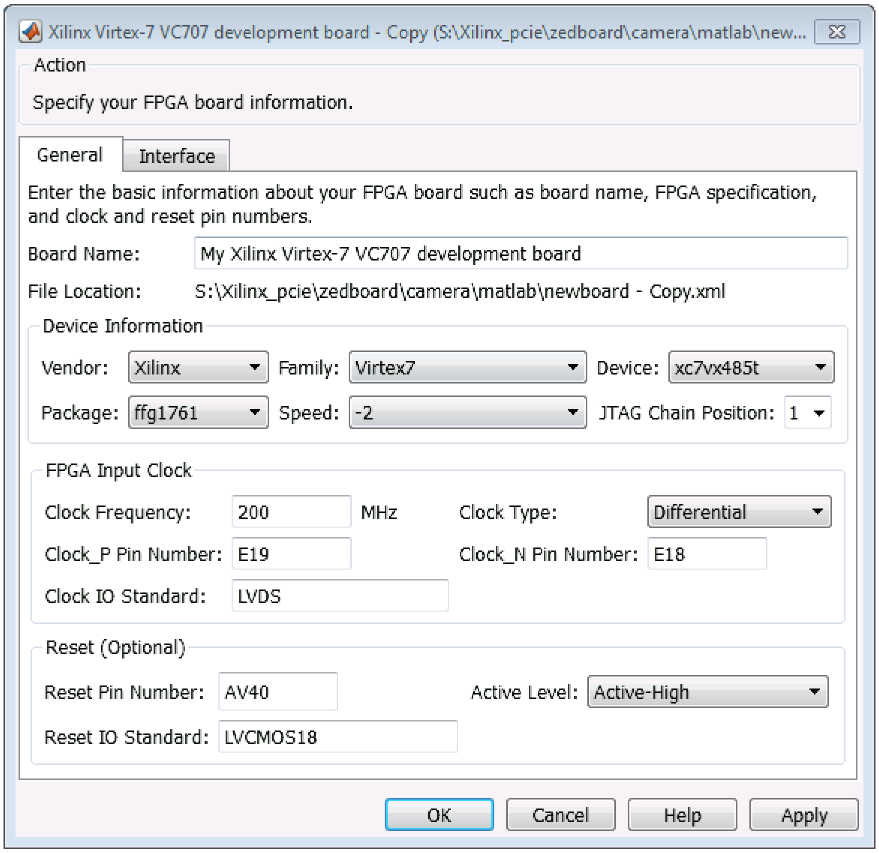
When you have completed validation, click **Finish**. See "Save Board Definition File" on page 24-17.

FPGA Board Editor

To edit a board definition XML file, first make it writeable. If the file is read-only, the FPGA Board Editor only lets you view the board configuration information. You cannot modify that information.

In this section...
"General Tab" on page 24-38
"Interface Tab" on page 24-40

General Tab



Xilinx Virtex-7 VC707 development board - Copy (S:\Xilinx_pcie\zedboard\camera\matlab\new...

Action

Specify your FPGA board information.

Enter the basic information about your FPGA board such as board name, FPGA specification, and clock and reset pin numbers.

Board Name: My Xilinx Virtex-7 VC707 development board

File Location: S:\Xilinx_pcie\zedboard\camera\matlab\newboard - Copy.xml

Device Information

Vendor: Xilinx
Family: Virtex7
Device: xc7vx485t
Package: ffg1761
Speed: -2
JTAG Chain Position: 1

FPGA Input Clock

Clock Frequency: 200 MHz
Clock Type: Differential
Clock_P Pin Number: E19
Clock_N Pin Number: E18
Clock IO Standard: LVDS

Reset (Optional)

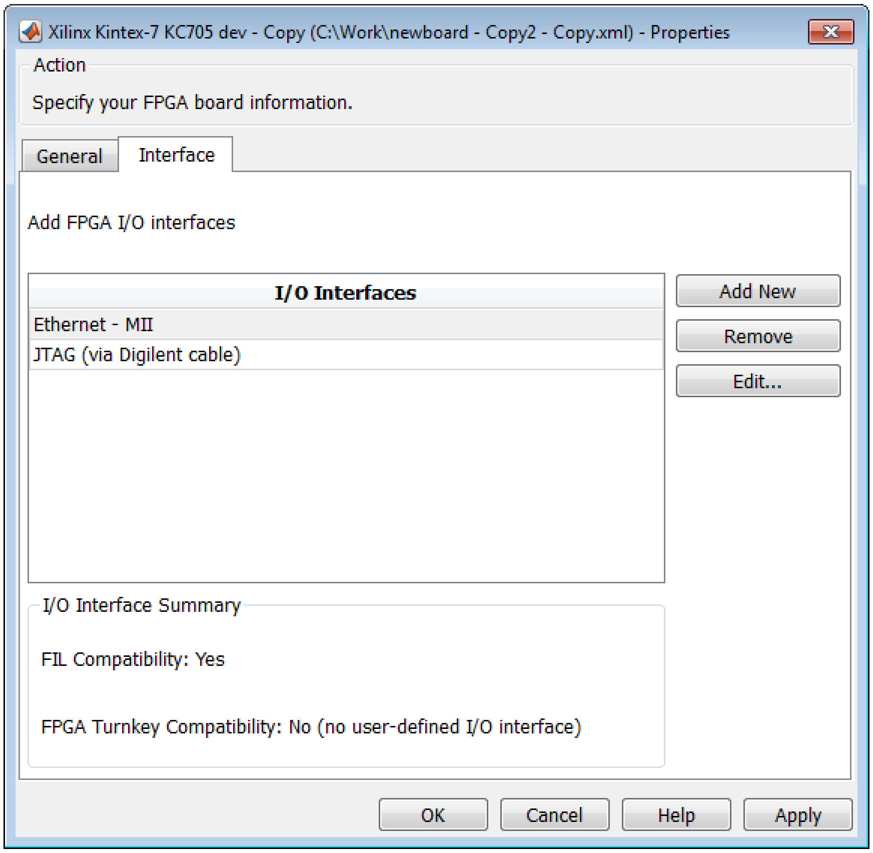
Reset Pin Number: AV40
Active Level: Active-High
Reset IO Standard: LVCMOS18

Board Name: Unique board name

Device Information:

- **Vendor**: `Xilinx` or `Altera`
- **Family**: Family depends on the specified vendor. See the board specification file for applicable settings.
- **Device**: Device depends on the specified vendor and family. See the board specification file for applicable settings.
- For Xilinx boards only:
    - **Package**: Package depends on specified vendor, family, and device. See the board specification file for applicable settings.
    - **Speed**: Speed depends on package. See the board specification file for applicable settings.
    - **JTAG Chain Position**: Value indicates the starting position for JTAG chain. Consult the board specification file for this information.
- **FPGA Input Clock**. Clock details are required for both the FIL and Turnkey workflows. You can find all necessary information in the board specification file.
    - **Clock Frequency**. Must be from 5 through 300. For an Ethernet interface, the suggested clock frequencies are 50, 100, 125, and 200 MHz.
    - **Clock Type**: `Single_Ended` or `Differential`.
    - **Clock Pin Number** (Single_Ended) — Must be specified. Example: `N10`.
    - **Clock_P Pin Number** (Differential) — Must be specified. Example: `E19`.
    - **Clock_N Pin Number** (Differential) — Must be specified. Example: `E18`.
    - **Clock IO Standard** — The programmable I/O Standard to use to configure input, output, or bi-directional ports. For example, `LVDS`.
- **Reset (Optional)**. If you want to indicate a reset, find the pin number and active level in the board specification file, and enter that information.
    - **Reset Pin Number**. Leave empty if you do not have one.
    - **Active Level** : `Active-Low` or `Active-High`.
    - **Reset IO Standard** — The programmable I/O Standard to use to configure input, output, or bi-directional ports. For example, `LVCMOS33`.

Interface Tab



Xilinx Kintex-7 KC705 dev - Copy (C:\Work\newboard - Copy2 - Copy.xml) - Properties

Action

Specify your FPGA board information.

General | Interface

Add FPGA I/O interfaces

I/O Interfaces
Ethernet - MII
JTAG (via Digilent cable)

I/O Interface Summary

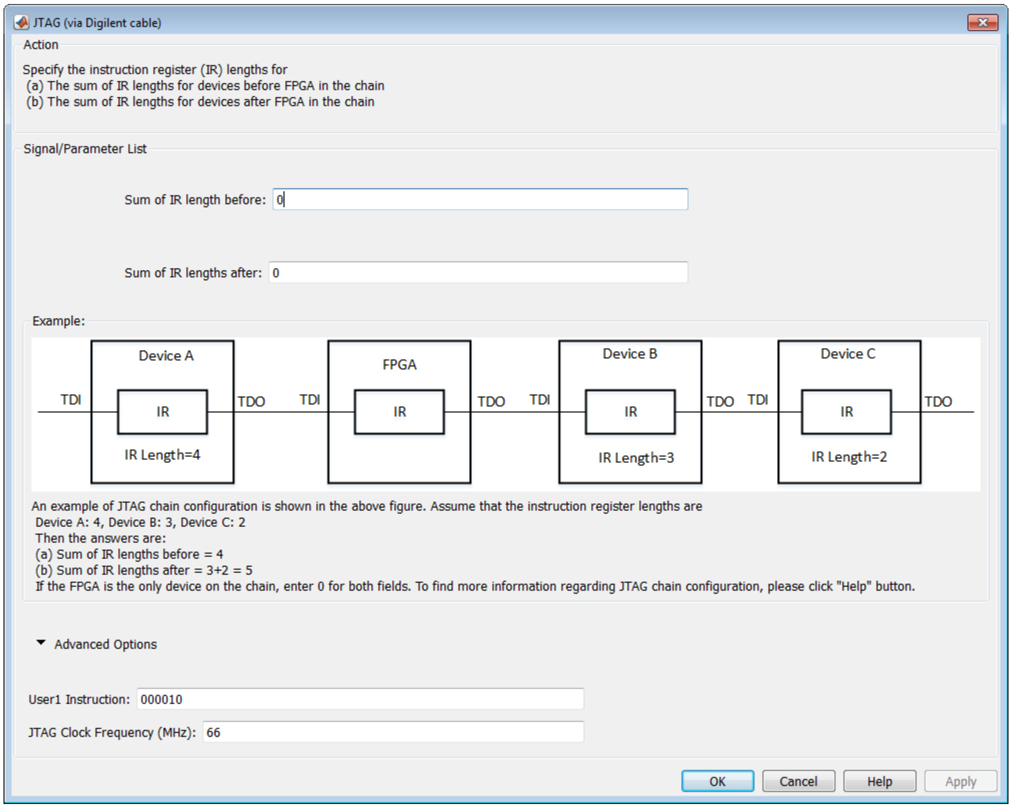
FIL Compatibility: Yes

FPGA Turnkey Compatibility: No (no user-defined I/O interface)

The Interface page describes the supported FPGA I/O Interfaces. Select any listed interface and click **View** to see the **Signal List**. If the board definition file has write permission, you can also **Add New** interface, **Edit** the interface, or **Remove** an interface.

JTAG with Digilent Cable Setup

Note: Enter information for the JTAG cable setup carefully. If the settings are incorrect, the simulation errors out and does not work. If you are still unsure about how to setup your JTAG cable after reading these instructions, contact MathWorks technical support with detailed information about your board.


JTAG (via Digilent cable)

Action

Specify the instruction register (IR) lengths for
(a) The sum of IR lengths for devices before FPGA in the chain
(b) The sum of IR lengths for devices after FPGA in the chain

Signal/Parameter List

Sum of IR length before: 0

Sum of IR lengths after: 0

Example:

An example of JTAG chain configuration is shown in the above figure. Assume that the instruction register lengths are
Device A: 4, Device B: 3, Device C: 2
Then the answers are:
(a) Sum of IR lengths before = 4
(b) Sum of IR lengths after = 3+2 = 5
If the FPGA is the only device on the chain, enter 0 for both fields. To find more information regarding JTAG chain configuration, please click "Help" button.

Advanced Options

User1 Instruction: 000010

JTAG Clock Frequency (MHz): 66

1 **Signal/Parameter List** — Provide the sum of the lengths of the instruction registers (IR) for all devices before and after the FPGA in the chain.

- If the FPGA is the only item in the device chain, use zeros in both **Sum of IR length before** and **Sum of IR length after**.
- If you are using a Zynq device, and it is the only item in the device chain, enter 4 in **Sum of IR length before** and 0 in **Sum of IR length after**.

If your board does not meet either of those conditions, follow these instructions to obtain the IR lengths:

a. Connect the FPGA board to your computer using the JTAG cable. Turn on the board.
b. Make sure that you installed the cable drivers during Vivado installation.
c. Open Vivado Hardware Manager and select **Open a new hardware target**. In the dialog box is a summary of the IR lengths for all devices for that target.
d. Sum the IR lengths before the FPGA and enter the total in **Sum of IR length before**. Sum the IR lengths after the FPGA and enter the total in **Sum of IR length after**.

Vivado Hardware Manager cannot recognize the IR length of less common devices. For these devices, consult the device manual for instruction register length.

2. **Advanced Options** — If the default values are not the same as the most common settings for many devices, set the **User1 Instruction** and **JTAG Clock Frequency (MHz)** parameters. The most common settings are `000010` and `66`, respectively.

   - **User1 Instruction** — The JTAG USER1 Instruction defined in the Xilinx Bscane2 primitive. This binary instruction number, defined by Xilinx, varies from device to device. For most of the 7-series devices, this instruction is `000010`. If your device has a different value, enter it in this parameter.

     To find this value, look at the `bsd` file for your specific device, found in your Vivado installation. For example, for the XA7A32T-CPG236 device, the `bsd` file is located in `Vivado\2014.2\data\parts\xilinx\artix7\aartix7\xa7a35t\cpg236`.

     Open this file. The `USER1` value is `000010`. Enter this value at **User1 Instruction**.

     ```
     "USER1  (000010),"
     ```

   - **JTAG Clock Frequency (MHz)** — Clock frequency used by the JTAG circuit. This value varies by device. You can find this value in the same `bsd` file described under **User1 Instruction**. For example, the JTAG clock frequency is 66 MHz for device XA7A32T-CPG236:

     ```
     attribute TAP_SCAN_CLOCK of TCK : signal is (66.0e6, BOTH);
     ```

HDL Workflow Advisor Tasks

HDL Workflow Advisor Tasks

In this section...

In this section...

HDL Workflow Advisor Tasks Overview

The HDL Workflow Advisor is a tool that supports a suite of tasks covering the stages of the FPGA design process. Some tasks perform model validation or checking; others run the HDL code generator or third-party tools. Each folder at the top level of the HDL Workflow Advisor contains a group of related tasks that you can select and run. To learn about how to run tasks in the HDL Workflow Advisor, see "Open and Run Tasks in the HDL Workflow Advisor" on page 22-2.

- **Set Target**: The tasks in this category enable you to select the desired target device and map its I/O interface to the inputs and outputs of your model.

  Prepare Model For HDL Code Generation: The tasks in this category check your model for HDL code generation compatibility. The tasks also report on model settings, blocks, or other conditions (such as algebraic loops) that would impede code generation, and provide advice on how to fix such problems.
- **HDL Code Generation**: This category supports all HDL-related options of the Configuration Parameters dialog, including setting HDL code and test bench generation parameters, and generating code, test bench, or a cosimulation model.
- **FPGA Synthesis and Analysis**: The tasks in this category support:
  - Synthesis and timing analysis through integration with third-party synthesis tools
  - Back annotation of the model with critical path and other information obtained during synthesis
- **FPGA-in-the-Loop Implementation**: This category implements the phases of FIL, including providing block generation, synthesis, logical mapping, PAR (place-and-route), programming file generation, and a communications channel. These capabilities are designed for a particular board and tailored to your RTL code. An HDL Verifier license is required for FIL.
- **Download to Target**: The tasks in this category depend on the selected target device and potentially include:
  - Generation of a target-specific FPGA programming file
  - Programming the target device
  - Generation of a model that contains a Simulink Real-Time interface subsystem

See Also

For summary information on each HDL Workflow Advisor folder or task, select the folder or task icon and then click the HDL Workflow Advisor **Help** button.

Set Target Overview

The tasks in the **Set Target** folder enable you to select a target FPGA device and define the interface generated for the device.

- **Set Target Device and Synthesis Tool**: Select a target FPGA device and synthesis tools.
- **Set Target Reference Design**: For `IP Core Generation` workflow, select a reference design for your target device.
- **Set Target Interface**: For `IP Core Generation`, `FPGA Turnkey`, and `Simulink Real—Time FPGA I/O` workflows, use the Target Platform Interface Table to assign each port on your DUT to an I/O resource on the target device.
- **Set Target Frequency**: Select the target clock rate for the FPGA implementation of your design.

See Also

For summary information on each **Set Target** task, select the task icon and then click the HDL Workflow Advisor **Help** button.

Set Target Device and Synthesis Tool

The **Set Target Device and Synthesis Tool** task enables you to select an FPGA target device and an associated synthesis tool from a pulldown menu that lists the devices that HDL Workflow Advisor currently supports.

Description

This task displays the following options:

- **Target Workflow**: A pulldown menu that lists the possible workflows that HDL Workflow Advisor supports. Choose from:
  - Generic ASIC/FPGA
  - FPGA-in-the-loop
  - FPGA Turnkey
  - Simulink Real-Time FPGA I/O
  - IP Core Generation
  - Customization for the USRP(R) device
  - Software Defined Radio
- **Target platform**: A pulldown menu that lists the devices the HDL Workflow Advisor currently supports. Not available for the Generic ASIC/FPGA workflow.
- **Synthesis tool**: Select a synthesis tool, then select the **Family**, **Device**, **Package**, and **Speed** for your synthesis target.

  If your synthesis tool is not one of the **Synthesis tool** options, see “Synthesis Tool Path Setup”. After you set up your synthesis tool path, click **Refresh** to make the tool available in the HDL Workflow Advisor.
- **Project folder**: Specify the project folder name.
- **Tool version**: This check box displays the current synthesis tool version.

Set Target Reference Design

The **Set Target Reference Design** task displays the reference design input parameters and the tool version. A **Reference design parameters** section displays any custom parameters that you specify for the reference design.

Description

The task displays the following options:

- **Reference design**: A pulldown menu that lists the reference designs that HDL Coder supports and any custom reference designs that you specify. To learn more about creating a custom board and reference design, see “Board and Reference Design Registration System” on page 27-17.
- **Reference design tool version**: A text box that displays the current reference design tool version. It is recommended to use a reference design tool version that is compatible with the supported tool version. If there is a tool version mismatch, HDL Coder generates an error when you run this task. The tool version mismatch can potentially cause the **Create Project** task to fail.

  If you select the **Ignore tool version mismatch** check box, HDL Coder generates a warning instead of an error. You can attempt to continue with creating the reference design project.
- **Reference design parameters**: Lists the parameters of the reference design. These can be parameters available with the default reference designs that HDL Coder supports, or parameters that you define for your custom reference design. For more information, see “Define Custom Parameters and Callback Functions for Custom Reference Design” on page 27-28.

Set Target Interface

The **Set Target Interface** task displays properties of input and output ports on your DUT, and enables you to map these ports to I/O resources on the target device.

Description

Set Target Interface displays the Target Platform Interface Table, which shows:

- The name, port type (input or output), and data type for each port on your DUT.
- A pulldown menu listing the available I/O resources for the target device.

  These resources are device-specific. For detailed information on each resource, see the documentation for your FPGA development board.

Set Target Frequency

Specify the target frequency for these workflows:

- `Generic ASIC/FPGA`: To specify the target frequency that you want your design to achieve. HDL Coder generates a timing constraint file for that clock frequency and adds the constraint to the FPGA synthesis tool project that you create in the **Create Project** task. If the target frequency is not achievable, the synthesis tool generates an error.
- `IP Core Generation`: To specify the target frequency for HDL Coder to modify the clock module setting in the reference design to produce the clock signal with that frequency. Enter a target frequency value that is within the **Frequency Range (MHz)**. If you do not specify the target frequency, HDL Coder uses the **Default (MHz)** target frequency.
- `Simulink Real-Time FPGA I/O`: For Speedgoat boards that are supported with `Xilinx ISE`, specify the target frequency to generate the clock module to produce the clock signal with that frequency.

  The Speedgoat boards that are supported with `Xilinx Vivado` use the `IP Core Generation` workflow infrastructure. Specify the target frequency for HDL Coder to modify the clock module setting in the reference design to produce the clock signal with that frequency. Enter a target frequency value that is within the **Frequency Range (MHz)**. If you do not specify the target frequency, HDL Coder uses the **Default (MHz)** target frequency.
- `FPGA Turnkey`: To generate the clock module to produce the clock signal with that frequency automatically.

See also “Target Frequency” on page 11-102.

Set Target Interface

Select a processor-FPGA synchronization mode, and map your DUT input and output ports to I/O resources on the target device.

Description

For **Processor/FPGA synchronization**, select:

- **Free running** if you do not want your processor and FPGA to be automatically synchronized.
- **Coprocessing – blocking** if you want HDL Coder to generate synchronization logic for the FPGA automatically, so that the processor and FPGA run in tandem. Select this mode when FPGA execution time is short relative to the processor sample time, and you want the FPGA to complete before the processor continues.
- **Coprocessing – nonblocking with delay** (not supported for `IP Core Generation` workflow) if you want HDL Coder to generate synchronization logic for the FPGA automatically, so that the processor and FPGA run in tandem. Select this mode when the FPGA processing time is long relative to the processor sample time, or you do not want the processor to wait for the FPGA to finish before the processor continues.

This setting is saved with the model as the `ProcessorFPGASynchronization` HDL block property for the DUT block.

The Target Platform Interface Table shows:

- The name, port type (input or output), and data type for each port on your DUT.
- A pulldown menu listing the available I/O resources for the target device.

  These resources are device-specific. For detailed information on each resource, see the documentation for your FPGA development board.

See Also

- “Processor and FPGA Synchronization” on page 26-16
- “Custom IP Core Generation” on page 26-2
- “FPGA Programming and Configuration” on page 27-33

Set Target Interface

Select a processor-FPGA synchronization mode, and map your DUT input and output ports to I/O resources on the target device. Optionally, specify a reference design.

Description

Reference design: Select the predefined embedded system integration project into which HDL Coder inserts your generated IP core.

Reference design path: Enter the path to your downloaded reference design components. This field is available only if the specified **Reference design** requires downloadable components.

For **Processor/FPGA synchronization**, select:

- **Free running** if you do not want your processor and FPGA to be automatically synchronized.
- **Coprocessing – blocking** if you want HDL Coder to generate synchronization logic for the FPGA automatically, so that the processor and FPGA run in tandem. Select this mode when FPGA execution time is short relative to the processor sample time, and you want the FPGA to complete before the processor continues.
- **Coprocessing – nonblocking with delay** (not supported for `IP Core Generation` workflow) if you want HDL Coder to generate synchronization logic for the FPGA automatically, so that the processor and FPGA run in tandem. Select this mode when the FPGA processing time is long relative to the processor sample time, or you do not want the processor to wait for the FPGA to finish before the processor continues.

This setting is saved with the model as the `ProcessorFPGASynchronization` HDL block property for the DUT block.

The Target Platform Interface Table shows:

- The name, port type (input or output), and data type for each port on your DUT.
- A dropdown menu listing the available I/O resources for the target device.

  These resources are device-specific. For detailed information on each resource, see the documentation for your FPGA development board.

See Also

- “Processor and FPGA Synchronization” on page 26-16
- “Custom IP Core Generation” on page 26-2
- “FPGA Programming and Configuration” on page 27-33

Prepare Model For HDL Code Generation Overview

The tasks in the **Prepare Model For HDL Code Generation** folder check the model for compatibility with HDL code generation. If a check encounters a condition that would raise a code generation warning or error, the right pane of the HDL Workflow Advisor displays information about the condition and how to fix it. The **Prepare Model For HDL Code Generation** folder contains the following checks:

- **Check Global Settings**: Check model parameters for compatibility with HDL code generation.
- **Check Algebraic Loops**: Check the model for algebraic loops.
- **Check Block Compatibility**: Check that blocks in the model support HDL code generation.
- **Check Sample Times**: Check the solver options, tasking mode, and rate transition diagnostic settings, given the model's sample times.
- **Check FPGA-in-the-Loop Compatibility**: Check model compatibility with FPGA-in-the-loop, specifically:

  - Not allowed: sink/source subsystems, single/double data types, zero sample time
  - Must be present: HDL Verifier license

  This option is available only if you select `FPGA-in-the-Loop` for Target workflow.
- **Check USRP Compatibility**: The model must have two input ports and two output ports of signed 16-bit signals.

  This option is available only if you select `Customization for the USRP(TM) Device` for Target workflow.

See Also

For summary information on each **Prepare Model For HDL Code Generation** task, select the task icon and then click the HDL Workflow Advisor **Help** button.

Check Global Settings

Check Global Settings checks model-wide parameter settings for HDL code generation compatibility.

Description

This check examines the model parameters for compatibility with HDL code generation and flags conditions that would raise an error or a warning during code generation. The HDL Workflow Advisor displays a table with the following information about each condition detected:

- *Block*: Hyperlink to the model configuration dialog box page that contains the error or warning condition
- *Settings*: Name of the model parameter that caused the error or warning condition
- *Current*: Current value of the setting
- *Recommended*: Recommended value of the setting
- *Severity*: Severity level of the warning or error condition. Minimally, you should fix settings that are tagged as `error`.

Tip

To set reported settings to their recommended values, click the **Modify All** button. You can then run the check again and proceed to the next check.

Check Algebraic Loops

Detect algebraic loops in the model.

Description

TheHDL Coder software does not support HDL code generation for models in which algebraic loop conditions exist. **Check Algebraic Loops** examines the model and fails the check if it detects an algebraic loop. Eliminate algebraic loops from your model before proceeding with further HDL Workflow Advisor checks or code generation.

See Also

For information about algebraic loops, see “Algebraic Loops” (Simulink) in the Simulink documentation.

Check Block Compatibility

Check the DUT for unsupported blocks.

Description

Check Block Compatibility checks blocks within the DUT for compatibility with HDL code generation. The check fails if it encounters blocks that HDL Coder does not support. The HDL Workflow Advisor reports incompatible blocks, including the full path to each block.

See Also

See "View HDL-Specific Block Documentation" on page 12-3 for a complete list of supported blocks and their implementations.

Check Sample Times

Check the solver, sample times, and tasking mode settings for the model.

Description

Check Sample Times checks the solver options, sample times, tasking mode, and rate transition diagnostics for HDL code generation compatibility. Solver options that the HDL Coder software requires or recommends are:

- **Type**: Fixed-step. (The coder currently supports variable-step solvers under limited conditions. See `hdlsetup` for details.)
- **Solver**: Discrete (no continuous states). Other fixed-step solvers could be selected, but this option is usually the best one for simulating discrete systems.
- **Tasking mode**: `SingleTasking`. The coder does not currently support models that execute in multitasking mode. Do not set **Tasking mode** to `Auto`.
- **Multitask rate transition** and **Single task rate transition** diagnostic options: set to `Error`.

Check FPGA-In-The-Loop Compatibility

HDL Verifier checks model for compatibility with FPGA-in-the-loop processing.

See Also

For HDL code and model compatibilities with FPGA-in-the-loop processing, see “Prepare DUT For FIL Interface Generation” (HDL Verifier).

HDL Code Generation Overview

The tasks in the **HDL Code Generation** folder enable you to:

- Set and validate HDL code and test bench generation parameters. Most parameters of the **HDL Code Generation** pane of the Configuration Parameters dialog box and the Model Explorer are supported.
- Generate any or all of:
  - RTL code
  - RTL test bench
  - Cosimulation model
  - SystemVerilog DPI test bench

To run the tasks in the **HDL Code Generation** folder automatically, select the folder and click **Run All**.

Tip: After each task in this folder runs, HDL Coder updates the Configuration Parameters dialog box and the Model Explorer.

Set Code Generation Options Overview

The tasks in the **Set Code Generation Options** folder enable you to set and validate HDL code and test bench generation parameters. Each task of the **Set Code Generation Options** folder supports options of the **HDL Code Generation** pane of the Configuration Parameters dialog box and the Model Explorer. The tasks are:

- **Set Basic Options**: Set parameters that affect overall code generation. See "HDL Code Generation Pane: General" on page 11-8 for information on each parameter.
- **Set Advanced Options**: Set parameters that specify detailed characteristics of the generated code, such as HDL element naming and whether certain optimizations apply. See "HDL Code Generation Pane: Global Settings" on page 11-21 for information on each parameter.
- **Set Testbench Options**: Set options that determine characteristics of generated test bench code. See "HDL Code Generation Pane: Test Bench" on page 11-122 for information on each parameter.

To run the tasks in the **Set Code Generation Options** folder automatically, select the folder and click **Run All**.

Set Basic Options

Set parameters that affect overall code generation.

Description

The **Set Basic Options** task sets options that are fundamental to HDL code generation. These options include selecting the DUT and selecting the target language. The basic options are the same as those found in the top-level **HDL Code Generation** pane of the Configuration Parameters dialog box, except that the **Code generation output** group is omitted.

See Also

See also "HDL Code Generation Pane: General" on page 11-8.

Set Advanced Options

Set parameters that specify detailed characteristics of the generated code.

Description

The advanced options are the same as those found in the **HDL Code Generation** > **Global Settings** pane of the Configuration Parameters dialog box and the Model Explorer.

See Also

See also "HDL Code Generation Pane: Global Settings" on page 11-21.

Set Optimization Options

Set parameters that specify optimizations such as resource sharing and pipelining to improve area and timing.

Description

The optimization options are the same as those found in the **HDL Code Generation** > **Target and Optimizations** pane of the Configuration Parameters dialog box and the Model Explorer.

See Also

See also "HDL Code Generation Pane: Target and Optimizations" on page 11-100.

Set Testbench Options

Set options that determine characteristics of generated test bench code.

Description

The test bench options are the same as those found in the **HDL Code Generation** > **Test Bench** pane of the Configuration Parameters dialog box and the Model Explorer.

See Also

See also "HDL Code Generation Pane: Test Bench" on page 11-122.

Generate RTL Code

Generate RTL code and HDL top-level wrapper.

Description

The **Generate RTL Code** task generates RTL code and an HDL top-level wrapper for the DUT subsystem. It also generates a constraint file that contains pin mapping information and clock constraints.

Generate RTL Code and Testbench

Select and initiate generation of RTL code, RTL test bench, and cosimulation model.

Description

The **Generate RTL Code and Testbench** task enables choosing what type of code or model that you want to generate. You can select any combination of the following:

- **Generate RTL code**: Generate RTL code in the target language.
- **Generate testbench**: Generate the test bench(es) selected in **Set Testbench Options**.
- **Generate validation model**: Generate a validation model that highlights generated delays and other differences between your original model and the generated cosimulation model. With a validation model, you can observe the effects of streaming, resource sharing, and delay balancing.

  The validation model contains the DUT from the original model and the DUT from the generated cosimulation model. 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.

See Also

See also "Generating a Simulink Model for Cosimulation with an HDL Simulator" (Filter Design HDL Coder).

Verify with HDL Cosimulation

Run this step to verify the generated HDL using cosimulation between the HDL Simulator and the Simulink test bench. This step shows only if you selected **Cosimulation model**, and specified an HDL simulator, in **Set Testbench Options**.

Generate RTL Code and IP Core

Select and initiate generation of RTL code and custom IP core.

Description

In the **Generate RTL Code and IP Core** task, specify characteristics of the generated IP core:

- **IP core name**: Enter the IP core name.

  This setting is saved with the model as the `IPCoreName` HDL block property for the DUT block.
- **IP core version**: Enter the IP core version number. HDL Coder appends the version number to the IP core name to generate the output folder name.

  This setting is saved with the model as the `IPCoreVersion` HDL block property for the DUT block.
- **IP core folder** (not editable): HDL Coder generates the IP core files in the output folder shown, including the HTML documentation.
- **IP repository**: If you have an IP repository folder, enter its path manually or by using the **Browse** button. The coder copies the generated IP core into the IP repository folder.
- **Additional source files**: If you are using a black box interface in your design to include existing Verilog or VHDL code, enter the file names. Enter each file name manually, separated with a semicolon (;), or by using the **Add** button. The source file language must match your target language.

  This setting is saved with the model as the `IPCoreAdditionalFiles` HDL block property for the DUT block.
- **Generate IP core report**: Select this option to generate HTML documentation for the IP core.

See Also

- "Custom IP Core Generation" on page 26-2
- "Generate a Board-Independent IP Core from Simulink" on page 26-12
- "Custom IP Core Report" on page 26-5

FPGA Synthesis and Analysis Overview

Create projects for supported FPGA synthesis tools, perform FPGA synthesis, mapping, and place/route tasks, and annotate critical paths in the original model

Description

The tasks in the **FPGA Synthesis and Analysis** folder enable you to:

- Create FPGA synthesis projects for supported FPGA synthesis tools.
- Launch supported FPGA synthesis tools, using the project files to perform synthesis, mapping, and place/route tasks.
- Annotate your original model with critical path information obtained from the synthesis tools.

For a list of supported third-party synthesis tools, see “Third-Party Synthesis Tools and Version Support”.

The tasks in the folder are:

- **Create Project**
- **Perform Synthesis and P/R**
- **Annotate Model with Synthesis Result**

See Also

See also “FPGA Synthesis and Analysis” on page 22-50.

Create Project

Create FPGA synthesis project for supported FPGA synthesis tool.

Description

This task creates a synthesis project for the selected synthesis tool and loads the project with the HDL code generated for your model.

When the project creation completes, the HDL Workflow Advisor displays a link to the project in the right pane. Click this link to view the project in the synthesis tool project window.

Synthesis objective

Select a synthesis objective to generate tool-specific optimization Tcl commands for your project. If you specify `None`, no Tcl commands are generated.

To learn how the synthesis objectives map to Tcl commands, see “Synthesis Objective to Tcl Command Mapping” on page 22-63.

Additional source files

Enter additional HDL source files you want included in your synthesis project. Enter each file name manually, separated with a semicolon (;), or by using the **Add Source** button.

For example, you can include HDL source files (.vhd or .v) or a constraint file (.ucf or .sdc).

Additional project creation Tcl files

Enter additional project creation Tcl files you want to include in your synthesis project. Enter each file name manually, separated with a semicolon (;), or by using the **Add Tcl** button.

For example, you can include a Tcl script (.tcl) to execute after creating the project.

See Also

- “Third-Party Synthesis Tools and Version Support”
- “Creating a Synthesis Project” on page 22-50
- “Synthesis Objective to Tcl Command Mapping” on page 22-63

Perform Synthesis and P/R Overview

Launch supported FPGA synthesis tools to perform synthesis, mapping, and place/route tasks.

Description

The tasks in the **Perform Synthesis and P/R** folder enable you to launch supported FPGA synthesis tool and:

- Synthesize the generated HDL code.
- Perform mapping and timing analysis.
- Perform place and route functions.

For a list of supported third-party synthesis tools, see “Third-Party Synthesis Tools and Version Support”.

See Also

See also “FPGA Synthesis and Analysis” on page 22-50

Perform Logic Synthesis

Launch supported FPGA synthesis tool and synthesize the generated HDL code.

Description

The **Perform Logic Synthesis** task:

- Launches the synthesis tool in the background.
- Opens the previously generated synthesis project, compiles HDL code, synthesizes the design, and emits netlists and related files.
- Displays a synthesis log in the **Result** subpane.

See Also

See also “Performing Synthesis, Mapping, and Place and Route” on page 22-52.

Perform Mapping

Launches supported FPGA synthesis tool and maps the synthesized logic design to the target FPGA.

Description

The **Perform Mapping** task:

- Launches the synthesis tool in the background.
- Runs a mapping process that maps the synthesized logic design to the target FPGA.
- Emits a circuit description file for use in the place and route phase.
- Also emits pre-routing timing information for use in critical path analysis and back annotation of your source model.
- Displays a log in the **Result** subpane.

Enable **Skip pre-route timing analysis** if your tool does not support early timing estimation. When this option is enabled, the **Annotate Model with Synthesis Result** task sets **Critical path source** to **post-route**.

See Also

See also “Performing Synthesis, Mapping, and Place and Route” on page 22-52.

Perform Place and Route

Launches the synthesis tool in the background and runs a Place and Route process.

Description

The **Perform Place and Route** task:

- Launches the synthesis tool in the background.
- Runs a Place and Route process that takes the circuit description produced by the previous mapping process, and emits a circuit description suitable for programming an FPGA.
- Also emits post-routing timing information for use in critical path analysis and back annotation of your source model.
- Displays a log in the **Result** subpane.

Tips

If you select **Skip this task** , the HDL Workflow Advisor executes the workflow, but omits the **Perform Place and Route** task, marking it `Passed`. You might want to select **Skip this task** if you prefer to do place and route work manually.

If **Perform Place and Route** fails, but you want to use the post-mapping timing results to find critical paths in your model, you can select **Ignore place and route errors** and continue to the **Annotate Model with Synthesis Result** task.

See Also

See also "Performing Synthesis, Mapping, and Place and Route" on page 22-52.

Run Synthesis

Launches Xilinx Vivado and executes the Vivado **Synthesis** step.

Enable **Skip pre-route timing analysis** if you do not want to do early timing estimation.

Run Implementation

Launches Xilinx Vivado and executes the Vivado **Implementation** step.

If you select **Skip this task** , the HDL Workflow Advisor omits the **Run Implementation** task, marking it `Passed`. Select **Skip this task** if you prefer to do place and route work manually.

If **Run Implementation** fails, you can select **Ignore place and route errors** and continue to the **Annotate Model with Synthesis Result** task.

Check Timing Report

If there are timing failures during this task, the task does not fail. You must check the timing report for timing failures.

Annotate Model with Synthesis Result

Analyzes pre- or post-routing timing information and visually highlights critical paths in your model

Description

The **Annotate Model with Synthesis Result** task helps you to identify critical paths in your model. At your option, the task analyzes pre- or post-routing timing information produced by the **Perform Synthesis and P/R** task group, and visually highlights one or more critical paths in your model.

If **Generate FPGA top level wrapper** is selected in the **Generate RTL Code and Testbench** task, **Annotate Model with Synthesis Result** is not available. To perform back-annotation analysis, clear the check box for **Generate FPGA top level wrapper**.

Input Parameters

Critical path source

Select **pre-route** or **post-route**.

The **pre-route** option is unavailable when **Skip pre-route timing analysis** is enabled in the previous task group.

Critical path number

You can annotate up to 3 critical paths. Select the number of paths you want to annotate.

Show all paths

Show critical paths, including duplicate paths.

Show unique paths

Show only the first instance of a path that is duplicated.

Show delay data

Annotate the cumulative timing delay on each path.

Show ends only

Show the endpoints of each path, but omit the connecting signal lines.

Results and Recommended Actions

When the **Annotate Model with Synthesis Result** task runs to completion, HDL Coder displays the DUT with critical path information highlighted.

See Also

"Annotating Your Model with Critical Path Information" on page 22-55

Download to Target Overview

The **Download to Target** folder supports the following tasks:

- **Generate Programming File**: Generate an FPGA programming file.
- **Program Target Device**: Download generated programming file to the target development board.
- **Generate Simulink Real-Time Interface** (for Speedgoat target devices only): Generate a model that contains a Simulink Real-Time interface subsystem.

See Also

For summary information on each **Download to Target** task, select the task icon and then click the HDL Workflow Advisor **Help** button.

Generate Programming File

The **Generate Programming File** task generates an FPGA programming file that is compatible with the selected target device.

Program Target Device

The **Program Target Device** task downloads the generated FPGA programming file to the selected target device.

Before executing the **Program Target Device** task, make sure that your host PC is properly connected to the target development board via the required programming cable.

Generate Simulink Real-Time Interface

The **Generate Simulink Real-Time Interface** task generates a model containing an interface subsystem that you can plug in to a Simulink Real-Time model.

The naming convention for the generated model is:

```
gm_fpgamodelname_slrt
```

where `fpgamodelname` is the name of the original model.

Save and Restore HDL Workflow Advisor State

You can 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.

See Also

For detailed information on how to create, save, and load a restore point, see "Save and Restore HDL Workflow Advisor State" on page 22-5.

FPGA-In-The-Loop Implementation

Set FIL options and run FIL processing.

Set FPGA-In-The-Loop Options

Set connection type, board IP, and MAC addresses and select additional files, if required.

Connection

Select either `JTAG` (Altera boards only) or `Ethernet`.

Board IP Address

Use this option for setting the IP address of the board if it is not the default IP address (192.168.0.2).

Board MAC Address

Under most circumstances, you do not need to change the Board MAC address. You will need to do so if you connect more than one FPGA development board to a single computer (for which you must have a separate NIC for each board). You must change the Board MAC address for additional boards so that each address is unique.

Additional Source Files

Select additional source files for the HDL design that is to be verified on the FPGA board, if required. HDL Workflow Advisor attempts to identify the file type; change the file type in the **File Type** column if it is incorrect.

Build FPGA-In-The-Loop

During the build process, the following actions occur:

- FPGA-in-the-loop generates a FIL block named after the top-level module and places it in a new model.
- After new model generation, FIL opens a command window. In this window, the FPGA design software performs synthesis, fit, place-and-route, timing analysis, and FPGA programming file generation. When the process completes, a message in the command prompts you to close the window.
- FPGA-in-the-loop builds a testbench model around the generated FIL block.

Check USRP® Compatibility

The model must have two input ports and two output ports of signed 16-bit signals.

Generate FPGA Implementation

This step initiates FPGA programming file creation. For Input Parameters, enter the path to the Ettus Research™ USRP® FPGA files you previously downloaded. If you have not yet downloaded these files, see the Support Package for USRP® Radio documentation.

When this step completes, see the instructions for downloading the programming file to the FPGA and running the simulation in the Support Package for USRP® Radio documentation for FPGA Targeting.

Check SDR Compatibility

The DUT must adhere to certain signal interface requirements. During Check SDR Compatibility, the following interface checks are performed (Inputs and Outputs go through the same checks).

- Must include single complex signal, two scalar signals, or single vectored signal of size 2
- Must have a bitwidth of 16
- Must be signed

- Must be single rate
- If have vectored ports must use Scalarize Vectors option
- If have multiple rates, must use Single clock
- Must use synchronous reset
- Must use active-high reset
- Must use a user overclocking factor of 1

All error checks are done for a given task run and reported in a table. This allows a single iteration to fix all errors.

SDR FPGA Implementation

The SDR FPGA integrates customer logic as generated in previous steps as well as SDR-specific code to provide data and control paths between an RF board and the host.

This step consists of the following tasks:

- Set SDR Options: Choose customization options
- Build SDR: Generate FPGA programming file for an SDR target.

Set SDR Options

Choose customization options for the completion of the SDR FPGA implementation.

SDR FPGA Component Options

- **RF board for target**

  Choose one of the following:

  - `Epic Bitshark FMC-1Rx RevB`
  - `Epic Bitshark FMC-1Rx RevC`
  - `Analog Devices AD FMCOMMS1—ABZ RevB`
- **Folder with vendor HDL source code**

  Specify the folder that contains the RF interface HDL downloaded from the vendor support site. Use **Browse** to navigate to the correct folder.

- **User logic synthesis frequency**

  Specify the maximum frequency at which you want to run your design. This value must be greater than the sampling frequencies for ADC and DAC as specified in the ADI FMCOMMS or Epiq Bitshark™ block.
- **User logic data path**

  Select either the `Receiver data path` or the `Transmitter data path`.

Radio IP Addresses

- **Board IP address**

  Set the board's IP address in this field if it is not the default IP address (192.168.10.1).
- **Board MAC address**

  Under most circumstances, you do not need to change the Board MAC address. However, you need to do so if you connect more than one FPGA development board to a single computer (for which you must have a separate NIC for each board). You must change the Board MAC address for additional boards so that each address is unique.

Additional Source and Project Files for the HDL Design

Specify files you want included in the ISE or Vivado project. You should include only file types supported by ISE or Vivado. If an included file does not exist, the HDL Workflow Advisor cannot create the project.

- **File**: Name of file added to design (with **Add**).
- **File Type**: File type. The software will attempt to determine the file type automatically, but you may override the selection. Options are `VHDL`, `Verilog`, `EDIF netlist`, `VQM netlist`, `QSF file`, `Constraints`, and `Others`.
- **Add**: Add a new file to the list.
- **Remove**: Removes the currently selected file from the list.
- **Up**: Moves the currently selected file up the list.
- **Down**: Moves the currently selected file down the list.

Show full paths to source files (checkbox) triggers a full path display. Leaving this box unchecked displays only the file name.

Build SDR

The HDL Workflow Advisor creates a new Xilinx ISE or Vivado project and adds the following:

- All the necessary files from the FPGA repository
- The generated HDL files for the selected subsystem and algorithm

If no errors are found during FPGA project generation and syntax checking, the FPGA programming file generation process starts. You can view this process in an external command shell and monitor its progress. When the process is finished, a message in the command window prompts you to close the window.

Embedded System Integration

Tasks in this folder integrate your generated HDL IP core with the embedded processor.

Create Project

Create project for embedded system tool.

In the message window, after the project is generated, you can click the project link to open the generated embedded system tool project.

Embedded system tool

Embedded design tool.

Project folder

Folder where your generated project files are saved.

Synthesis objective

Select a synthesis objective to generate tool-specific optimization Tcl commands for your project. If you specify `None`, no Tcl commands are generated.

To learn how the synthesis objectives map to Tcl commands, see “Synthesis Objective to Tcl Command Mapping” on page 22-63.

Generate Software Interface Model

Generate a software interface model with IP core driver blocks for embedded C code generation.

After you generate the software interface model, you can generate C code from it using Embedded Coder®.

Skip this task: Select this option if you want to provide your own embedded C code, or do not have an Embedded Coder license.

Operating system: Select your target operating system.

Build FPGA Bitstream

Generate bitstream for embedded system.

Run build process externally

Enable this option to run the build process in parallel with MATLAB. If this option is disabled, you cannot use MATLAB until the build is finished.

Tcl file for synthesis build

To customize your synthesis build, save your custom Tcl commands in a file and select `Custom`. Enter the file path manually or by using the **Browse** button. The contents of your custom Tcl file are inserted between the Tcl commands that open and close the project.

If you select `Custom` and want to generate a bitstream, the bitstream generation Tcl command must refer to the top file wrapper name and location either directly or implicitly. For example, the following Xilinx Vivado Tcl command generates a bitstream and implicitly refers to the top file name and location:

```
launch_runs impl_1 -to_step write_bitstream
```

Program Target Device

Program the connected target SoC device. Specify the **Programming method** for the target device:

- `JTAG`: This is the default **Programming method**.
- `Ethernet`: If you have Embedded Coder and the Embedded Coder Support Package for Xilinx Zynq or Altera SOC Platform, you can use an Ethernet connection to program the target device.
- `Download`: Copies the generated FPGA bistream, device tree, and system initialization scripts to the SD card on the Zynq board, and keeps the bitstream on the SD card persistently.

To define your own function to program the target device in your custom reference design, you can use the `Custom` **Programming method**. To use the custom programming, register the function handle of the custom programming function using the `CallbackCustomProgrammingMethod` method of the `hdlcoder.ReferenceDesign` class. For example:

```
hRD.CallbackCustomProgrammingMethod = ...
    @parameter_callback.callback_CustomProgrammingMethod;
```

Hardware-Software Codesign

Hardware-Software Co-Design Basics

Custom IP Core Generation

Using the HDL Workflow Advisor, you can generate a custom IP core from a model or algorithm. The generated IP core is sharable and reusable. You can integrate it with a larger design by adding it in an embedded system integration environment, such as Altera Qsys, Xilinx EDK, or Xilinx IP Integrator.

To learn how to generate a custom IP core from Simulink, see "Generate a Board-Independent IP Core from Simulink" on page 26-12.

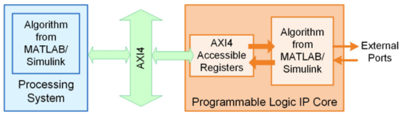
To learn how to generate a custom IP core from a MATLAB design, see "Generate a Board-Independent IP Core from MATLAB" on page 5-50.

In this section...
"Custom IP Core Architectures" on page 26-2
"Target Platform Interfaces" on page 26-3
"Processor/FPGA Synchronization" on page 26-3
"Custom IP Core Generated Files" on page 26-4

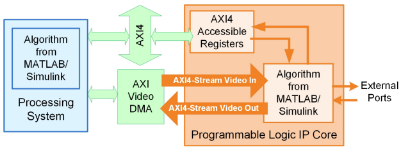
Custom IP Core Architectures

You can generate an IP core with an AXI4 or AXI4-Lite interface. You can also generate an IP core with an AXI4 or AXI4-Lite interface and AXI4-Stream Video interfaces.

An IP core with an AXI4 or AXI4-Lite interface:


Algorithm from MATLAB/ Simulink
Processing System
AXI4
AXI4 Accessible Registers
Algorithm from MATLAB/ Simulink
External Ports
Programmable Logic IP Core

An IP core with an AXI4 or AXI4-Lite interface and AXI4-Stream Video interfaces:


Algorithm from MATLAB/ Simulink
Processing System
AXI4
AXI Video DMA
AXI4 Accessible Registers
AXI4-Stream Video In
AXI4-Stream Video Out
Algorithm from MATLAB/ Simulink
External Ports
Programmable Logic IP Core

The *Algorithm from MATLAB/Simulink* block represents your DUT. HDL Coder generates the rest of the IP core based on your target platform interface settings and processor/FPGA synchronization mode.

Target Platform Interfaces

You can map each port in your DUT to one of the following target platform interfaces in the IP core:

- AXI4-Lite: Use this slave interface to access control registers or for lightweight data transfer. HDL Coder generates memory-mapped registers and allocates address offsets for the ports you map to this interface.
- AXI4: Use this slave interface to connect to components that support burst data transmission. HDL Coder generates memory-mapped registers and allocates address offsets for the ports you map to this interface.
- AXI4-Stream Video: Use this interface to send or receive a 32-bit scalar video data stream.
- External ports: Use external ports to connect to FPGA external IO pins, or to other IP cores with external ports.

To learn more about the AXI4, AXI4-Lite and AXI4-Stream Video protocols, refer to your target hardware documentation.

Processor/FPGA Synchronization

HDL Coder generates synchronization logic in the IP core based on the processor/FPGA synchronization mode you choose.

When generating a custom IP core, the following processor/FPGA synchronization options are available:

- `Free running` (default)
- `Coprocessing — blocking`

To learn more about the processor/FPGA synchronization modes, see “Processor and FPGA Synchronization” on page 26-16.

Custom IP Core Generated Files

After you generate a custom IP core, the IP core files are in the `ipcore` folder within your project folder. In the HDL Workflow Advisor, you can view the IP core folder name in the **IP core folder** field of the **HDL Code Generation** > **Generate RTL Code and IP Core** task.

The IP core folder contains the following generated files:

- IP core definition files.
- HDL source files (.vhd or .v).
- A C header file with the register address map.
- (Optional) An HTML report with instructions for using the core and integrating the IP core in your embedded system project.

Custom IP Core Report

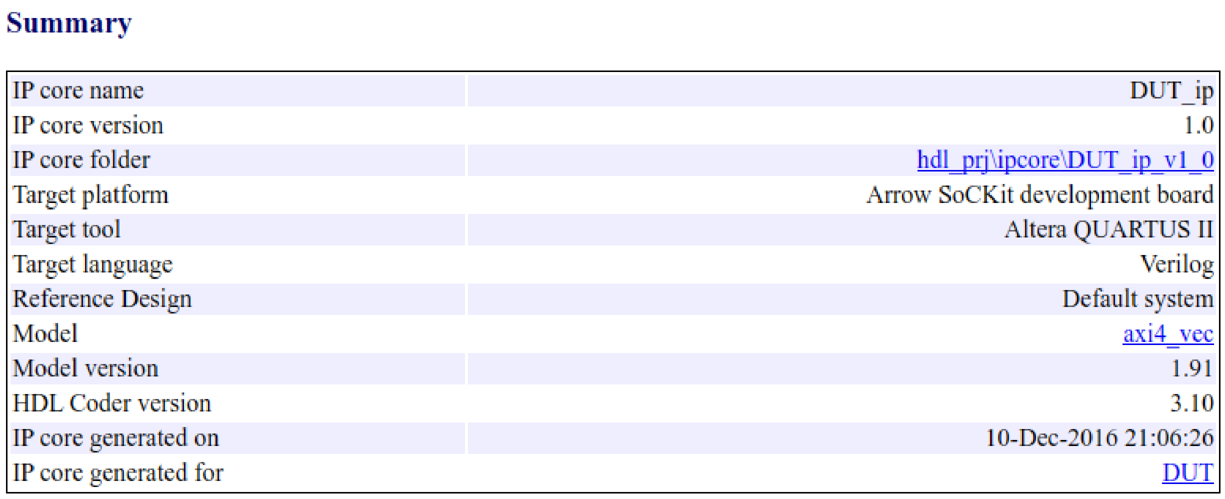
You generate an HTML custom IP core report by default when you generate a custom IP core. The report describes the behavior and contents of the generated custom IP core.

In this section...

Summary

The Summary section shows your coder settings when you generated the custom IP core.

The following figure is an example of a Summary section.



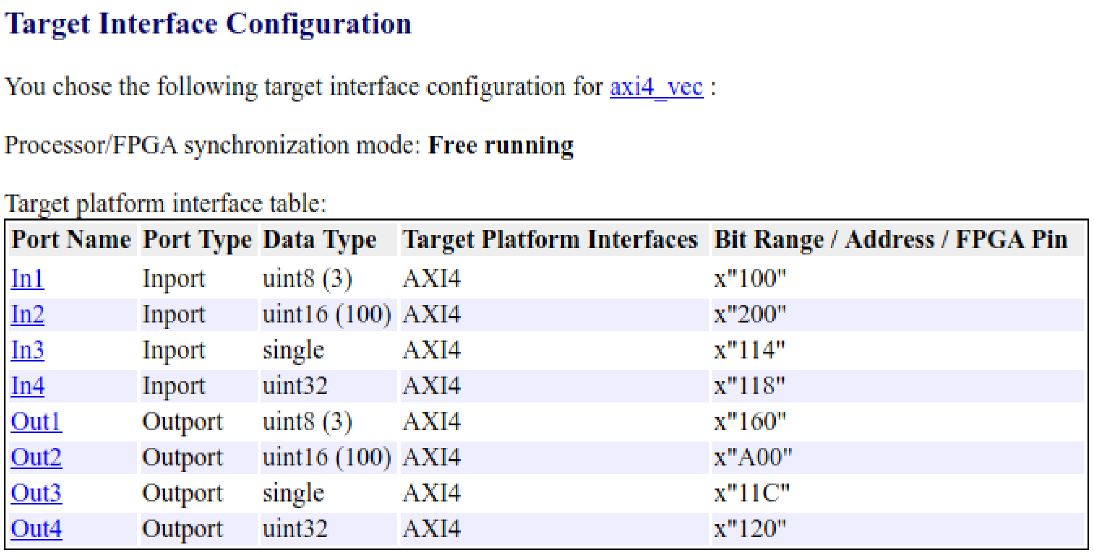
Summary

IP core name	DUT_ip
IP core version	1.0
IP core folder	hdl_prj\ipcore\DUT_ip_v1_0
Target platform	Arrow SoCKit development board
Target tool	Altera QUARTUS II
Target language	Verilog
Reference Design	Default system
Model	axi4_vec
Model version	1.91
HDL Coder version	3.10
IP core generated on	10-Dec-2016 21:06:26
IP core generated for	DUT

Target Interface Configuration

The Target Interface Configuration section shows how your DUT ports map to the target hardware interface and the processor/FPGA synchronization mode.

The following figure is an example of a Target Interface Configuration section.



Target Interface Configuration

You chose the following target interface configuration for axi4_vec :

Processor/FPGA synchronization mode: **Free running**

Target platform interface table:

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
In1	Inport	uint8 (3)	AXI4	x"100"
In2	Inport	uint16 (100)	AXI4	x"200"
In3	Inport	single	AXI4	x"114"
In4	Inport	uint32	AXI4	x"118"
Out1	Outport	uint8 (3)	AXI4	x"160"
Out2	Outport	uint16 (100)	AXI4	x"A00"
Out3	Outport	single	AXI4	x"11C"
Out4	Outport	uint32	AXI4	x"120"

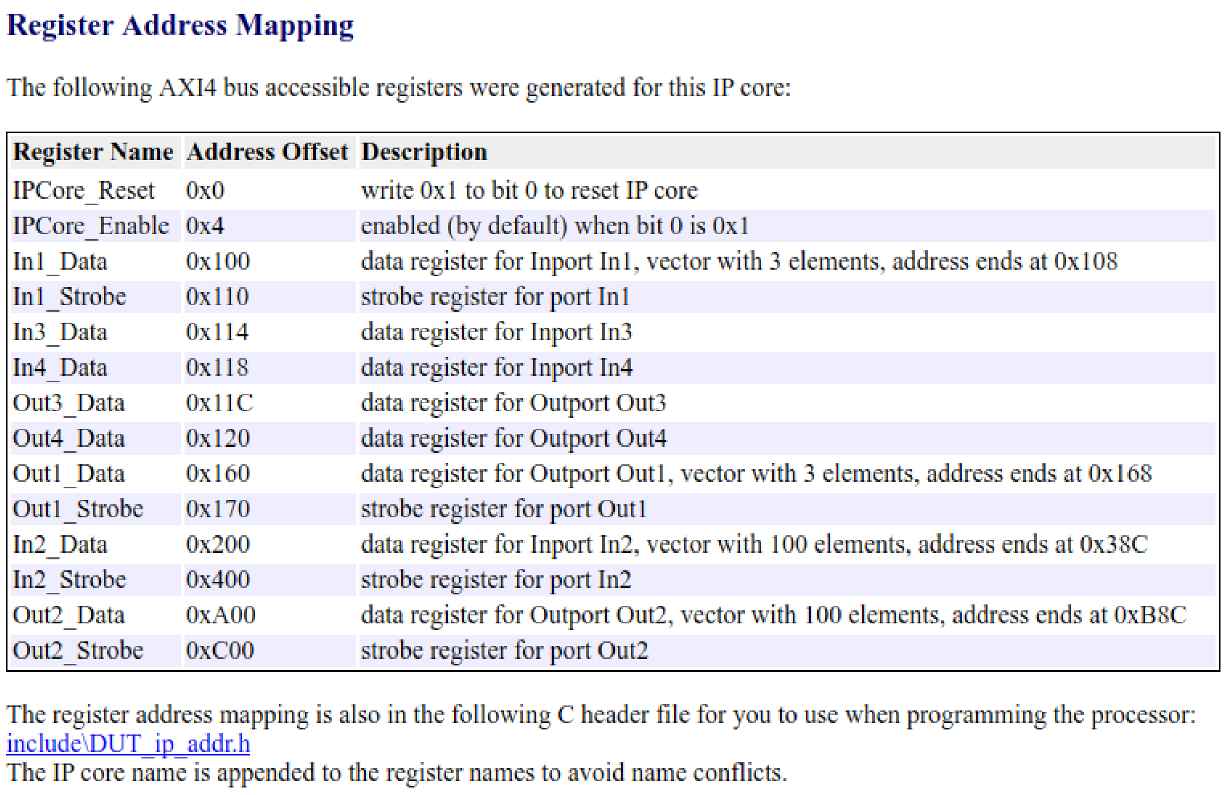
To learn more about processor/FPGA synchronization modes, see “Processor and FPGA Synchronization” on page 26-16.

To learn more about target platform interfaces, see “Custom IP Core Generation” on page 26-2.

Register Address Mapping

The Register Address Mapping section shows the address offsets for AXI4-Lite bus accessible registers in your custom IP core, and the name of the C header file that contains the same address offsets.

The following figure is an example of a Register Address Mapping section.



Register Address Mapping

The following AXI4 bus accessible registers were generated for this IP core:

Register Name	Address Offset	Description
IPCore_Reset	0x0	write 0x1 to bit 0 to reset IP core
IPCore_Enable	0x4	enabled (by default) when bit 0 is 0x1
In1_Data	0x100	data register for Inport In1, vector with 3 elements, address ends at 0x108
In1_Strobe	0x110	strobe register for port In1
In3_Data	0x114	data register for Inport In3
In4_Data	0x118	data register for Inport In4
Out3_Data	0x11C	data register for Outport Out3
Out4_Data	0x120	data register for Outport Out4
Out1_Data	0x160	data register for Outport Out1, vector with 3 elements, address ends at 0x168
Out1_Strobe	0x170	strobe register for port Out1
In2_Data	0x200	data register for Inport In2, vector with 100 elements, address ends at 0x38C
In2_Strobe	0x400	strobe register for port In2
Out2_Data	0xA00	data register for Outport Out2, vector with 100 elements, address ends at 0xB8C
Out2_Strobe	0xC00	strobe register for port Out2

The register address mapping is also in the following C header file for you to use when programming the processor:
include\DUT_ip_addr.h
The IP core name is appended to the register names to avoid name conflicts.

IP Core User Guide

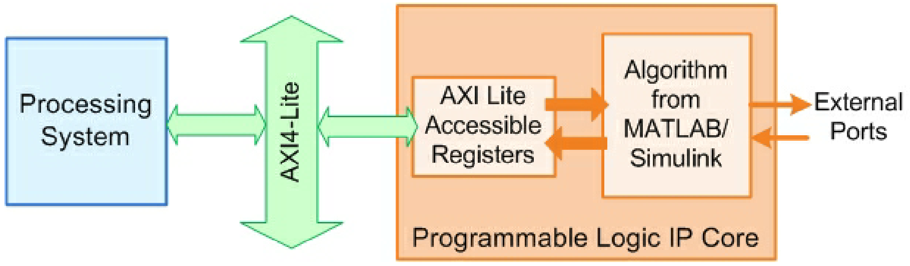
The IP Core User Guide section gives a high-level overview of the system architecture, describes the processor and FPGA synchronization mode, and gives instructions for integrating the IP core in your embedded system integration environment.

The following figure is an example of an IP Core User Guide system architecture description.

Theory of Operation

This IP core is designed to be connected to an embedded processor with an **AXI4-Lite bus.** The processor acts as bus master, and the IP core acts as slave. By accessing the generated registers via the AXI4-Lite bus, the processor can control the IP core, and read and write data from and to the IP core.

For example, to reset the IP core, write 0x1 to the bit 0 of IPCore_Reset register. To enable or disable the IP core, write 0x1 or 0x0 to the IPCore_Enable register. To access the data ports of the MATLAB/Simulink algorithm, read or write to the associated data registers.

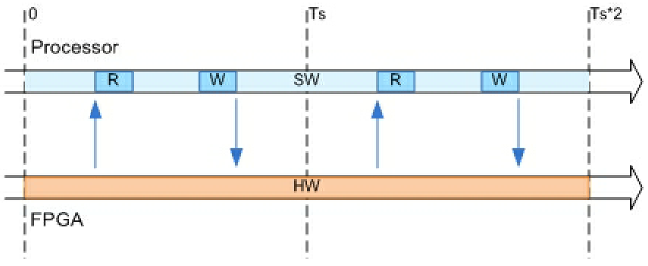

Processing System
AXI4-Lite
AXI Lite Accessible Registers
Algorithm from MATLAB/ Simulink
External Ports
Programmable Logic IP Core

This IP core also support the **External Port** interface. To connect the external ports to the FPGA external IO pins, add FPGA pin assignment constraints in the Xilinx EDK environment.

The following figure is an example of a processor/FPGA synchronization description.

Processor/FPGA Synchronization

The **Free running** mode means there is no explicit synchronization between embedded processor software execution (SW) and the IP core (HW). SW and HW runs independently. The data written from the processor to IP core takes effect immediately, and the data read from the IP core is the latest data available on the IP core output ports.

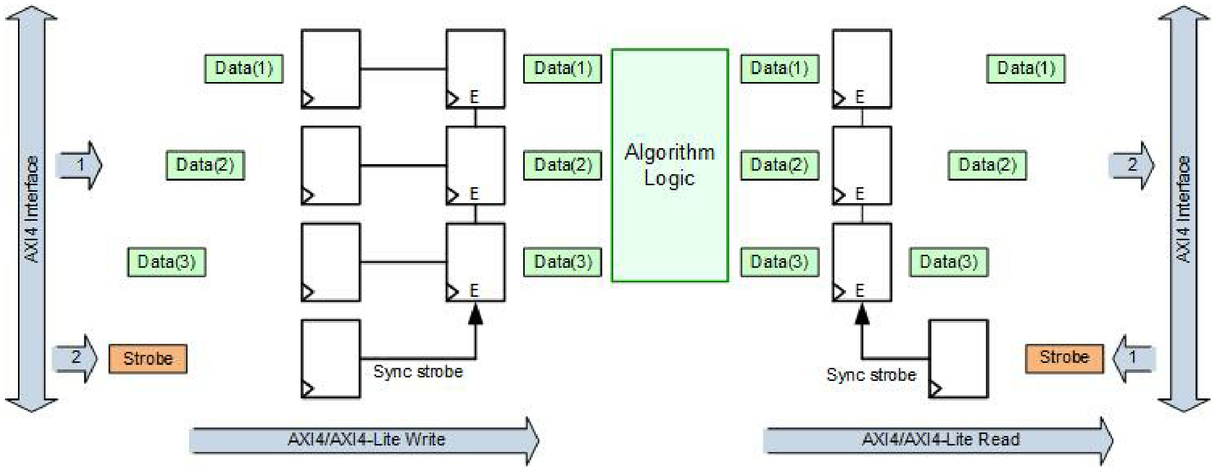


If you use vector data signals at the DUT interface, the IP core report displays this section that shows how the code generator synchronizes vector data across the AXI4 interface.

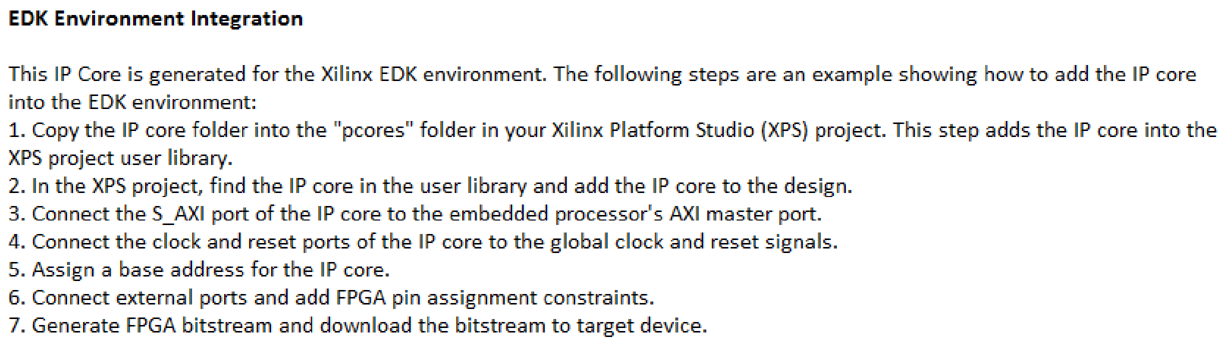
Vector Data Read/Write with Strobe Synchronization

All the elements of vector data are treated as synchronous to the IP core algorithm logic. Additional strobe registers added for each vector input and output port maintain this synchronization across multiple sequential AXI4 reads/writes. For input ports, the strobe register controls the enables on a set of shadow registers, allowing the IP core logic to see all the updated vector elements simultaneously. For output ports, the strobe register controls the synchronous capturing of vector data to be read.

To read a vector data port, first write the strobe address with 0x1, then read each desired data element from corresponding address range. To write a vector data port, first write each desired data element, then write 0x1 to the strobe address to complete the transaction.



The following figure is an example of instructions for integrating the IP core into your embedded system integration environment on the Xilinx platform. If you are targeting an Altera platform, the report displays similar instructions for integrating the IP core into the Altera Qsys environment.



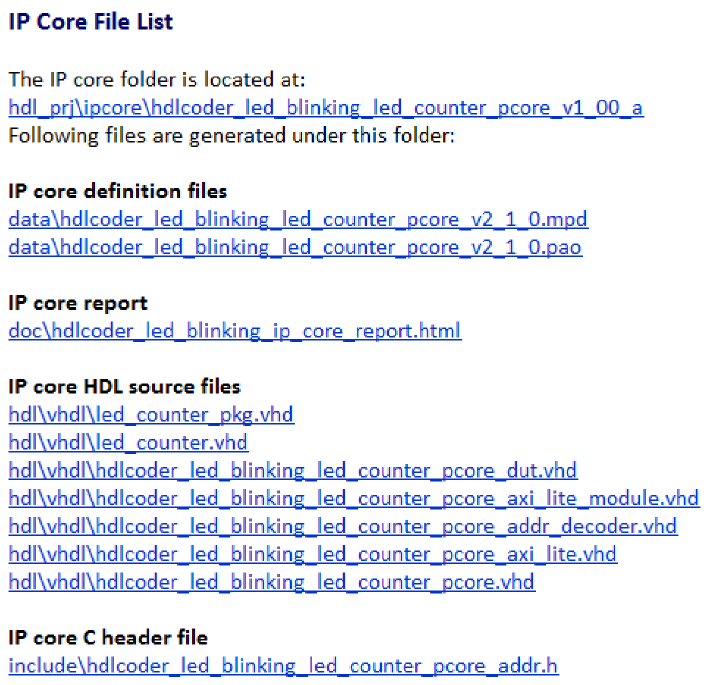
EDK Environment Integration

This IP Core is generated for the Xilinx EDK environment. The following steps are an example showing how to add the IP core into the EDK environment:
1. Copy the IP core folder into the "pcores" folder in your Xilinx Platform Studio (XPS) project. This step adds the IP core into the XPS project user library.
2. In the XPS project, find the IP core in the user library and add the IP core to the design.
3. Connect the S_AXI port of the IP core to the embedded processor's AXI master port.
4. Connect the clock and reset ports of the IP core to the global clock and reset signals.
5. Assign a base address for the IP core.
6. Connect external ports and add FPGA pin assignment constraints.
7. Generate FPGA bitstream and download the bitstream to target device.

IP Core File List

The IP Core File List section lists the files and file folders that comprise your custom IP core.

The following figure is an example of an IP core file list.



IP Core File List

The IP core folder is located at:
hdl_prj\ipcore\hdlcoder_led_blinking_led_counter_pcore_v1_00_a
Following files are generated under this folder:

IP core definition files
data\hdlcoder_led_blinking_led_counter_pcore_v2_1_0.mpd
data\hdlcoder_led_blinking_led_counter_pcore_v2_1_0.pao

IP core report
doc\hdlcoder_led_blinking_ip_core_report.html

IP core HDL source files
hdl\vhdl\led_counter_pkg.vhd
hdl\vhdl\led_counter.vhd
hdl\vhdl\hdlcoder_led_blinking_led_counter_pcore_dut.vhd
hdl\vhdl\hdlcoder_led_blinking_led_counter_pcore_axi_lite_module.vhd
hdl\vhdl\hdlcoder_led_blinking_led_counter_pcore_addr_decoder.vhd
hdl\vhdl\hdlcoder_led_blinking_led_counter_pcore_axi_lite.vhd
hdl\vhdl\hdlcoder_led_blinking_led_counter_pcore.vhd

IP core C header file
include\hdlcoder_led_blinking_led_counter_pcore_addr.h

More About

- “Custom IP Core Generation” on page 26-2
- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 27-2

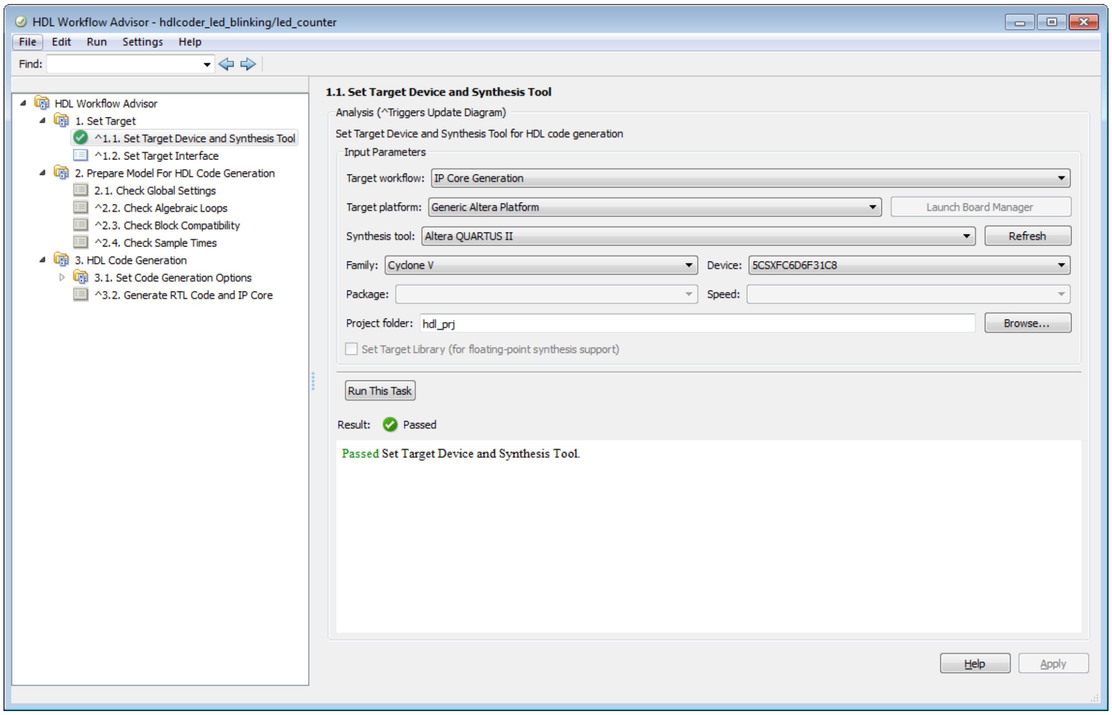
Generate a Board-Independent IP Core from Simulink

In this section...
"Generate a Board-Independent IP Core" on page 26-12
"Requirements and Limitations for IP Core Generation" on page 26-15

Generate a Board-Independent IP Core

To generate a board-independent custom IP core to use in an embedded system integration environment, such as Altera Qsys, Xilinx EDK, or Xilinx IP Integrator:

1. Select your DUT in your Simulink model and open the HDL Workflow Advisor.
2. In the **Set Target** > **Set Target Device and Synthesis Tool** task, for **Target workflow**, select `IP Core Generation`.
3. For **Target platform**, select `Generic Altera Platform` or `Generic Xilinx Platform` and click **Run This Task**.


HDL Workflow Advisor - hdlcoder_led_blinking/led_counter
File Edit Run Settings Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis Tool
^1.2. Set Target Interface
2. Prepare Model For HDL Code Generation
2.1. Check Global Settings
^2.2. Check Algebraic Loops
^2.3. Check Block Compatibility
^2.4. Check Sample Times
3. HDL Code Generation
3.1. Set Code Generation Options
^3.2. Generate RTL Code and IP Core
Launch Board Manager
Refresh
Browse...
Run This Task
Help
Apply

1.1. Set Target Device and Synthesis Tool

Analysis (^Triggers Update Diagram)

Set Target Device and Synthesis Tool for HDL code generation

Input Parameters

Target workflow: IP Core Generation

Target platform: Generic Altera Platform

Synthesis tool: Altera QUARTUS II

Family: Cyclone V Device: 5CSXFC6D6F31C8

Package: Speed:

Project folder: hdl_prj

Set Target Library (for floating-point synthesis support)

Result: Passed

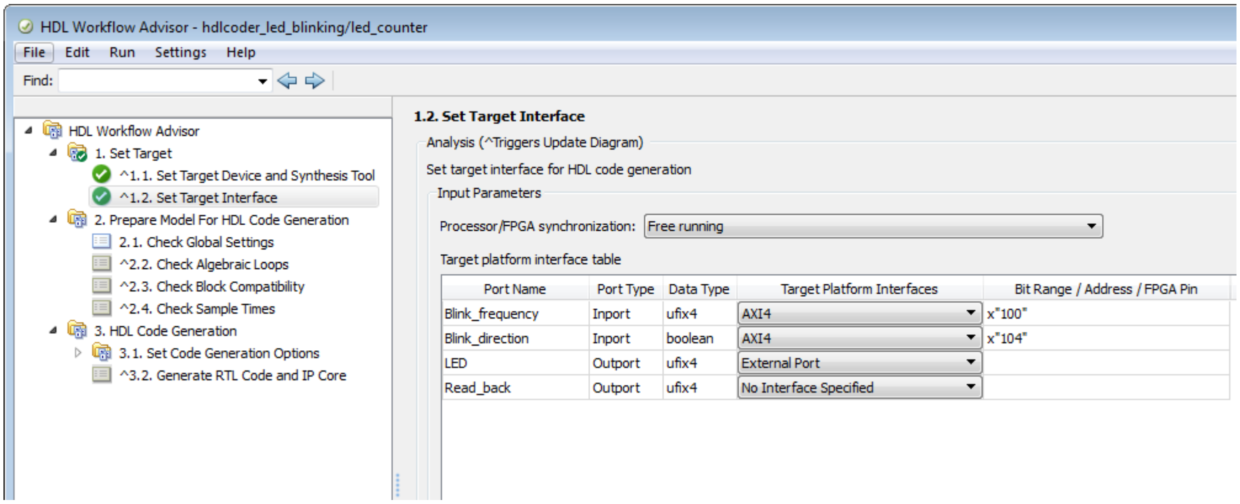
Passed Set Target Device and Synthesis Tool.

4 In the **Set Target > Set Target Interface** task, select a **Target Platform Interface** for each port, then click **Apply**.

You can map each DUT port to one of the following interfaces:

- `AXI4-Lite`: Use this slave interface to access control registers or for lightweight data transfer. HDL Coder generates memory-mapped registers and allocates address offsets for the ports you map to this interface.
- `AXI4`: Use this slave interface to connect to components that support burst data transmission. HDL Coder generates memory-mapped registers and allocates address offsets for the ports you map to this interface.
- `AXI4-Stream`: Use this interface to send or receive a 32-bit scalar data stream.
- `AXI4-Stream Video`: Use this interface to send or receive a 32-bit scalar video data stream.

- `External Port`: Use the external ports to connect to FPGA external IO pins, or to other IP cores with external ports.


HDL Workflow Advisor - hdlcoder_led_blinking/led_counter
File Edit Run Settings Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis Tool
^1.2. Set Target Interface
2. Prepare Model For HDL Code Generation
2.1. Check Global Settings
^2.2. Check Algebraic Loops
^2.3. Check Block Compatibility
^2.4. Check Sample Times
3. HDL Code Generation
3.1. Set Code Generation Options
^3.2. Generate RTL Code and IP Core

1.2. Set Target Interface

Analysis (^Triggers Update Diagram)

Set target interface for HDL code generation

Input Parameters

Processor/FPGA synchronization: Free running

Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
Blink_frequency	Inport	ufix4	AXI4	x"100"
Blink_direction	Inport	boolean	AXI4	x"104"
LED	Outport	ufix4	External Port	
Read_back	Outport	ufix4	No Interface Specified	

5 If you want to set options in the other HDL Workflow Advisor tasks, set them.

6 In the **HDL Code Generation** > **Generate RTL Code and IP Core** task, set the following fields:

- **IP repository**: If you have an IP repository folder, enter its path manually or by using the **Browse** button. The coder copies the generated IP core into the IP repository folder.
- **Additional source files**: If you are using a black box interface in your design to include existing Verilog or VHDL code, enter the file names. Enter each file name manually, separated with a semicolon (**;**), or by using the **Add** button. The source file language must match your target language.
- **Generate IP core report**: Enable this option to generate HTML documentation for the IP core.

7 Right-click the **HDL Code Generation** > **Generate RTL Code and IP Core** task and select `Run to Selected Task`.

HDL Coder generates the IP core files in the output folder shown the **IP core folder** field, including the HTML documentation.

To view the IP core report, click the link in the message window.

To learn more about custom IP core generation, see “Custom IP Core Generation” on page 26-2.

Requirements and Limitations for IP Core Generation

To generate a custom IP core:

- The DUT must be an atomic system.
- There cannot be both an AXI4 interface and AXI4-Lite interface in the same IP core.
- The DUT cannot contain Xilinx System Generator blocks or Altera DSP Builder Advanced blocks.
- If your target language is VHDL, and your synthesis tool is Xilinx ISE or Altera Quartus II, the DUT cannot contain a model reference.

To map your DUT ports to an AXI4-Lite interface, the input and output ports must:

- Have a bit width less than or equal to 32 bits.
- Be scalar.

When mapping your DUT ports to an AXI4-Stream Video interface, the following requirements and limitations apply:

- Ports must have a 32-bit width.
- Ports must be scalar.
- The model must be single rate.
- You can have a maximum of one input video port and one output video port.
- Your synthesis tool must be Xilinx ISE.

The AXI4-Stream Video interface is not supported in **Coprocessing – blocking** processor/FPGA synchronization mode.

Processor and FPGA Synchronization

In the HDL Workflow Advisor, you can choose a **Processor/FPGA synchronization mode** for your processor and FPGA when you:

- Generate a custom IP core to use in an embedded system integration project.
- Use the **Simulink Real-Time FPGA I/O** workflow.

The following synchronization modes are available:

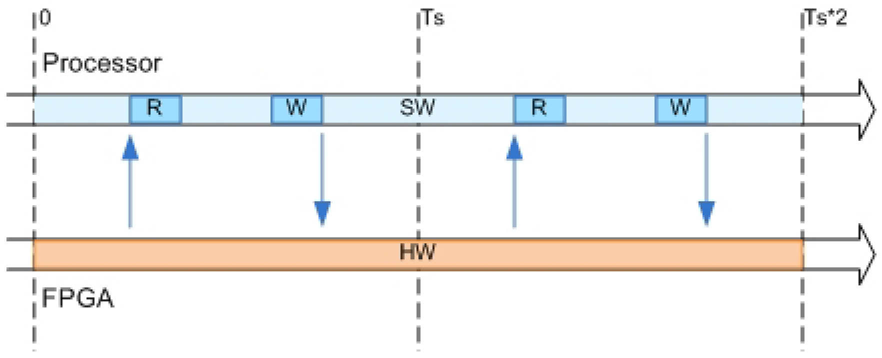
- `Free running` (default)
- `Coprocessing – blocking`
- `Coprocessing – nonblocking with delay` (available only for the **Simulink Real-Time FPGA I/O** workflow)

Free Running Mode

In free running mode, the processor and FPGA each run nonsynchronized, continuously, and in parallel.

Select **Free running** as the **Processor/FPGA synchronization mode** when you do not want your processor and FPGA to be automatically synchronized.

The following diagram shows how the processor and FPGA can communicate in free running mode. The shaded areas indicate that the processor and FPGA are running continuously.

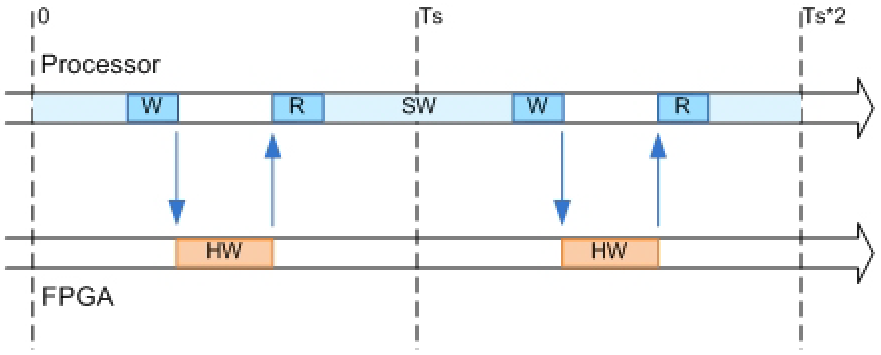


Coprocessing – Blocking Mode

In blocking coprocessor mode, HDL Coder automatically generates synchronization logic for the FPGA so that the processor and FPGA run in tandem.

Select **Coprocessing – blocking** as the **Processor/FPGA synchronization mode** when FPGA execution time is short relative to the processor sample time, and you want the FPGA to complete before the processor continues.

The following diagram shows how the processor and FPGA run in blocking coprocessing mode.



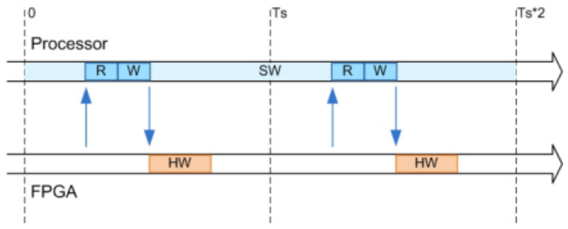
The shaded areas indicate when the processor and FPGA are running. During each sample time, the processor writes to the FPGA, then stops and waits for an indication that the FPGA has finished processing before continuing to run. Each time the FPGA runs, it executes the logic generated for one DUT subsystem sample time.

Coprocessing – Nonblocking With Delay Mode

In delayed nonblocking coprocessor mode, HDL Coder automatically generates synchronization logic for the FPGA so that the processor and FPGA run in tandem. This mode is available only for the **Simulink Real-Time FPGA I/O** workflow.

Select **Coprocessing – nonblocking with delay** as the **Processor/FPGA synchronization mode** when the FPGA processing time is long relative to the processor sample time, or you do not want the processor to wait for the FPGA to finish before the processor continues to run.

The following diagram shows how the processor and FPGA run in delayed nonblocking coprocessor mode.



The shaded areas indicate when the processor and FPGA are running. During each sample time, the processor reads FPGA data from the previous sample time, then writes to the FPGA and continues to run without waiting for the FPGA to finish. Each time the FPGA runs, it executes the logic generated for one DUT subsystem sample time.

IP Caching for Faster Reference Design Synthesis

In this section...

For target platforms that support the `IP Core Generation` workflow with Xilinx Vivado, you can use IP caching. IP caching reduces the synthesis time of reference designs that have many IP modules or that have IP modules with a significant synthesis run time. When you enable IP caching, the Vivado project uses an out-of-context (OOC) workflow. This workflow synthesizes the IP in the reference design out of context from the top-level design. The OOC workflow accelerates project runs because the synthesis tool reuses the IP cache, and does not have to resynthesize the IP when you run the workflow.

If you do not enable IP caching, by default, the Vivado project uses the global synthesis flow. This flow synthesizes the IP modules in the reference design along with the top-level design. In subsequent project runs, this workflow resynthesizes the IP modules in the reference design.

Requirements for Using IP Caching

- **Target workflow**:
  - `IP Core Generation`
  - `Simulink Real-Time FPGA I/O` for Speedgoat boards that use Xilinx Vivado
- **Synthesis tool**: Xilinx Vivado

What Is an IP Cache?

An IP cache is a folder that consists of subfolders corresponding to IP modules in the reference design. Each subfolder is organized by a hash index that corresponds to the file name. For each IP module, the subfolder consists of Xilinx Core Instance (XCI) files, Design Checkpoint (DCP) files, and synthesis log files. The DCP is a container file that contains synthesized netlists, black box HDL stub files, and the output clock constraints.

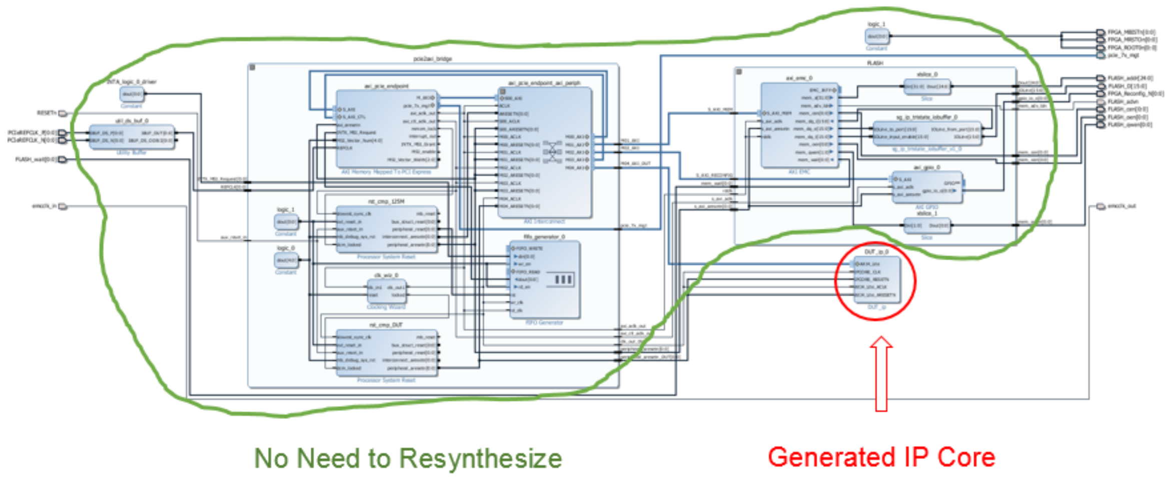
To reuse the IP cache when you run the workflow, the IP synthesis has to match the hash index in the IP cache. The hash index match corresponds to a hit in the IP cache. To hit the IP cache in subsequent runs, use the same:

- Part, language, and target platform settings
- Reference design version
- Target frequency
- `hdl_prj` folder when you created the IP cache

How IP Caching Works

When you enable IP caching, the Xilinx Vivado project uses an out-of-context (OOC) workflow. The OOC design flow is a bottom-up workflow that:

1. Synthesizes the IP modules in the reference design separately from the top-level design. The synthesis output is the Design Checkpoint (DCP) file.
2. Synthesizes your top-level design while treating the IP in the reference design as a black box by using the HDL stub files provided with the DCP.
3. Implements your design on the target device by linking the netlists from the IP design checkpoint files with your top-level netlist.

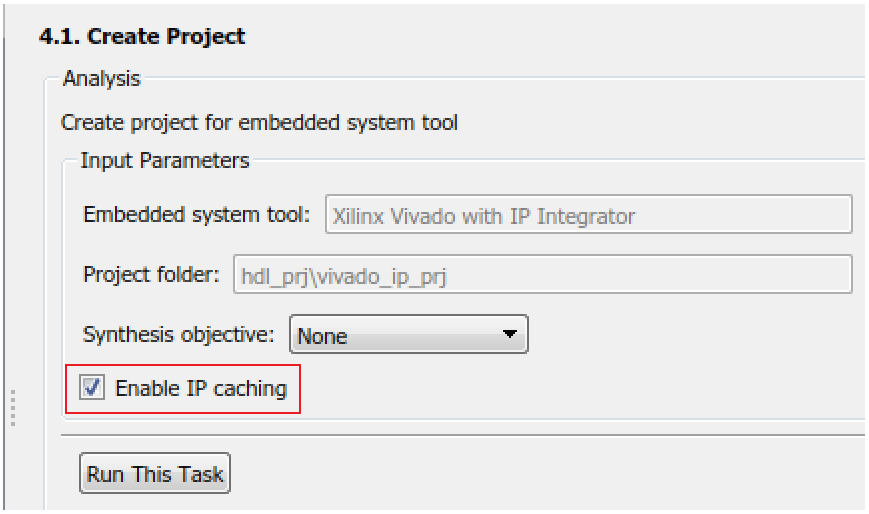


For large reference designs, the OOC flow improves synthesis run time, because you do not have to resynthesize the IP when you modify your design and run the workflow. To learn more about the OOC workflow and IP synthesis options, see the Xilinx documentation.

Enable IP Caching

Before you enable IP caching, specify `IP Core Generation` as the target workflow, and then specify the target platform settings. To enable IP caching:

- From the HDL Workflow Advisor, in the **Create Project** task, select the **Enable IP caching** check box.



4.1. Create Project

Analysis

Create project for embedded system tool

Input Parameters

Embedded system tool: Xilinx Vivado with IP Integrator

Project folder: hdl_prj\vivado_ip_prj

Synthesis objective: None

[✓] Enable IP caching

- From the command line, use the `EnableIPCaching` property of the `hdlcoder.WorkflowConfig` class. To use this property, create an object of the `hdlcoder.WorkflowConfig` class, or export the HDL Workflow Advisor settings to a script.

```
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado','TargetWorkflow','IP C
% ...
% ...
hWC.EnableIPCaching = true;
```

IP Caching in HDL Coder Reference Designs

Use IP caching for large reference designs that have a significant synthesis time. For example, the HDL Coder reference design `Default video system (requires HDMI FMC module)` is a potential candidate for IP caching.

Note: The `Speedgoat IO333-325K` board that you use with the `Simulink Real-Time FPGA I/O` workflow comes with an IP cache. The first time that you run the workflow, the code generator reuses this IP cache, which improves reference design synthesis time.

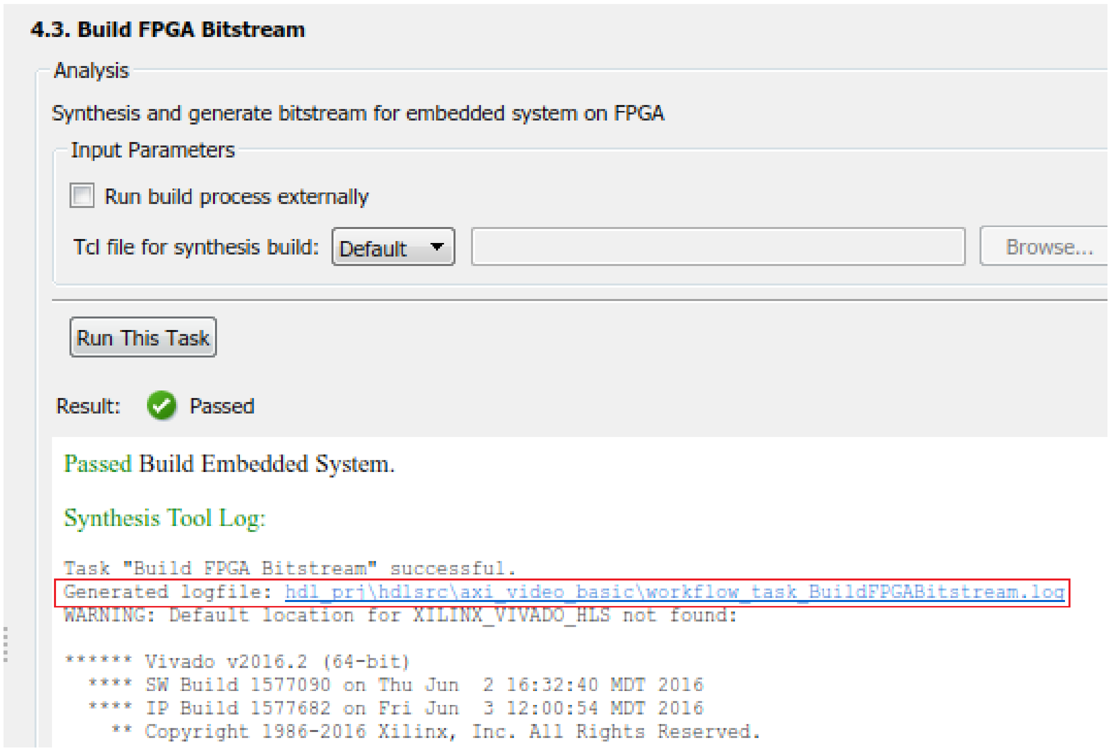
To enable IP caching, in the HDL Workflow Advisor, specify `IP Core Generation` as the target workflow, and then specify the target platform settings. Before you run the workflow for the first time:

1. In the **Create Project** task, select the **Enable IP caching** check box.

   When you run this task, the workflow creates an empty IP cache folder. You can see the `ipcache` folder in the `hdl_prj/vivado_ip_prj` path.
2. Run the **Build FPGA Bitstream** task.

   This task populates the IP cache folder with synthesis logs and design checkpoint files generated for the HDL IP core and other IP blocks in the reference design. When this task has run successfully, you can see the generated files in the `ipcache` folder.

When you run the `IP Core Generation` workflow a second time, in the **Build FPGA Bitstream** task, you can see an improvement in the task run time. Make sure that you use the same IP settings and `hdl_prj` folder as the first time that you ran the workflow. When this task has run successfully, to see if your workflow reused the IP cache, open the `workflow_task_buildFPGABitstream.log` file.



Synthesis and generate bitstream for embedded system on FPGA

Run build process externally: ☐

Tcl file for synthesis build: Default

Result: ✓ Passed

Passed Build Embedded System.

Synthesis Tool Log:

```
Task "Build FPGA Bitstream" successful.
Generated logfile: hdl_prj\hdlsrc\axi_video_basic\workflow_task_BuildFPGABitstream.log
WARNING: Default location for XILINX_VIVADO_HLS not found:

****** Vivado v2016.2 (64-bit)
  **** SW Build 1577090 on Thu Jun  2 16:32:40 MDT 2016
  **** IP Build 1577682 on Fri Jun  3 12:00:54 MDT 2016
    ** Copyright 1986-2016 Xilinx, Inc. All Rights Reserved.
```

This code snippet shows that the Vivado project launches a maximum number of jobs to synthesize the design and reuse the IP modules in the IP cache folder. You can see that the `cacheID` of the IP modules match the file names of the subfolders in the `ipcache` folder.

```
...
# reset_run impl_1
# reset_run synth_1
# launch_runs -jobs 4 synth_1
...
...
...
INFO: [IP_Flow 19-4760] Using cached IP synthesis design for IP system_top_RGBtoYCbCr_
INFO: [IP_Flow 19-4760] Using cached IP synthesis design for IP system_top_YCbCrtoRGB_
INFO: [IP_Flow 19-4760] Using cached IP synthesis design for IP system_top_xbar_0, cac
INFO: [IP_Flow 19-4760] Using cached IP synthesis design for IP system_top_axis2hdmi_0
```

...

IP Caching in Custom Reference Designs

If you are using your own custom reference design, IP caching can accelerate reference design synthesis when you run the workflow for the first time. To reuse the IP cache, create an IP cache zip file, and then make sure that the reference design definition file points to this zip file.

To create an IP cache zip file:

1. Open the HDL Workflow Advisor for any Simulink model that has a DUT subsystem, and then run the **IP Core Generation** workflow to the **Generate RTL Code and IP Core** task.
2. In the **Create Project** task, select the **Enable IP caching** check box, and then click **Run This Task**. This task creates an empty IP cache folder.
3. Run the workflow to the **Build FPGA Bitstream** task. This task populates the IP cache with the HDL IP core and the reference design IP modules.
4. In the IP cache folder, delete the IP core files generated for the DUT. Extract the remaining files from this folder into a zip file, name it `ipcache.zip`, and then save the file in the reference design folder.

To reuse the IP cache, in the reference design definition file `plugin_rd.m`, use the `IPCacheZipFile` property of the `hdlcoder.ReferenceDesign` class. By using that property, you add the `ipcache.zip` file to the Xilinx Vivado project.

```
function hRD = plugin_rd()
% Reference design definition

hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');
% ...
% ...
hRD.IPCacheZipFile = 'ipcache.zip';
```

When you use the workflow to target your custom reference design, the code generator selects the **Enable IP caching** check box. To see the improvement in synthesis time, run the **Build FPGA Bitstream** task.

See Also

hdlcoder.Board | hdlcoder.ReferenceDesign

More About

- “Custom IP Core Generation” on page 26-2
- “Custom IP Core Report” on page 26-5
- “Board and Reference Design Registration System” on page 27-17
- “Run HDL Workflow with a Script” on page 22-65

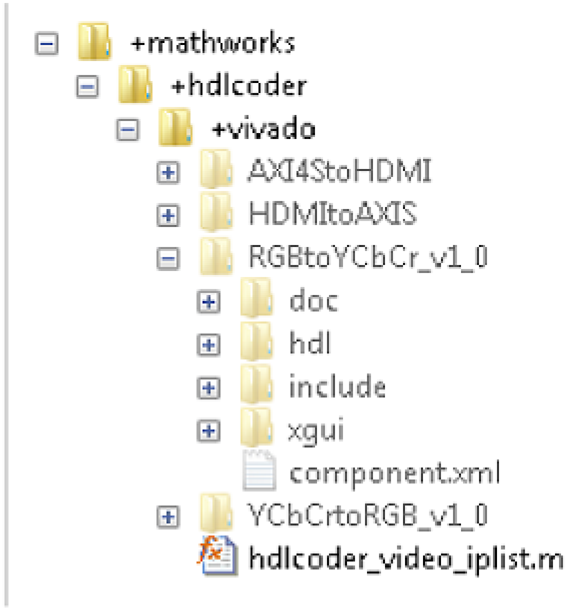
Define and Add IP Repository to Custom Reference Design

In this section...

When you create your custom reference design, you might require custom IP modules that do not come with Altera Qsys or Xilinx Vivado. To use custom IP modules, create your own IP repository folder that contains IP module subfolders. The `IP Core Generation` workflow then uses these custom IP modules when creating the reference design project. You can create multiple IP repositories and add all or some of the IP modules in each repository to your custom reference design project. You can also reuse and share IP repositories across multiple reference designs.

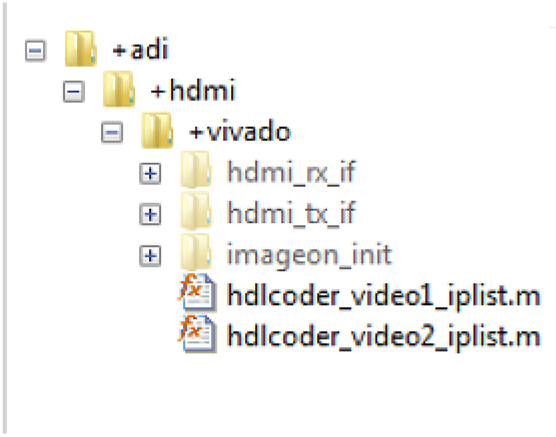
Create an IP Repository Folder Structure

Create an IP repository folder anywhere on the MATLAB path. This figure shows a typical IP repository folder structure.



When you create the folder structure, use the naming convention `+(company)/+(product)/+(tool)`. In this example, the folder structure is `+(mathworks)/+(hdlcoder)/+(vivado)`. The folder `+vivado` acts as the IP repository. This folder contains subfolders corresponding to IP modules such as `AXI4StoHDMI` and `HDMItoAXIS`. The folder also contains a `hdlcoder_video_iplist` MATLAB function. Using this function, specify the IP modules to add to the reference design.

The same repository folder can have multiple MATLAB functions. This figure shows two MATLAB functions, `hdlcoder_video1_iplist` and `hdlcoder_video2_list`, in the `+vivado` folder. The functions can share the same IP modules or point to different IP modules in the repository.



Note: If your synthesis tool is Xilinx Vivado, the IP modules in the repository folder can be in zip file format.

Define IP List Function

Create a MATLAB function that specifies the IP modules to add to the reference design. Save this function in the IP repository folder. For the function name, use the naming convention `hdlcoder_<specific_use>_iplist`. This example uses `hdlcoder_video_iplist` as the function name because it targets video applications. Using this function, specify whether you want to add all or some of the IP modules in the repository to the reference design project. To add all IP modules, use an empty cell array for `ipList`. This MATLAB code shows how to add all IP modules in the repository to the reference design.

```
function [ ipList ] = hdlcoder_video_iplist(  )
% All IP modules in the repository folder.

ipList = {};
```

To add some of the IP modules that are in the folder, specify the IP modules as a cell array of character vectors. This MATLAB code specifies the `AXI4StoHDMI` IP and the `HDMItoAXIS` IP as the IP modules to add to your custom reference design.

```
function [ ipList ] = hdlcoder_video_iplist(  )
% AXI4StoHDMI and HDMItoAXIS IP in the repository folder.

ipList = {'AXI4StoHDMI','HDMItoAXIS'};
```

Add IP List Function to Reference Design Project

Using the `addIPRepository` method of the `hdlcoder.ReferenceDesign` class, add the IP list function to your custom reference design. This example reference design adds `hdlcoder_video_iplist` to the custom reference design `My Reference Design`.

```
function hRD = plugin_rd()
% Reference design definition

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');
hRD.ReferenceDesignName = 'My Reference Design';
hRD.BoardName = 'ZedBoard'

% Tool information
hRD.SupportedToolVersion = {'2016.2'};

%% Add custom design files
hRD.addCustomVivadoDesign( ...
    'CustomBlockDesignTcl', 'system_top.tcl', ...
    'VivadoBoardPart',      'em.avnet.com:zed:part0:1.0');

% Add IP Repository
hRD.addIPRepository(...
    'IPListFunction', 'mathworks.hdlcoder.vivado.hdlcoder_video_iplist',
   'NotExistMessage', 'IP repository not found');

% ...
% ...
```

To use the IP modules when the code generator creates the project, open the HDL Workflow Advisor, and run the `IP Core Generation` workflow to the **Create Project** task. After running this task, you can see the IP module subfolders in the repository copied over to the `ipcore` folder of the project. The `CustomBlockDesignTcl` can then use these IP modules.

See Also

hdlcoder.Board | hdlcoder.ReferenceDesign

Related Examples

- Define and Register Custom Board and Reference Design for SoC Workflow

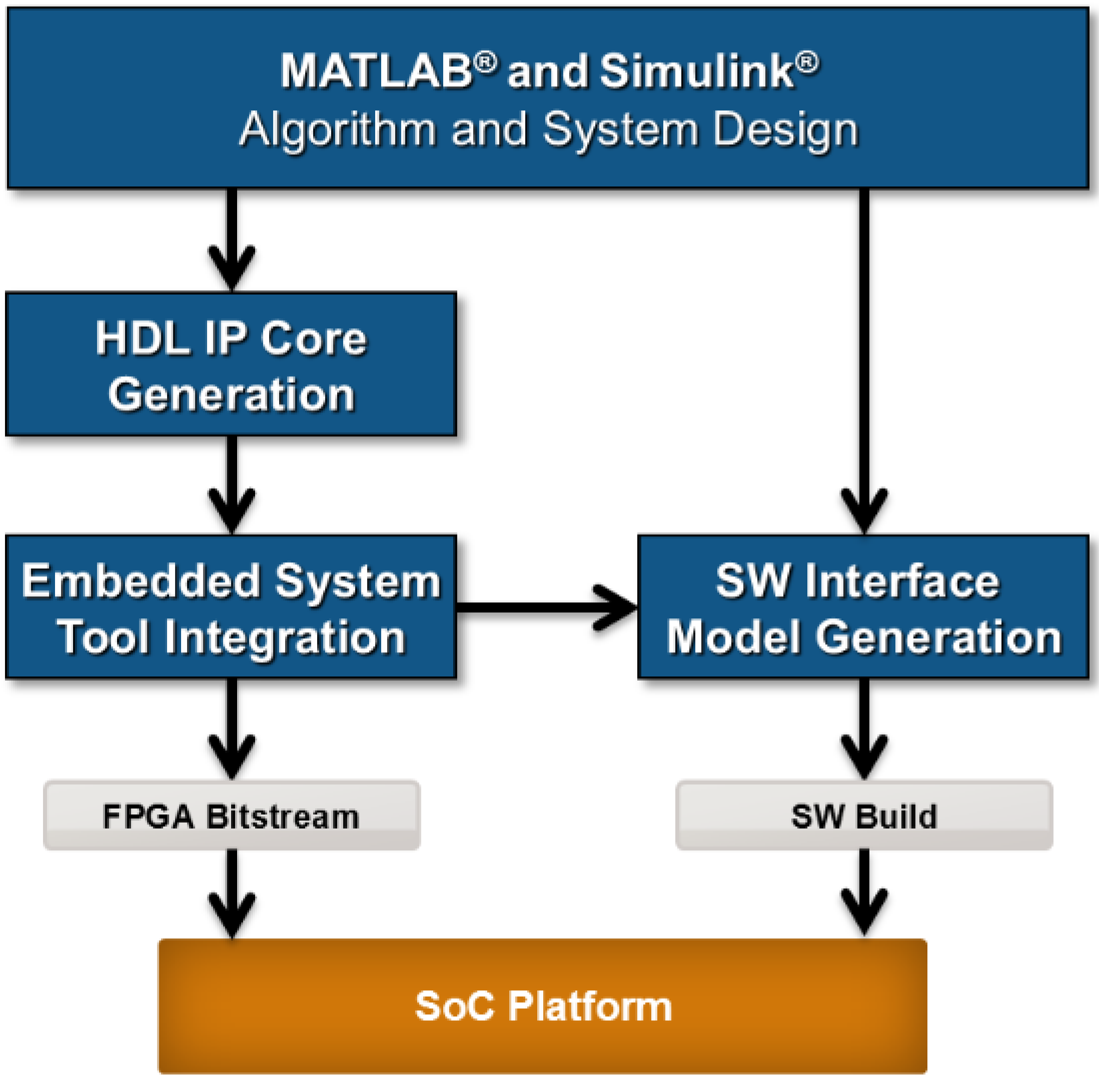
More About

- “Board and Reference Design Registration System” on page 27-17
- “Register a Custom Board” on page 27-21
- “Register a Custom Reference Design” on page 27-24

Target SoC Platforms and Speedgoat Boards

Hardware-Software Co-Design Workflow for SoC Platforms

The HDL Coder hardware-software co-design workflow helps automate the deployment of your MATLAB and Simulink design to a Zynq-7000 platform or Altera SoC platform. You can explore the best ways to partition and deploy your design by iterating through the following workflow.

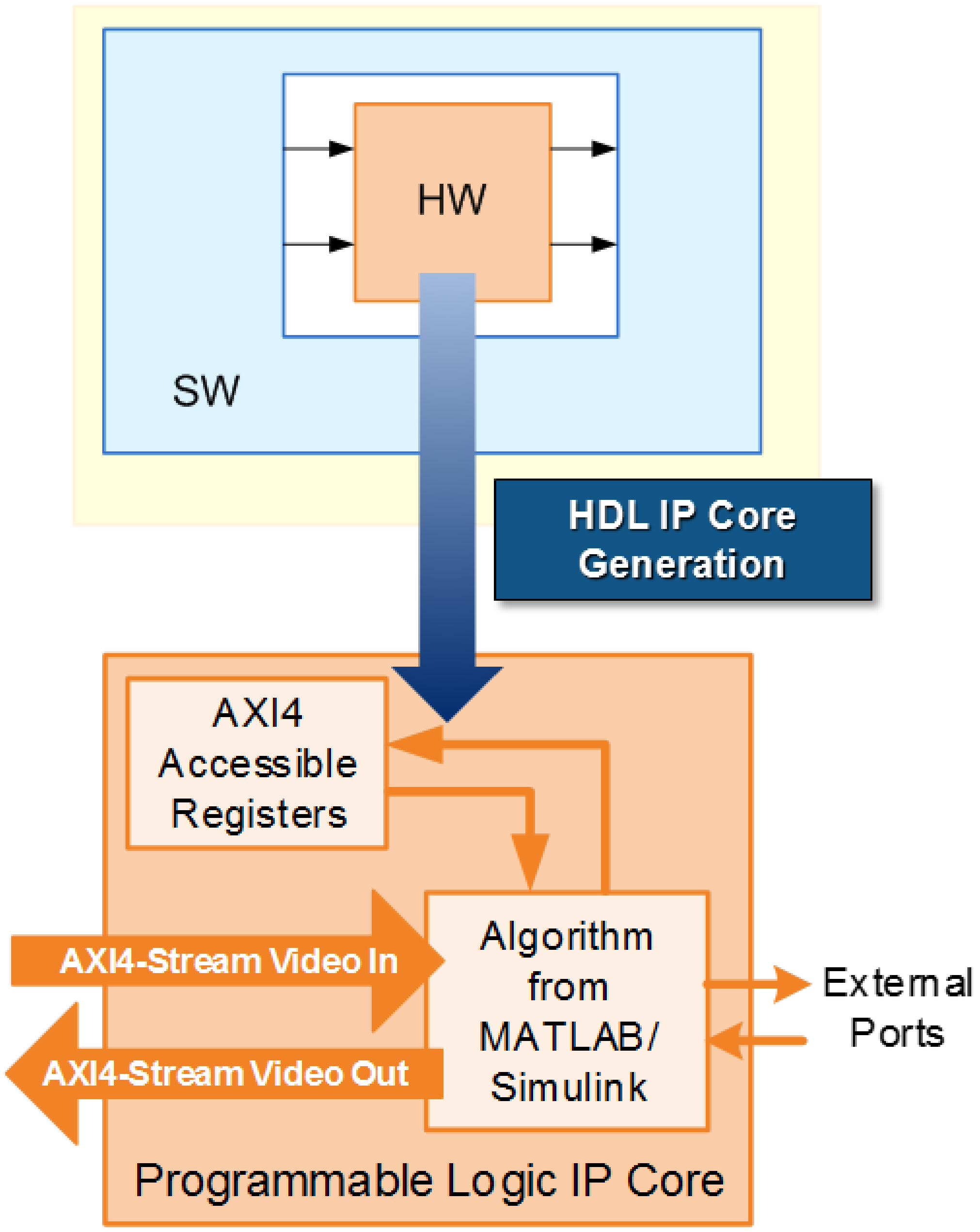

MATLAB® and Simulink®
Algorithm and System Design
HDL IP Core Generation
Embedded System Tool Integration
SW Interface Model Generation
FPGA Bitstream
SW Build
SoC Platform

1. *MATLAB and Simulink Algorithm and System Design*: You begin by implementing your design in MATLAB or Simulink. When the design behavior meets your requirements, decide how to partition your design: which parts you want to run in hardware, and which parts you want to run in embedded software.

   The part of the design that you want to run in hardware must use MATLAB syntax or Simulink blocks that are supported and configured for HDL code generation. See:

   - “MATLAB Algorithm Design”
   - “Model and Architecture Design”
2. *HDL IP Core Generation*: Enclose the hardware part of your design in an atomic Subsystem block or MATLAB function, and use the HDL Workflow Advisor to define and generate an HDL IP core.

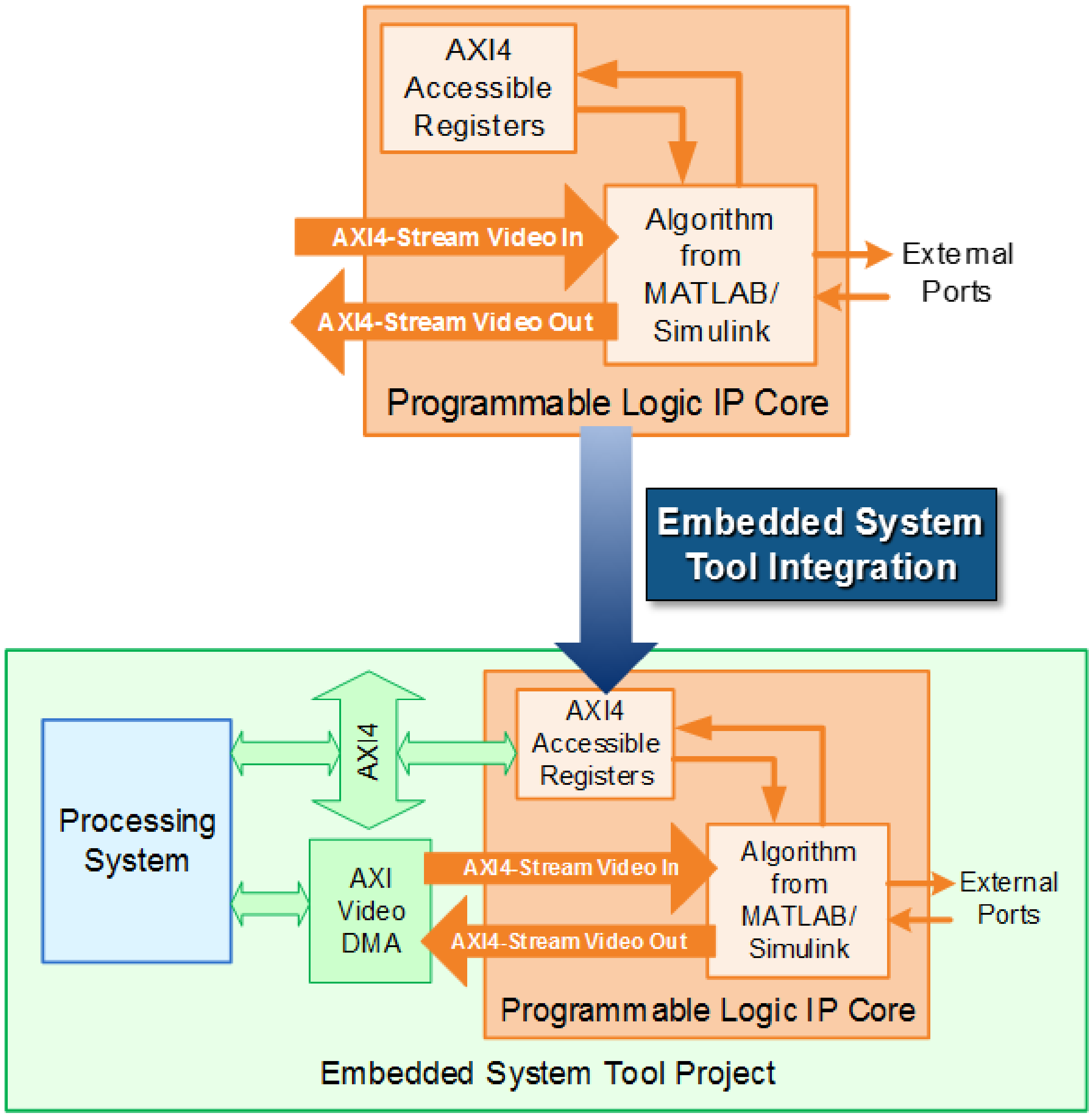
   The following diagram shows a design that has been partitioned into a hardware part, in orange, and software part, in blue. HDL IP core generation creates an IP core from the hardware part of the model. The IP core includes hardware interface components such as AXI4 or AXI4-Lite interface-accessible registers, AXI4 or AXI4-Lite interfaces, AXI4-Stream Video interfaces, and external ports.


HW
SW
HDL IP Core Generation
AXI4 Accessible Registers
AXI4-Stream Video In
AXI4-Stream Video Out
Algorithm from MATLAB/ Simulink
External Ports
Programmable Logic IP Core

3 *Embedded System Tool Integration*: As part of the HDL Workflow Advisor IP core generation workflow, you insert your generated IP core into a *reference design*, and generate an FPGA bitstream for the SoC hardware.

   The *reference design* is a predefined embedded system integration project. It contains all the elements the Altera or Xilinx software needs to deploy your design to the SoC platform, except for the custom IP core and embedded software that you generate.

   The following diagram shows the relationship between the reference design, in green, and the generated IP core, in orange.

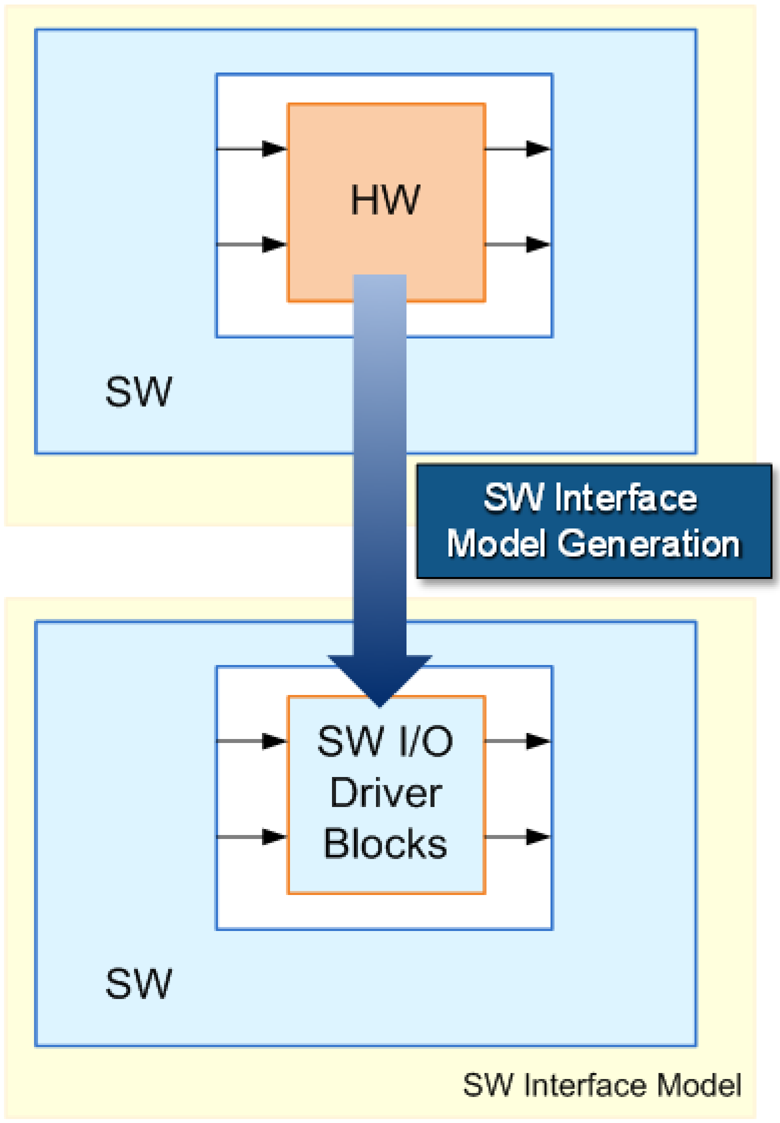

AXI4 Accessible Registers
AXI4-Stream Video In
AXI4-Stream Video Out
Algorithm from MATLAB/ Simulink
External Ports
Programmable Logic IP Core
Embedded System Tool Integration
Processing System
AXI4
AXI Video DMA
AXI4 Accessible Registers
AXI4-Stream Video In
AXI4-Stream Video Out
Algorithm from MATLAB/ Simulink
External Ports
Programmable Logic IP Core
Embedded System Tool Project

4 *SW Interface Model Generation* (requires a Simulink license and Embedded Coder license): In the HDL Workflow Advisor, after you generate the IP core and insert it into the reference design, you can optionally generate a software interface model. The software interface model is your original model with AXI driver blocks replacing the hardware part.

   If you have an Embedded Coder license, you can automatically generate the software interface model, generate embedded code from it, and build and run the executable on the Linux kernel on the ARM® processor. The generated embedded software includes AXI driver code, generated from the AXI driver blocks, that controls the HDL IP core.

   If you do not have an Embedded Coder license or Simulink license, you can write the embedded software and manually build it for the ARM processor.

   The following diagram shows the difference between the original model and the software interface model.



5 *SoC Platform* and *External Mode PIL*: Using the HDL Workflow Advisor, you program your FPGA bitstream to the SoC platform. You can then run the software

interface model in external mode, or processor-in-the-loop (PIL) mode, to test your deployed design.

If your deployed design does not meet your design requirements, you can repeat the workflow with a modified model, or a different hardware-software partition.

Related Examples

- “Xilinx Zynq Platform”
- “Altera SoC Platform”

Model Design for AXI4-Stream Interface Generation

In this section...

With the HDL Coder software, you can implement a simplified, streaming protocol in your model. The software generates AXI4-Stream interfaces in the IP core.

Simplified Streaming Protocol

When you want to generate an AXI4-Stream interface in your IP core, in your DUT interface, implement the following signals:

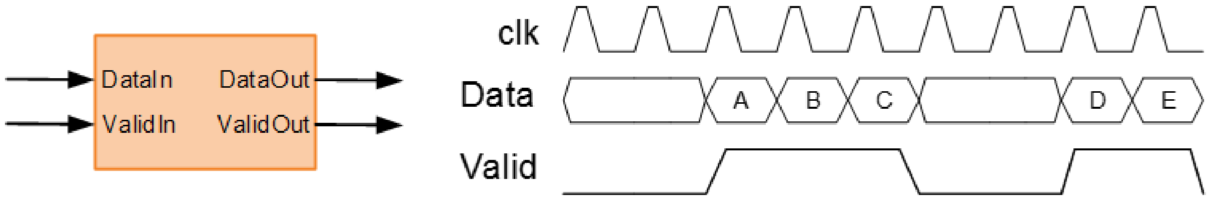
- Data
- Valid

When you map scalar DUT ports to an AXI4-Stream interface, you can optionally model the following signals and map them to the AXI4-Stream interface:

- Ready
- Other protocol signals, such as:
  - `TSTRB`
  - `TKEEP`
  - `TLAST`
  - `TID`
  - `TDEST`
  - `TUSER`

Data and Valid Signals

When the Data signal is valid, the Valid signal is asserted.



Map Scalar Ports To AXI4-Stream Interface

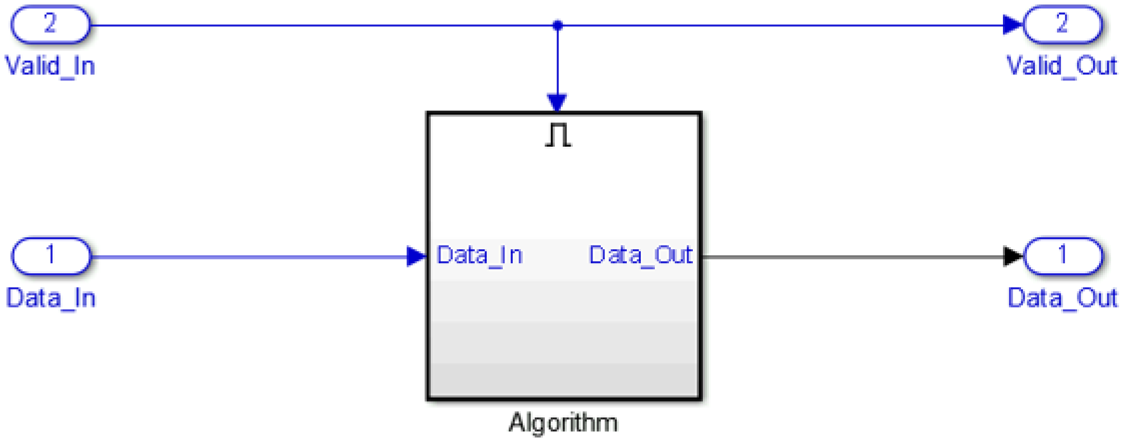
If you want to generate a hardware IP core, but do not need to model and simulate the interaction between the software and hardware, use scalar data ports at your DUT interface. Map the data ports to AXI4-Stream interfaces.

Data and Valid Signal Modeling Pattern

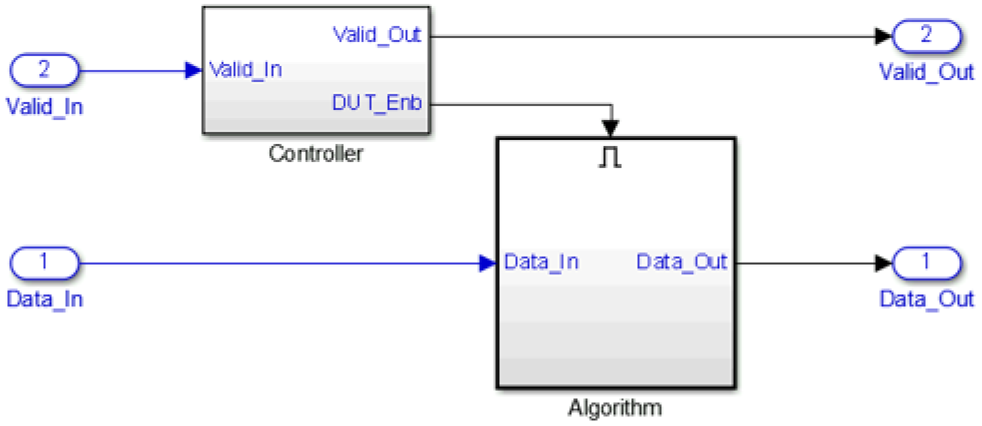
To model the Data and Valid signals in Simulink:

1. Enclose the algorithm that processes the Data signal by using an enabled subsystem.
2. Control the enable port of the enabled subsystem by using the Valid signal.

For example, you can directly connect the Valid signal to the enable port.



You can also use a controller in your DUT that generates an enable signal for the enabled subsystem.

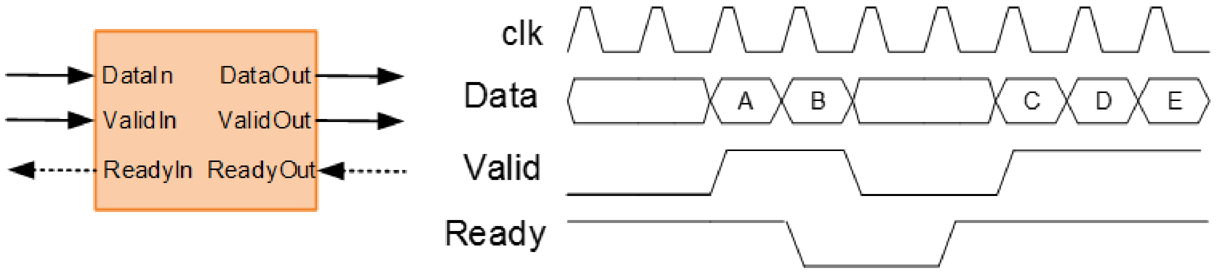


Ready Signal (Optional)

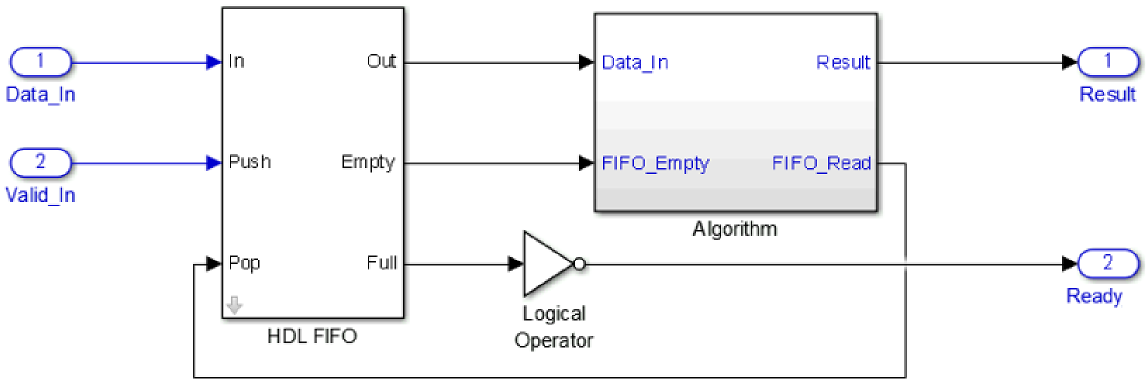
The AXI4-Stream interfaces in your DUT can optionally include a Ready signal. In a Slave interface, the Ready signal enables you to apply back pressure. In a Master interface, the Ready signal enables you to respond to back pressure.

If you model the Ready signal in your AXI4-Stream interfaces, your Master interface must deassert its Valid signal one cycle after the Ready signal is deasserted.

If you do not model the Ready signal, the coder generates the signal and the associated back pressure logic.



For example, if you have a FIFO in your DUT to store a frame of data, to apply back pressure to the upstream component, you can model the Ready signal based on the FIFO Full signal.



Note: If you enable delay balancing, the coder can insert one or more delays on the Ready signal. Disable delay balancing for the Ready signal path.

Other Protocol Signals (optional)

You can optionally model other AXI4-Stream protocol signals. If you model only the required Data and Valid signals, the coder generates the `TREADY` and `TLAST` AXI4-Stream protocol signals.

If you do not model the `TLAST` signal, the coder generates a programmable register in the IP core so that you can specify your packet size. The details of the programmable packet size register are in your IP core generation report.

Map Vector Ports To AXI4-Stream Interface

If you want to model and simulate the system interaction between the software and hardware, and generate code for the software driver, use vector data ports at your DUT interface. Map the data ports to AXI4-Stream interfaces.

Data and Valid Signal Modeling Requirements

When you map vector ports to AXI4-Stream interfaces, your model has these requirements:

- Connect each DUT input vector data port to a Serializer1D block.

  The Serializer1D block must have a ValidOut port and the Ratio set to the vector bit width.
- Connect each DUT output vector data port to a Deserializer1D block.

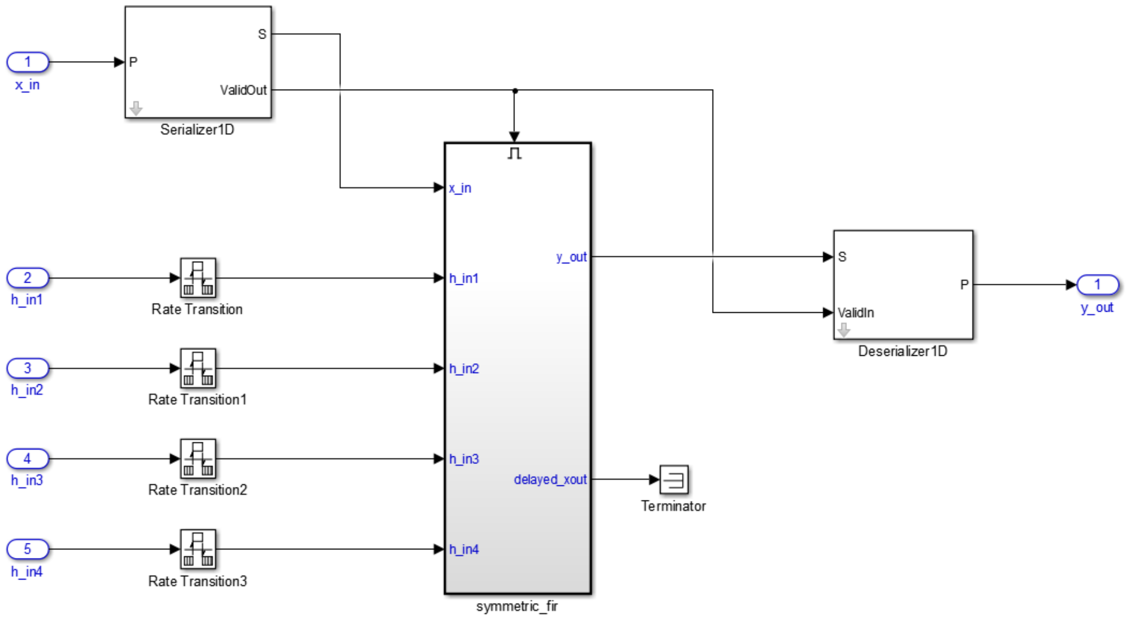
  The Deserializer1D block must have a ValidIn port and the Ratio set to the vector bit width.
- Connect ach scalar port that maps to an AXI4-Lite interface to a Rate Transition block.

  The Ratio in the Rate Transition block must match the Ratio in the Serializer1D and Deserializer1D blocks.
- Each scalar port that maps to an external port must have the same sample time as the streaming algorithm subsystem.

The streaming algorithm subsystem follows the same Data and Valid signal modeling pattern as for mapping scalar ports to an AXI4-Stream interfaces. See "Data and Valid Signal Modeling Pattern" on page 27-12.

Example

For an example that shows how to map vector ports to AXI4-Stream interfaces, open the `hdlcoder_sfir_fixed_vector` model. In the `hdlcoder_sfir_fixed_vector` model, `symmetric_fir` is the streaming algorithm subsystem.



Restrictions

When you map scalar or vector DUT ports to AXI4-Stream interfaces:

- The DUT can have at most one AXI4-Stream Master channel and one AXI4-Stream Slave channel.
- Xilinx Zynq-7000 must be your target platform.
- Xilinx Vivado must be your synthesis tool.
- **Processor/FPGA synchronization** must be `Free running`.

When you map scalar DUT ports to AXI4-Stream interfaces, the DUT must be single-rate.

When you map vector DUT ports to AXI4-Stream interfaces:

- The streaming algorithm must be single-rate.
- You cannot use protocol signals other than Data and Valid. For example, Ready and TLAST are not supported.

Board and Reference Design Registration System

In this section...

You can define custom boards and custom reference designs so that they are available as target hardware options in the SoC workflow. Custom boards and custom reference designs use the same system that HDL Coder uses for predefined board and reference design targets.

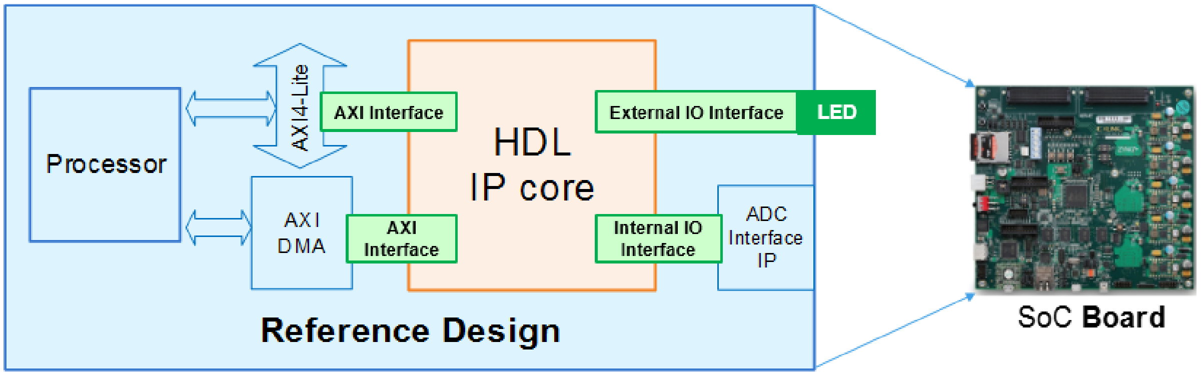
Board, IP Core, and Reference Design Definitions

A *reference design* is the embedded system design that your generated IP core integrates with. The *board* is the SoC platform.

For a custom board or custom reference design, you can define different kinds of interfaces:

- *AXI interface*: an interface between your generated IP core and an AXI4 or AXI4-Lite interface.
- *External IO interface*: an interface between your generated IP core and an external interface.
- *Internal IO interface*: an interface between your generated IP core and another IP core in the reference design.

After you integrate your reference design and IP core in an embedded system design project, you can program the board with the embedded system design.



Board Registration Files

To define and register a board, you must have a *board definition*, a *board plugin*, and a *board registration file*.

Board Definition

A board definition is a file that defines the characteristics of a board. You can define more than one custom board.

Board Plugin

A board plugin is a package folder that contains:

- The board definition.
- All reference design plugins that are associated with the board.

A board plugin has one board definition, but can have multiple reference designs.

Board Registration File

A board registration file is always named `hdlcoder_board_customization.m`, and contains a list of board plugins. There can be multiple board registration files on your MATLAB path, but a board plugin cannot be listed in more than one board registration file.

When the HDL Workflow Advisor opens, it searches the MATLAB path for files named `hdlcoder_board_customization.m`, and uses the information to populate the target board options. Interfaces you add and define for the board appear as options in the **Target Platform Interface** dropdown list.

Reference Design Registration Files

To define and register a reference design, you must have a *reference design definition*, a *reference design plugin*, and a *reference design registration file*.

Reference Design Definition

A reference design definition is a file that defines the characteristics of a reference design, including its associated board and interfaces. You can define multiple custom reference designs per board.

Reference Design Plugin

A reference design plugin is a package folder that contains:

- The reference design definition.
- Files that are part of the embedded system design project, and are specific to your third-party synthesis tool, including Tcl, project, and design files.

A reference design plugin has one reference design definition and is associated with one board.

Reference Design Registration File

A reference design registration file is always named `hdlcoder_ref_design_customization.m`, and contains a list of reference design plugins for a specific board. There can be multiple reference design registration files for a specific board on your MATLAB path, but a reference design plugin cannot be listed in more than one reference design plugin registration file.

When the HDL Workflow Advisor opens, it searches the MATLAB path for files named `hdlcoder_ref_design_customization.m`, and uses the information to populate the reference design options for each board. Interfaces you add and define for the reference design appear as options in the **Target Platform Interface** dropdown list.

Predefined Board and Reference Design Examples

For examples of working board and reference design definitions, refer to the predefined Altera SoC and Xilinx Zynq board plugins that include predefined reference design plugins:

- `*support_package_installation_folder*/toolbox/hdlcoder/supportpackages/zynq7000/+ZedBoard/`
- `*support_package_installation_folder*/toolbox/hdlcoder/supportpackages/zynq7000/+ZynqZC702/`
- `*support_package_installation_folder*/toolbox/hdlcoder/supportpackages/alterasoc/+AlteraCycloneV/`
- `*support_package_installation_folder*/toolbox/hdlcoder/supportpackages/alterasoc/+ArrowSoCKit/`

See Also

hdlcoder.Board | hdlcoder.ReferenceDesign

Related Examples

- "Register a Custom Board" on page 27-21
- "Register a Custom Reference Design" on page 27-24
- Define and Register Custom Board and Reference Design for SoC Workflow

Register a Custom Board

To register a custom board, you must:

1. Define a board.
2. Create a board plugin.
3. Define a board registration function, or add the new board plugin to an existing board registration function.

In this section...

Define a Board

Before you begin, have the board documentation at hand so you can refer to the details of the board.

Requirements

A board definition must be:

- A MATLAB function that returns an hdlcoder.Board object.

  The board definition function can have any name.
- In its board plugin folder.

How To Define A Board

1. Create a new file that defines a MATLAB function with any name.
2. In the MATLAB function, create an hdlcoder.Board object and specify its properties and interfaces according the characteristics of your custom board.
3. Optionally, to check that the definition is complete, run the `validateBoard` method.

For example, the following code defines a board:

```
function hB = plugin_board()
```

```
% Board definition

% Construct board object
hB = hdlcoder.Board;

hB.BoardName    = 'Digilent Zynq ZyBo';

% FPGA device information
hB.FPGAVendor   = 'Xilinx';
hB.FPGAFamily   = 'Zynq';
hB.FPGADevice   = 'xc7z010';
hB.FPGAPackage  = 'clg400';
hB.FPGASpeed    = '-2';

% Tool information
hB.SupportedTool = {'Xilinx Vivado'};

% FPGA JTAG chain position
hB.JTAGChainPosition = 2;

%% Add interfaces
% Standard "External Port" interface
hB.addExternalPortInterface( ...
    'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});
```

Create a Board Plugin

Requirements

A board plugin:

- Must be a package folder that contains the board definition file.

  A package folder has a + prefix before the folder name. For example, the board plugin can be a folder named `+ZedBoard`.
- Must be on the MATLAB path.
- Can contain one or more reference design plugins.

How To Create a Board Plugin

1. Create a folder that has a name with a + prefix.
2. Save your board definition file to the folder.

3 Add the folder to your MATLAB path.

Define a Board Registration Function

Requirements

A board registration function:

- Must be named `hdlcoder_board_customization.m`.
- Returns a list of board plugins, specified as a cell array of character vectors.
- Must be on the MATLAB path.

How To Define a Board Registration Function

1. Create a file named `hdlcoder_board_customization.m` and save it anywhere on the MATLAB path.
2. In `hdlcoder_board_customization.m`, define a function that returns a list of board plugins as a cell array of character vectors.

   For example, the following code defines a board registration function.

   ```
   function r = hdlcoder_board_customization
   % Board plugin registration files
   % Format: % board_folder.board_definition_function

   r = {'ZyboRegistration.plugin_board'};

   end
   ```

See Also

hdlcoder.Board | hdlcoder.ReferenceDesign

Related Examples

- "Register a Custom Reference Design" on page 27-24
- Define and Register Custom Board and Reference Design for SoC Workflow

More About

- "Board and Reference Design Registration System" on page 27-17

Register a Custom Reference Design

In this section...

To register a custom reference design:

1. Define a reference design.
2. Create a reference design plugin.
3. Define a reference design registration function, or add the new reference design plugin to an existing reference design registration function.

Define a Reference Design

A reference design definition must be a MATLAB function that returns an hdlcoder.ReferenceDesign object. Create the reference design definition function in the reference design plugin folder. You can use any name for the reference design definition function.

To create a reference design definition:

1. Create a new file that defines a MATLAB function with any name.
2. In the MATLAB function, create an hdlcoder.ReferenceDesign object and specify its properties and interfaces according to the characteristics of your embedded system design.
3. If you want to check that the definition is complete, run the `validateReferenceDesign` method.

This MATLAB function defines a custom reference design:

```
function hRD = plugin_rd()
% Reference design definition

% Construct reference design object
hRD = hdlcoder.createReferenceDesign('SynthesisTool', 'Xilinx Vivado');

hRD.ReferenceDesignName = 'Demo system (Vivado 2014.2)';
hRD.BoardName = 'Digilent Zynq ZyBo';

% Tool information
% It is recommended to use a tool version that is compatible with the supported tool
% version. If you choose a different tool version, it is possible that HDL Coder is
% unable to create the reference design project for IP core integration.
hRD.SupportedToolVersion = {'2015.4'};

%% Add custom design files
% add custom Vivado design
hRD.addCustomVivadoDesign( ...
    'CustomBlockDesignTcl', 'design_led.tcl');

hRD.CustomFiles = {'ZYBO_zynq_def.xml'};
%% Add interfaces
% add clock interface
hRD.addClockInterface( ...
    'ClockConnection',   'clk_wiz_0/clk_out1', ...
    'ResetConnection',   'proc_sys_reset_0/peripheral_aresetn');

% add AXI4 and AXI4-Lite slave interfaces
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'axi_interconnect_0/M00_AXI', ...
    'BaseAddress',         '0x40010000', ...
    'MasterAddressSpace',  'processing_system7_0/Data');
```

By default, HDL Coder generates an IP core with the default settings and integrates it into the reference design project. To customize these default settings, use the properties in the hdlcoder.ReferenceDesign object to define custom parameters and to register the function handle of the custom callback functions. For more information, see “Define Custom Parameters and Callback Functions for Custom Reference Design” on page 27-28.

Create a Reference Design Plugin

A reference design plugin is a package folder that you define on the MATLAB path. The folder contains the board definition file and any custom callback functions.

To create a reference design plugin:

1 In the board plugin folder for the associated board, create a new folder that has a name with a + prefix.

For example, the reference design plugin can be a folder named `+vivado_base_ref_design`.

2. In the new folder, save your reference design definition file and any custom callback functions that you create.
3. In the new folder, save any files that are required by the embedded system design project, and are specific to your third-party synthesis tool, including Tcl, project, and design files.
4. Add the folder to your MATLAB path.

Define a Reference Design Registration Function

A reference design registration function contains a list of reference design functions and the associated board name. You must name the function `hdlcoder_ref_design_customization.m`. When the HDL Workflow Advisor opens, it searches the MATLAB path for files named `hdlcoder_ref_design_customization.m`, and uses the information to populate the reference design options for each board.

To define a reference design registration function:

1. Create a file named `hdlcoder_ref_design_customization.m` and save it anywhere on the MATLAB path.
2. In `hdlcoder_board_customization.m`, define a function that returns the associated board name, specified as a character vector, and a list of reference design plugins, specified as a cell array of character vectors.

   For example, the following code defines a reference design registration function.

   ```
   function [rd, boardName] = hdlcoder_ref_design_customization
   % Reference design plugin registration file

   rd = {'ZyBoRegistration.Vivado2015_4.plugin_rd', ...
        };

   boardName = 'Digilent Zynq ZyBo';

   end
   ```

The reference design registration function returns the associated board name, specified as a character vector, and a list of reference design plugins, specified as a cell array of character vectors.

See Also

hdlcoder.Board | hdlcoder.ReferenceDesign

Related Examples

- “Register a Custom Board” on page 27-21
- “Define Custom Parameters and Callback Functions for Custom Reference Design” on page 27-28
- Define and Register Custom Board and Reference Design for SoC Workflow

More About

- “Board and Reference Design Registration System” on page 27-17

Define Custom Parameters and Callback Functions for Custom Reference Design

In this section...

When you define your custom reference design, you can optionally use the properties in the hdlcoder.ReferenceDesign object to define custom parameters and callback functions.

Define Custom Parameters and Register Callback Function Handle

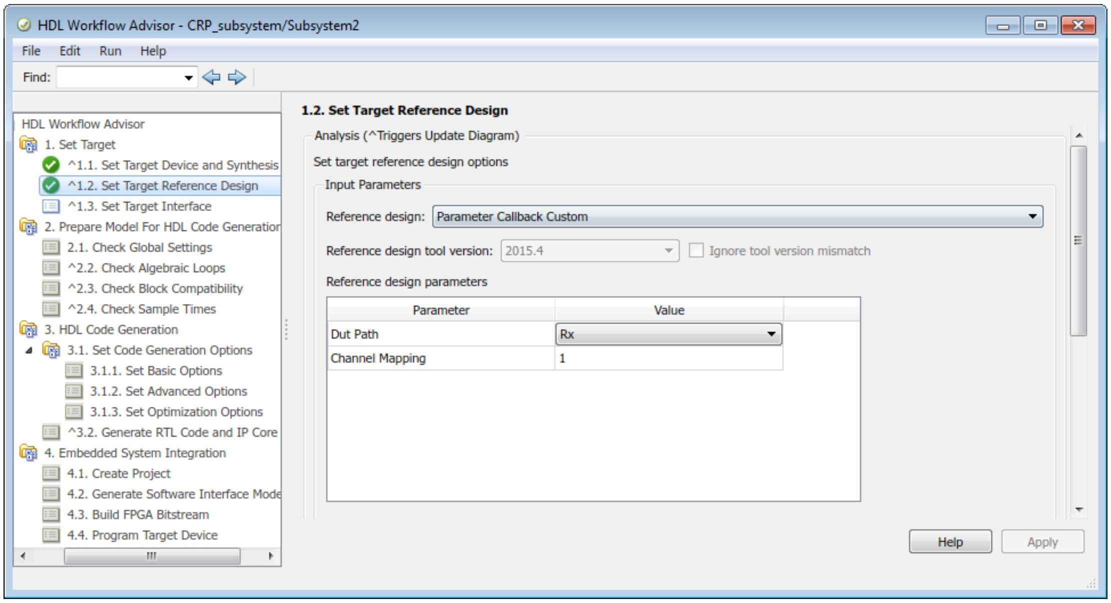
This MATLAB code shows how to define custom parameters and register the function handle of the custom callback functions in the reference design definition function.

```
function hRD = plugin_rd()
% Reference design definition

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');
hRD.ReferenceDesignName = 'My Reference Design';
hRD.BoardName = 'ZedBoard';

% Tool information
hRD.SupportedToolVersion = {'2015.4'};

%% Add custom design files
% ...
% ...

%% Add custom parameters by using addParameter property. These are optional.
% Specify custom 'DUT path' and 'Channel Mapping' parameters. The parameters get
% populated in the 'Set Target Reference Design' task in the HDL Workflow Advisor.
hRD.addParameter( ...
    'ParameterID',   'DutPath', ...
    'DisplayName',   'Dut Path', ...
    'DefaultValue',  'Rx', ...
    'ParameterType',  hdlcoder.ParameterType.Dropdown, ...
    'Choice',       {'Rx', 'Tx'});
hRD.addParameter( ...
    'ParameterID',   'ChannelMapping', ...
    'DisplayName',   'Channel Mapping', ...
    'DefaultValue',  '1');

%% Add custom callback functions. These are optional.
% With the callback functions, you can enable custom validations, customize the
% project creation, software interface model generation, and the bistream build.
% Register the function handle of these callback functions.

% Specify an optional callback for 'Set Target Reference Design' task in Workflow Advis
% Use the property name 'PostTargetReferenceDesignFcn'.
hRD.PostTargetReferenceDesignFcn = ...
    @my_reference_design.callback_PostTargetReferenceDesign;

% Specify an optional callback for 'Set Target Interface' task in Workflow Advisor.
% Use the property name 'PostTargetInterfaceFcn'.
hRD.PostTargetInterfaceFcn     = ...
    @my_reference_design.callback_PostTargetInterface;

% Specify an optional callback for 'Create Project' task
% Use the property name 'PostCreateProjectFcn' for the ref design object.
hRD.PostCreateProjectFcn       = ...
    @my_reference_design.callback_PostCreateProject;

% Specify an optional callback for 'Generate Software Interface Model' task
% Use the property name 'PostSWInterfaceFcn' for the ref design object.
hRD.PostSWInterfaceFcn         = ...
    @my_reference_design.callback_PostSWInterface;

% Specify an optional callback for 'Build FPGA Bitstream' task
% Use the property name 'PostBuildBitstreamFcn' for the ref design object.
hRD.PostBuildBitstreamFcn      = ...
    @my_reference_design.callback_PostBuildBitstream;

% Specify an optional callback for 'Program Target Device'
% task to use a custom programming method.
hRD.CallbackCustomProgrammingMethod = ...
    @my_reference_design.callback_CustomProgrammingMethod;

%% Add interfaces
% ...
% ...
```

Define Custom Parameters

With the `addParameter` method of the `hdlcoder.ReferenceDesign` class, you can define custom parameters. In the preceding example, the reference design defines two custom parameters, `DUT Path` and `Channel Mapping`. To learn more about the `addParameter` method, see addParameter (hdlcoder.ReferenceDesign)

When you open the HDL Workflow Advisor, HDL Coder populates the **Set Target Reference Design** task with the reference design name, tool version, and the custom parameters that you specify.


HDL Workflow Advisor - CRP_subsystem/Subsystem2

File
Edit
Run
Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis
^1.2. Set Target Reference Design
^1.3. Set Target Interface
2. Prepare Model For HDL Code Generation
2.1. Check Global Settings
^2.2. Check Algebraic Loops
^2.3. Check Block Compatibility
^2.4. Check Sample Times
3. HDL Code Generation
3.1. Set Code Generation Options
3.1.1. Set Basic Options
3.1.2. Set Advanced Options
3.1.3. Set Optimization Options
^3.2. Generate RTL Code and IP Core
4. Embedded System Integration
4.1. Create Project
4.2. Generate Software Interface Mode
4.3. Build FPGA Bitstream
4.4. Program Target Device
Help
Apply

1.2. Set Target Reference Design

Analysis (^Triggers Update Diagram)

Set target reference design options

Input Parameters

Reference design: Parameter Callback Custom

Reference design tool version: 2015.4 ☐ Ignore tool version mismatch

Reference design parameters

Parameter	Value
Dut Path	Rx
Channel Mapping	1

HDL Coder then passes these parameter values to the callback functions in the input structure.

If your synthesis tool is Xilinx Vivado, HDL Coder sets the reference design parameter values to variables. The variables are then input to the block design Tcl file. This code snippet is an example from the reference design project creation Tcl file.

```
update_ip_catalog
set DutPath {Rx}
set ChannelMapping {1}
source vivado_custom_block_design.tcl
```

The code shows how HDL Coder sets the reference design parameters before sourcing the custom block design Tcl file.

Register Callback Function Handles

In the reference design definition, you can register the function handle to reference the custom callback functions. You then can:

- Enable custom validations.
- Customize the reference design project creation settings.
- Change the generated software interface model.
- Customize the FPGA bitstream build process.
- Specify custom FPGA programming method.

With the `hdlcoder.ReferenceDesign` class, you can define callback property names. The callback properties have a naming convention. The callback functions can have any name. In the HDL Workflow Advisor, you can define callback functions to customize these tasks.

Workflow Advisor Task	Callback Property Name	Functionality
Set Target Reference Design	`PostTargetRefere`	Enable custom validations. For an example that shows how you can validate that the **Reset type** is `Synchronous`, see PostTargetReferenceDesignFcn (hdlcoder.ReferenceDesign).
Set Target Interface	`PostTargetInterf`	Enable custom validations. For an example that shows how you can validate not choosing a certain interface for a certain custom parameter setting, see PostTargetInterfaceFcn (hdlcoder.ReferenceDesign).
Create Project	`PostCreateProjec`	Specify custom settings when HDL Coder creates the project. For an example, see PostCreateProjectFcn (hdlcoder.ReferenceDesign).
Generate Software Interface Model	`PostSWInterfaceF`	Change the generated software interface model. For an example, see PostSWInterfaceFcn (hdlcoder.ReferenceDesign).

Workflow Advisor Task	Callback Property Name	Functionality
Build FPGA Bitstream	`PostBuildBitstre`	Specify custom settings when you build the FPGA bitstream. For an example, see PostBuildBitstreamFcn (hdlcoder.ReferenceDesign).
Program Target Device	`CallbackCustomPr`	Specify a custom FPGA programming method. For an example, see CallbackCustomProgrammingMethod (hdlcoder.ReferenceDesign).

Define Custom Callback Functions

1. For each of the callback function that you want HDL Coder to execute after running a task, create a file that defines a MATLAB function with any name.
2. Make sure that the callback function has the documented input and output arguments.
3. Verify that the functions are accessible from the MATLAB path.
4. Register the function handle of the callback functions in the reference design definition function.
5. Follow the naming conventions for the callback property names.

To learn more about these callback functions, see hdlcoder.ReferenceDesign.

See Also

hdlcoder.Board | hdlcoder.ReferenceDesign

Related Examples

- "Register a Custom Board" on page 27-21
- "Register a Custom Reference Design" on page 27-24
- Define and Register Custom Board and Reference Design for SoC Workflow

More About

- "Board and Reference Design Registration System" on page 27-17

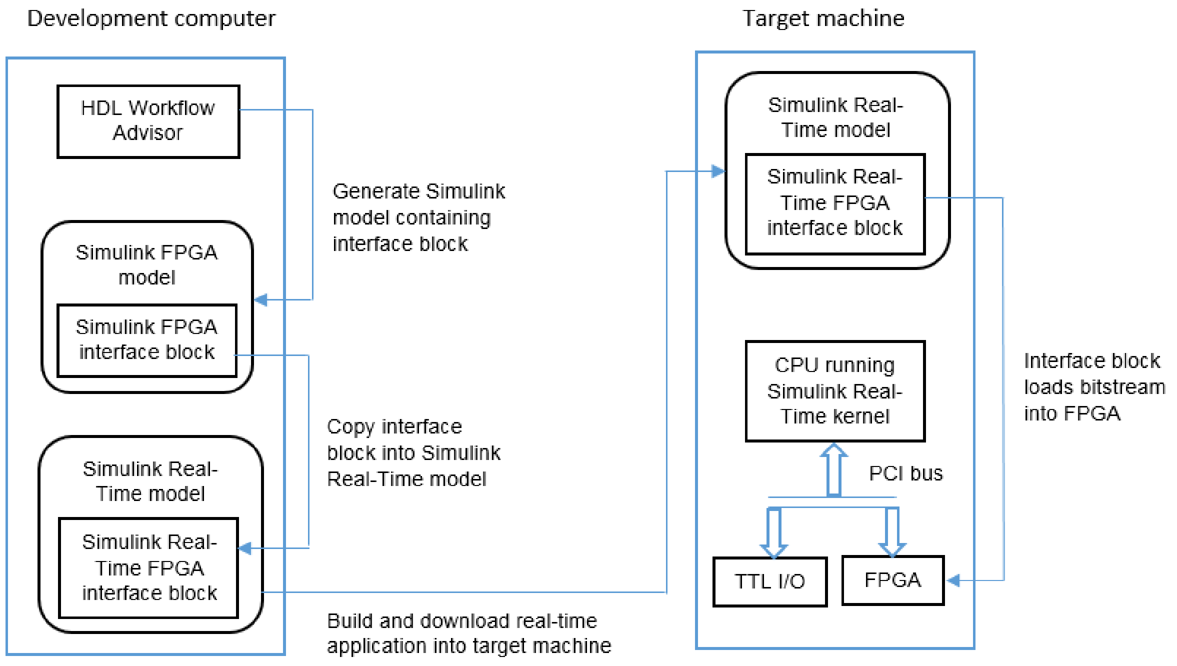
FPGA Programming and Configuration

This example shows how to implement a Simulink® algorithm on a Speedgoat FPGA I/O board by using HDL Workflow Advisor to:

1. Specify an FPGA board and its I/O interface.
2. Synthesize the Simulink algorithm for FPGA programming.
3. Generate a Simulink® Real-Time™ interface subsystem model.

The interface subsystem model contains blocks to program the FPGA and communicate with the FPGA I/O board during real-time application execution. You add the generated subsystem to your Simulink Real-Time domain model.

The entire workflow looks like this figure.


Development computer
Target machine
HDL Workflow Advisor
Simulink FPGA model
Simulink FPGA interface block
Simulink Real-Time model
Simulink Real-Time FPGA interface block
Generate Simulink model containing interface block
Copy interface block into Simulink Real-Time model
Build and download real-time application into target machine
Simulink Real-Time model
Simulink Real-Time FPGA interface block
CPU running Simulink Real-Time kernel
PCI bus
TTL I/O
FPGA
Interface block loads bitstream into FPGA

This example uses the Speedgoat IO331. You can use any FPGA I/O module supported by Simulink Real-Time and HDL Coder that meets the speed, size, and pinout requirements of the model.

Requirements and Preconditions

HDL Coder™

Before you start, complete an FPGA subsystem plan.

For the IO331 board, HDL Workflow Advisor requires the Xilinx® ISE toolset. To install this toolset, in the Command Window, type:

```
hdlsetuptoolpath('ToolName', 'Xilinx ISE', 'ToolPath', toolpath)
```

where `toolpath` is the full path to the synthesis tool executable.

For the toolset requirements of other boards, see Supported Third-Party Tools and Hardware (HDL Coder).

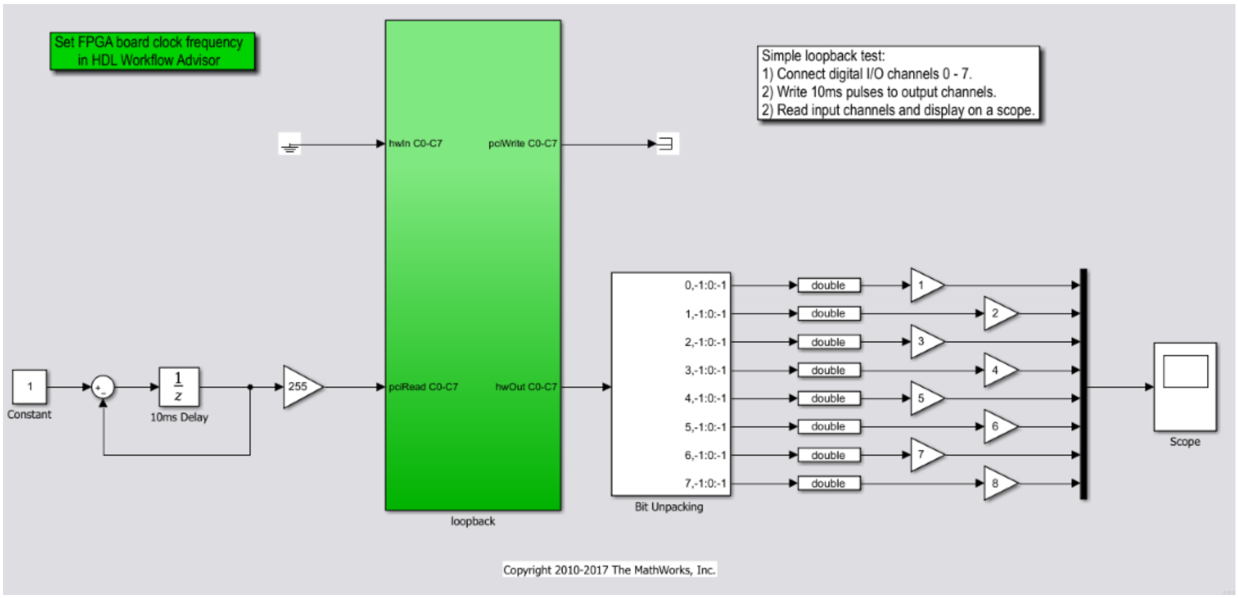
Step 1. Simulink Domain Model

The Simulink FPGA domain model contains a subsystem (algorithm) to be programmed onto the FPGA chip. Using this model, you can test your FPGA algorithm in a simulation environment before you download the algorithm to an FPGA board.

1. Create a Simulink model that contains the algorithm that you want to load onto the FPGA, in this case a loopback test.
2. Place the algorithm to be programmed on the FPGA inside a Subsystem block. The model can include other blocks and subsystems for testing. However, one subsystem must contain the FPGA algorithm.
3. Set or confirm the subsystem inport and outport names and data types. The HDL Coder HDL Workflow Advisor uses these settings for routing and mapping algorithm signals to I/O connector channels.
4. Save the model.

This model is your FPGA domain model. It represents the simulation sample rate of the clock on your FPGA board. For example, the Speedgoat IO331 has an onboard 125 MHz clock. One second of simulation equals 125e6 iterations of the model.

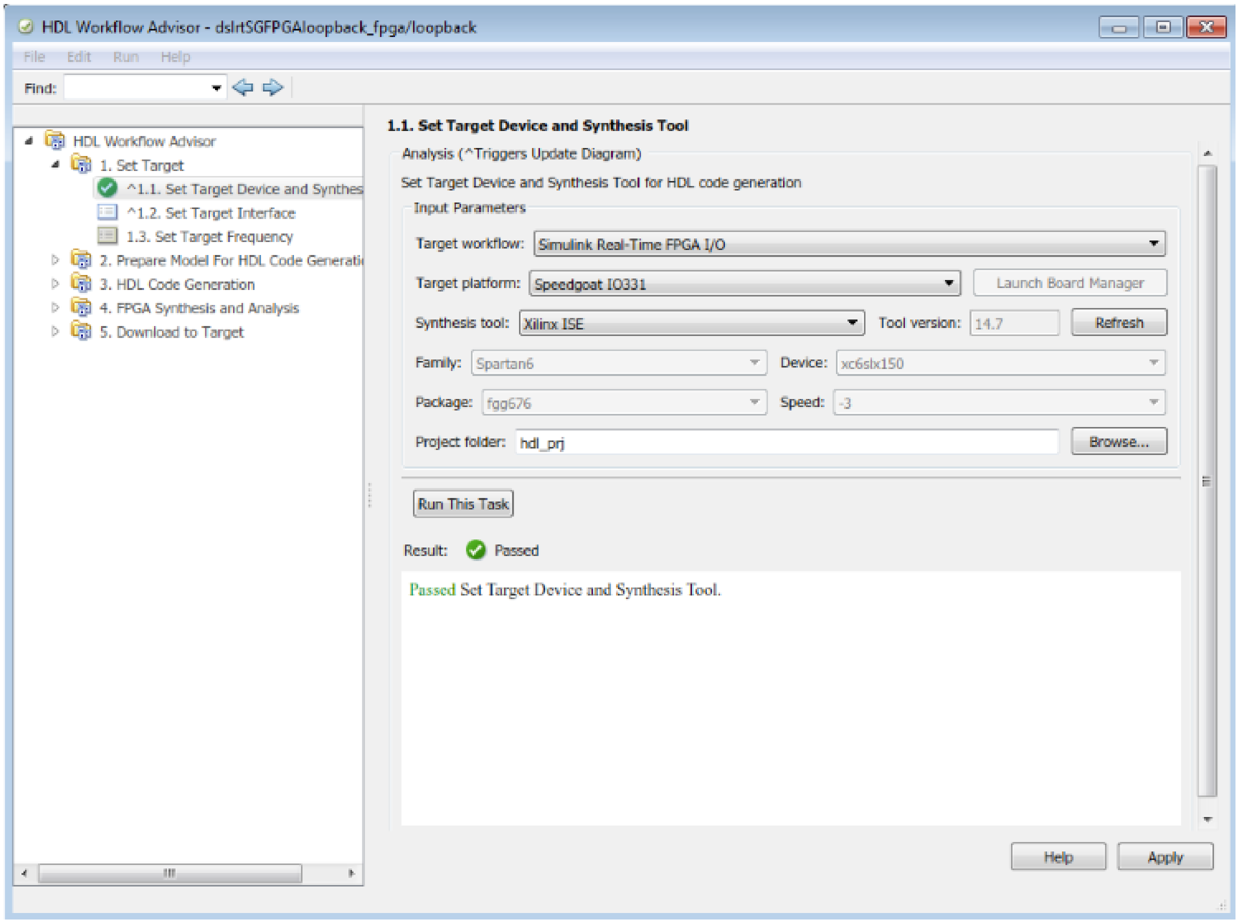
For an example of an FPGA domain model, see `dslrtSGFPGAloopback_fpga`. The `ServoSystem` subsystem contains the FPGA algorithm.


Set FPGA board clock frequency in HDL Workflow Advisor
Simple loopback test:
1) Connect digital I/O channels 0 - 7.
2) Write 10ms pulses to output channels.
2) Read input channels and display on a scope.
hwIn C0-C7
pciWrite C0-C7
pciRead C0-C7
hwOut C0-C7
loopback
Constant
10ms Delay
255
Bit Unpacking
double
Scope
Copyright 2010-2017 The MathWorks, Inc.

Step 2. FPGA Target Configuration

This procedure uses the `dslrtSGFPGAloopback_fpga` example. You must have already created an FPGA subsystem (algorithm) in an FPGA domain model and developed an FPGA subsystem plan.

1. Open the FPGA domain model `dslrtSGFPGAloopback_fpga`.
2. In the FPGA model, right-click the FPGA subsystem (`ServoSystem`). From the context menu, select **HDL Code** > **HDL Workflow Advisor**. The HDL Workflow Advisor dialog box displays several tasks for the subsystem. Address only your required subset of the tasks.
3. Expand the **Set Target** folder and select task **1.1 Set Target Device and Synthesis Tool**.
4. Set **Target Workflow** to `Simulink Real-Time FPGA I/O`.
5. From the **Target platform** list, select the Speedgoat FPGA I/O board installed in your Speedgoat target machine, in this case the `Speedgoat IO331`. Check that HDL Workflow advisor sets the synthesis tool to the Xilinx® ISE Design Suite.
6. Click **Run This Task**.


HDL Workflow Advisor - dslrtSGFPGAloopback_fpga/loopback
File Edit Run Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthes
^1.2. Set Target Interface
1.3. Set Target Frequency
2. Prepare Model For HDL Code Generati
3. HDL Code Generation
4. FPGA Synthesis and Analysis
5. Download to Target
Launch Board Manager
Refresh
Browse...
Run This Task
Help
Apply

1.1. Set Target Device and Synthesis Tool

Analysis (^Triggers Update Diagram)

Set Target Device and Synthesis Tool for HDL code generation

Input Parameters

Target workflow: Simulink Real-Time FPGA I/O

Target platform: Speedgoat IO331

Synthesis tool: Xilinx ISE Tool version: 14.7

Family: Spartan6 Device: xc6slx150

Package: fgg676 Speed: -3

Project folder: hdl_prj

Result: Passed

Passed Set Target Device and Synthesis Tool.

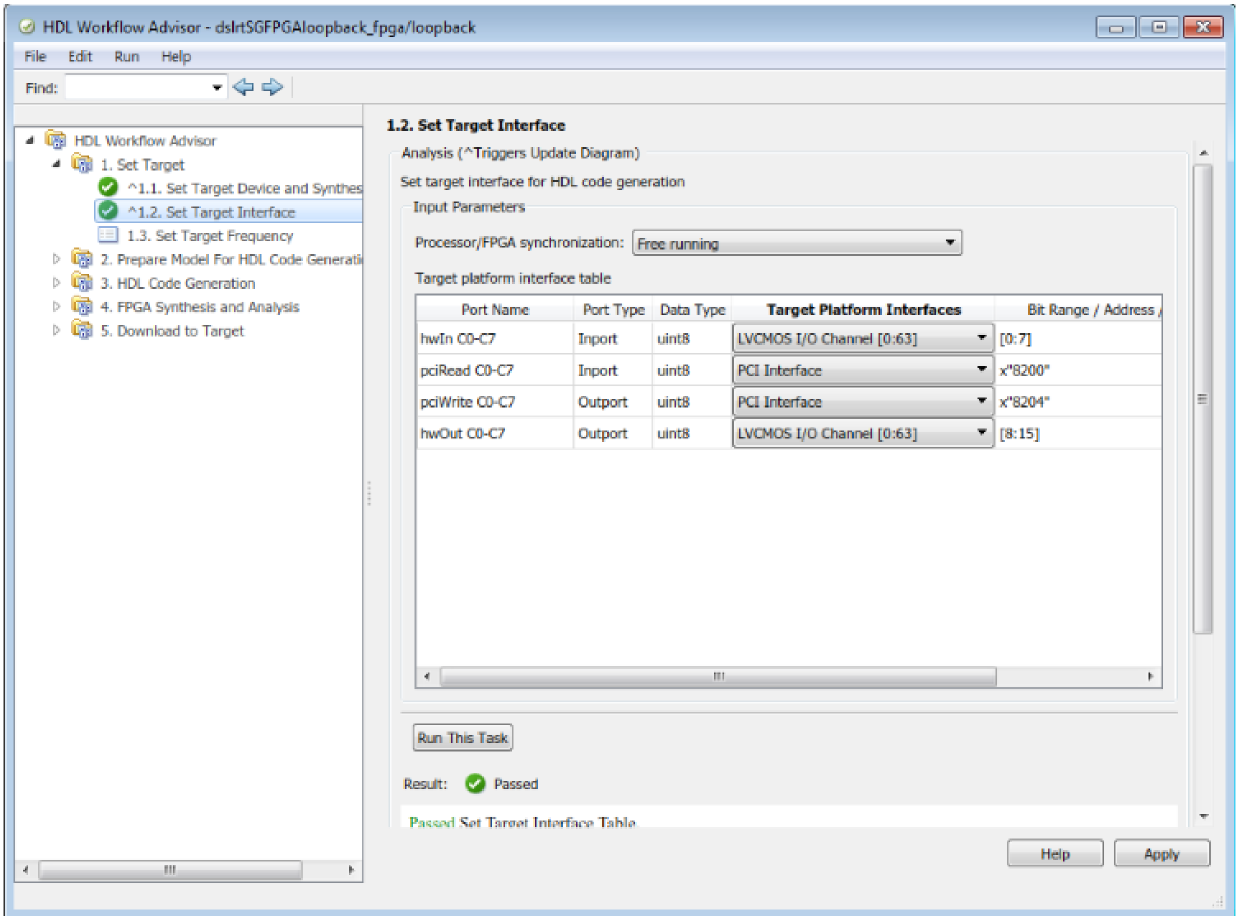
Step 3. FPGA Target Interface Configuration

You must have already configured the FPGA target.

1. In the **Set Target** folder, select task **1.2 Set Target Interface**.
2. In the **Processor/FPGA synchronization** box, select `Free running`.
3. For signals `hwIn` and `hwOut`, in the **Target Platform Interfaces** column, select `LVCMOS I/O Channel [0:63]`. In the **Bit Range/Address/FPGA Pin** column, enter the channel value for each signal, or take the defaults.

4 For signals `pciRead` and `pciWrite`, in the **Target Platform Interfaces** column, select `PCI Interface`. In the **Bit Range/Address/FPGA Pin** column, use the automatically generated values. Do not enter PCI address values.

5 Click **Run This Task**.


HDL Workflow Advisor - dslrtSGFPGAloopback_fpga/loopback

1.2. Set Target Interface

Analysis (^Triggers Update Diagram)

Set target interface for HDL code generation

Input Parameters

Processor/FPGA synchronization: Free running

Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address
hwIn C0-C7	Inport	uint8	LVCMOS I/O Channel [0:63]	[0:7]
pciRead C0-C7	Inport	uint8	PCI Interface	x"8200"
pciWrite C0-C7	Outport	uint8	PCI Interface	x"8204"
hwOut C0-C7	Outport	uint8	LVCMOS I/O Channel [0:63]	[8:15]

Result: Passed

Passed Set Target Interface Table

File
Edit
Run
Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthes
^1.2. Set Target Interface
1.3. Set Target Frequency
2. Prepare Model For HDL Code Generati
3. HDL Code Generation
4. FPGA Synthesis and Analysis
5. Download to Target
Run This Task
Help
Apply

Step 4. FPGA Target Frequency Configuration

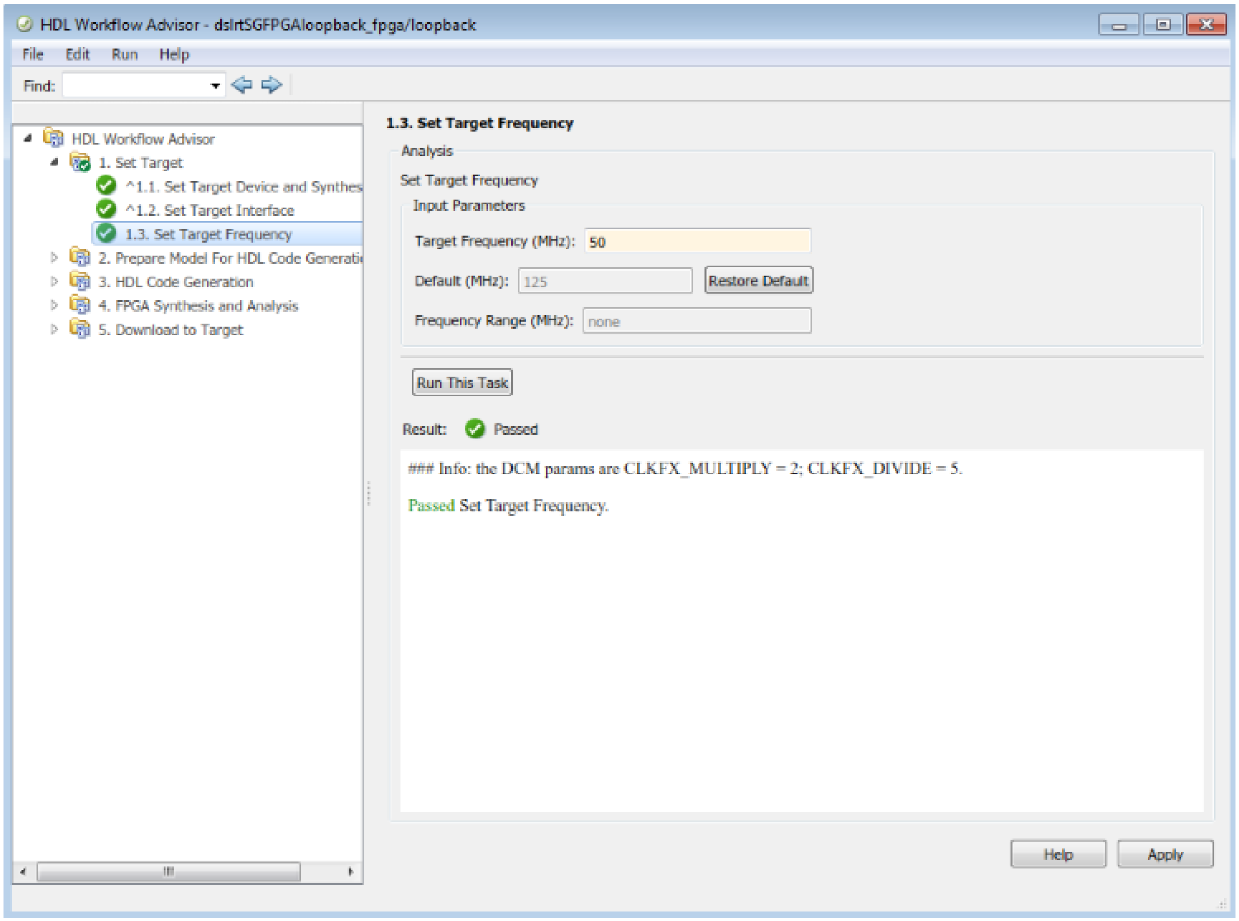
You must have already configured the FPGA target interface.

1 In the **Set Target** folder, select task **1.3 Set Target Frequency** (optional). The **Set Target Frequency** pane contains fields showing the FPGA input clock frequency

(fixed) and the FPGA system clock frequency. The FPGA system clock frequency defaults to the FPGA input clock frequency.

2 To specify a different system clock frequency (for example, 50 MHz), type the new value in the field **FPGA system clock frequency (MHz)**. For the permitted range for the system clock rate, see the Speedgoat board characteristics table. The system sometimes sets a value different from the one you specified.

3 Click **Run This Task**.


HDL Workflow Advisor - dslrtSGFPGAloopback_fpga/loopback
File
Edit
Run
Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthes
^1.2. Set Target Interface
1.3. Set Target Frequency
2. Prepare Model For HDL Code Generati
3. HDL Code Generation
4. FPGA Synthesis and Analysis
5. Download to Target
1.3. Set Target Frequency
Analysis
Set Target Frequency
Input Parameters
Restore Default
Run This Task
Help
Apply

Target Frequency (MHz): 50

Default (MHz): 125

Frequency Range (MHz): none

Result: Passed

```
### Info: the DCM params are CLKFX_MULTIPLY = 2; CLKFX_DIVIDE = 5.

Passed Set Target Frequency.
```

Step 5. Simulink Real-Time Interface Subsystem Generation

This procedure generates an interface subsystem file for the dxpcSGFPGAloopback_fpga example.

Assign distinct names to blocks that contain different HDL code. The name of the interface subsystem file is derived directly from the block name. If two blocks containing different HDL code have the same name, the names collide and one of the blocks gets the wrong code.

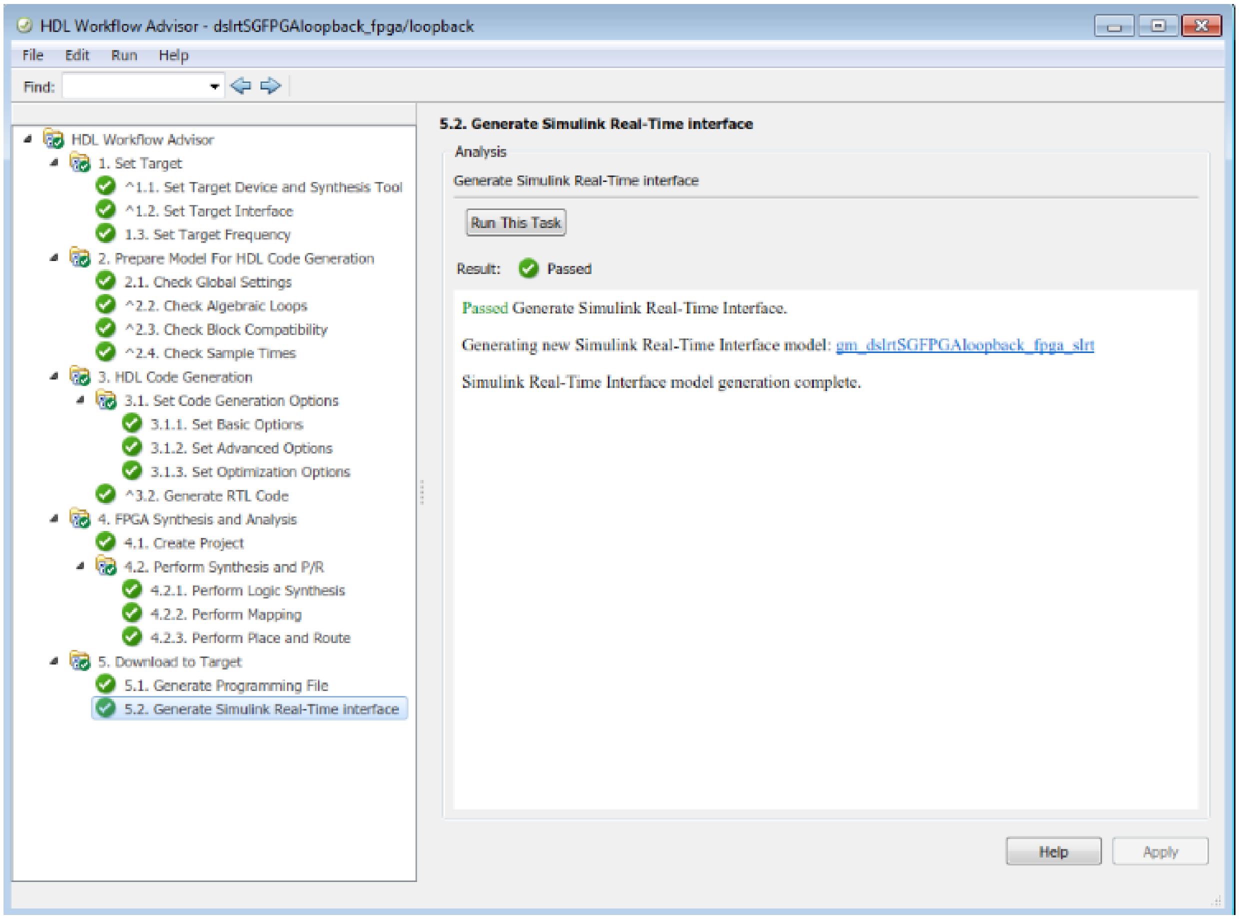
You must have already configured the FPGA target interface and the required target frequency. If you have specified vector inports or outports, you must have already selected the **Scalarize vector ports** check box. This check box is on the **Coding style** tab of node **Global Settings**, under node **HDL Code Generation** in the Configuration Parameters dialog box.

1. Expand the **Download to Target** folder, and right-click task **5.2 Generate Simulink Real-Time Interface**.
2. In this pane, click **Run To Selected Task**.

This action:

- Runs the remaining tasks.
- Creates the FPGA bitstream file in the `hdlsrc` folder. The Simulink Real-Time interface subsystem references this bitstream file during the build and download process.
- Generates a model named `gm_dslrtSGFPGAloopback_fpga_slrt`, which contains the Simulink Real-Time interface subsystem.

Here is an example of the HDL Coder HDL Workflow Advisor after this action.


HDL Workflow Advisor - dslrtSGFPGAloopback_fpga/loopback
File Edit Run Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis Tool
^1.2. Set Target Interface
1.3. Set Target Frequency
2. Prepare Model For HDL Code Generation
2.1. Check Global Settings
^2.2. Check Algebraic Loops
^2.3. Check Block Compatibility
^2.4. Check Sample Times
3. HDL Code Generation
3.1. Set Code Generation Options
3.1.1. Set Basic Options
3.1.2. Set Advanced Options
3.1.3. Set Optimization Options
^3.2. Generate RTL Code
4. FPGA Synthesis and Analysis
4.1. Create Project
4.2. Perform Synthesis and P/R
4.2.1. Perform Logic Synthesis
4.2.2. Perform Mapping
4.2.3. Perform Place and Route
5. Download to Target
5.1. Generate Programming File
5.2. Generate Simulink Real-Time interface
Run This Task
Help
Apply

5.2. Generate Simulink Real-Time interface

Analysis

Generate Simulink Real-Time interface

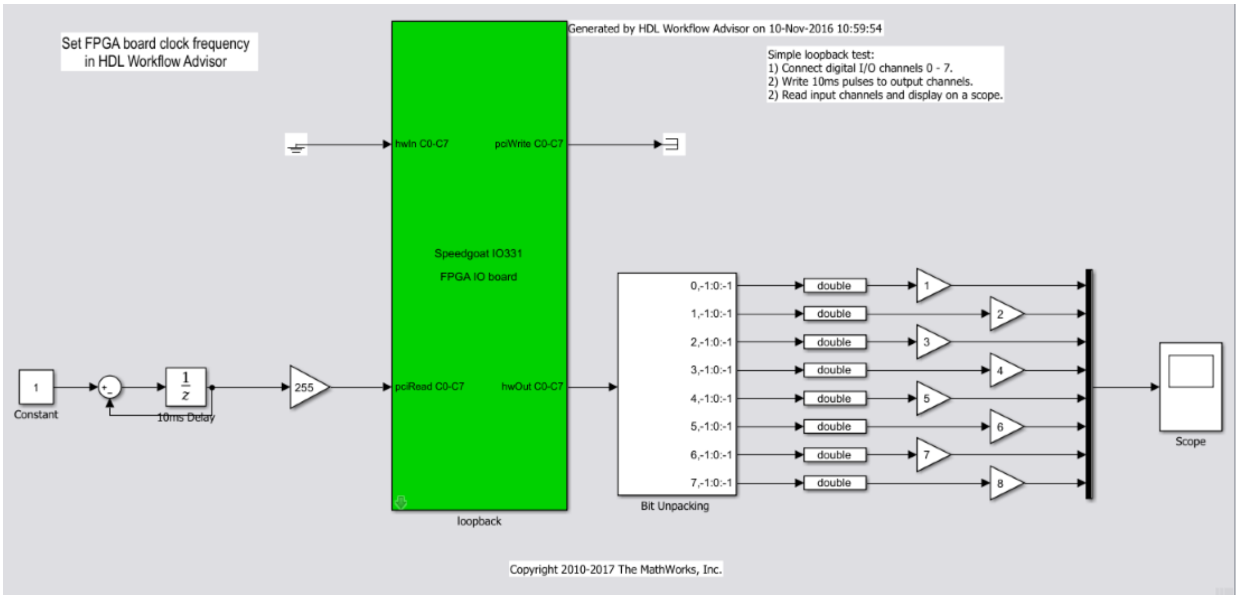
Result: ✓ Passed

Passed Generate Simulink Real-Time Interface.

Generating new Simulink Real-Time Interface model: gm_dslrtSGFPGAloopback_fpga_slrt

Simulink Real-Time Interface model generation complete.

The generated interface subsystem looks like this figure.


Set FPGA board clock frequency in HDL Workflow Advisor
Generated by HDL Workflow Advisor on 10-Nov-2016 10:59:54
Simple loopback test:
1) Connect digital I/O channels 0 - 7.
2) Write 10ms pulses to output channels.
2) Read input channels and display on a scope.
hwIn C0-C7
pciWrite C0-C7
Speedgoat IO331
FPGA IO board
pciRead C0-C7
hwOut C0-C7
loopback
1
Constant
1/z
10ms Delay
255
Bit Unpacking
double
Scope
Copyright 2010-2017 The MathWorks, Inc.

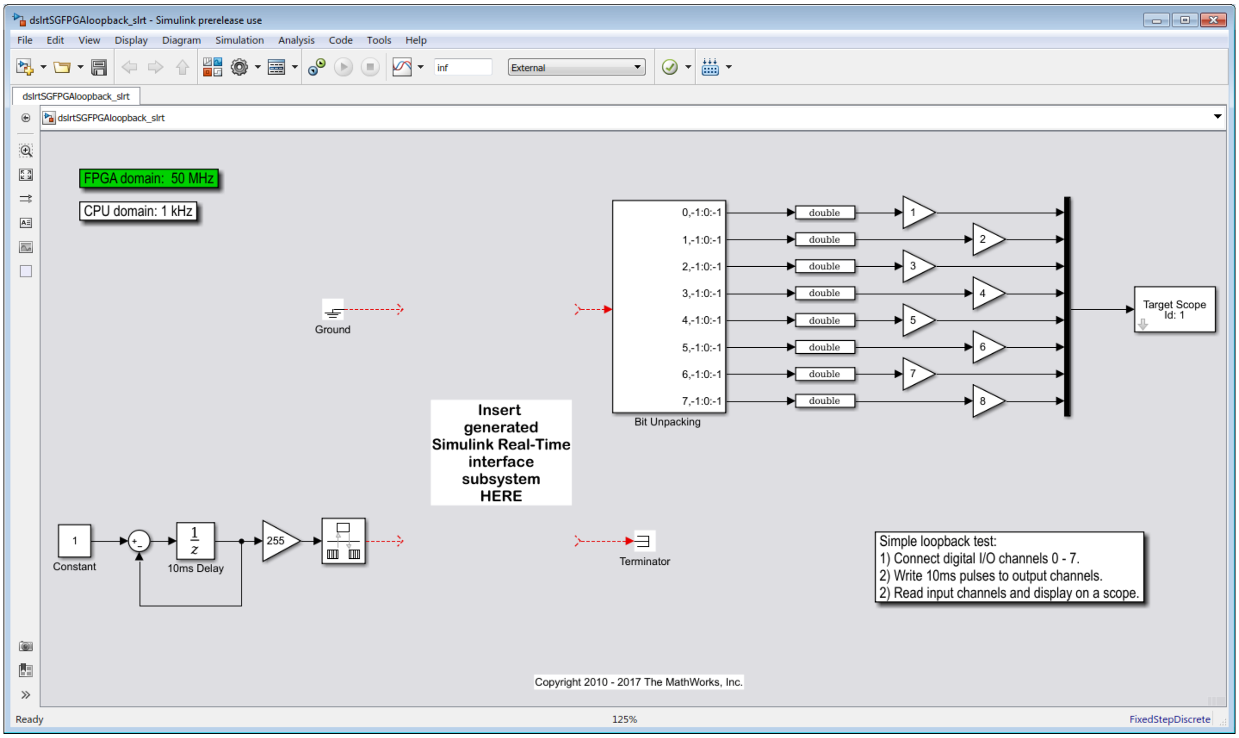
This generated model contains a masked subsystem with the same name as the subsystem in the Simulink FPGA domain model. Although the appearance is similar, this subsystem does not contain the Simulink algorithm. Instead, the algorithm is implemented in an FPGA bitstream. You reference and load this algorithm into the FPGA from this subsystem.

Step 6. Simulink Real-Time Domain Model

Using the Simulink Real-Time software, transform a Simulink or Stateflow® domain model into a Simulink Real-Time domain model and execute it on a Speedgoat target machine for real-time testing applications. After creating a Speedgoat FPGA interface subsystem. You can then include the FPGA board in your Simulink Real-Time domain model by inserting the interface subsystem.

1. Create a Simulink Real-Time domain model with the functionality that you want to simulate with the FPGA algorithm. Leave the inports and outports of the FPGA subsystem disconnected.
2. Save the model.

The Simulink Real-Time domain model looks like this figure. See example model `dslrtSGFPGAloopback_slrt`.


dslrtSGFPGAloopback_slrt - Simulink prerelease use
File Edit View Display Diagram Simulation Analysis Code Tools Help
inf
External
dslrtSGFPGAloopback_slrt
FPGA domain: 50 MHz
CPU domain: 1 kHz
Ground
Insert generated Simulink Real-Time interface subsystem HERE
Bit Unpacking
0,-1:0:-1
1,-1:0:-1
2,-1:0:-1
3,-1:0:-1
4,-1:0:-1
5,-1:0:-1
6,-1:0:-1
7,-1:0:-1
double
Target Scope Id: 1
Constant
10ms Delay
255
Terminator
Simple loopback test:
1) Connect digital I/O channels 0 - 7.
2) Write 10ms pulses to output channels.
2) Read input channels and display on a scope.
Copyright 2010 - 2017 The MathWorks, Inc.
Ready
125%
FixedStepDiscrete

Step 7. Simulink Real-Time Interface Subsystem Integration

In the Simulink Real-Time interface subsystem mask, set three parameters:

- Device index
- PCI slot
- Sample time

To integrate the interface subsystem:

1. In the Simulink editor, open `gm_dslrtSGFPGAloopback_fpga_slrt`.
2. Copy the Simulink Real-Time interface subsystem and paste it into the Simulink Real-Time domain model.
3. Save or discard `gm_dslrtSGFPGAloopback_fpga_slrt`. You can recreate it as required using the HDL Coder HDL Workflow Advisor.

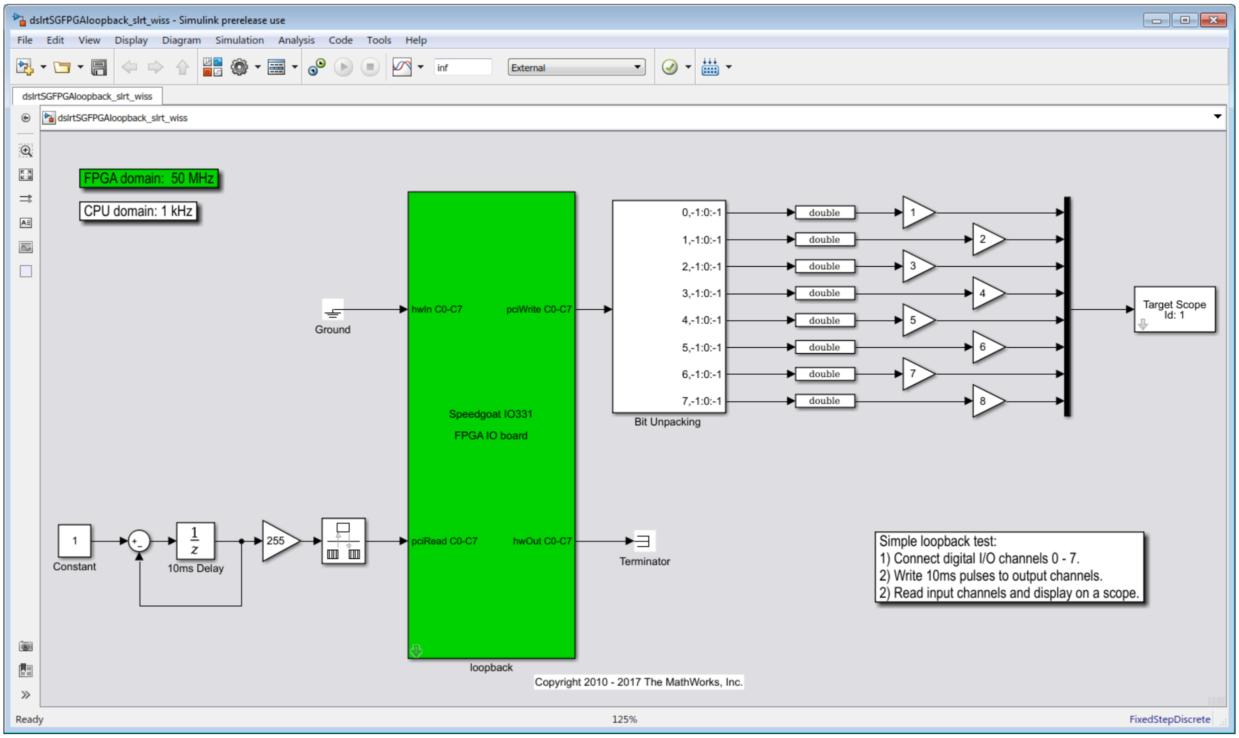
4 In the domain model, connect signals to the inports and outports of the interface subsystem.

5 Set the block parameters according to the FPGA I/O boards in your Speedgoat target machine.

- If you have a single FPGA I/O board, leave the device index and PCI slot at the default values. You can set the sample time or leave it at –1 for inheritance.
- If you have multiple FPGA I/O boards, give each board a unique device index.
- If you have two or more boards of the same type (for example, two Speedgoat IO331 boards), specify the PCI slot ([bus, slot]) for each board. Get this information with the SimulinkRealTime.target.getPCIInfo function.

6 Save the model.

The updated Simulink Real-Time domain model looks like this figure. See example model `dslrtSGFPGAloopback_slrt_wiss`.



Simple loopback test:
1) Connect digital I/O channels 0 - 7.
2) Write 10ms pulses to output channels.
2) Read input channels and display on a scope.

Step 8. Real-Time Application Execution

To do this procedure, you must have already created a Simulink Real-Time domain model that includes a Simulink Real-Time interface subsystem generated from the HDL Coder HDL Workflow Advisor.

1. Configure the Speedgoat target machine and connect it to the development computer.
2. Build and download the Simulink Real-Time application. The real-time application loads onto the Speedgoat target machine and the FPGA algorithm bitstream loads onto the FPGA.
3. If you are using I/O lines (channels), confirm that you have connected the lines to the external hardware under test.

The start and stop of the Simulink Real-Time model controls the start and stop of the FPGA algorithm. The FPGA algorithm executes at the clock frequency of the FPGA I/O board, while the real-time application executes in accordance with the model sample time.

Related Examples

- “Processor and FPGA Synchronization” on page 26-16
- “IP Core Generation Workflow for Speedgoat Boards” on page 27-60
- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 27-2

Model Design for AXI4-Stream Video Interface Generation

In this section...

With the HDL Coder software, you can implement a simplified, streaming pixel protocol in your model. The software generates an HDL IP core with AXI4-Stream Video interfaces.

Streaming Pixel Protocol

You can use the streaming pixel protocol for AXI4-Stream Video interface mapping. Video algorithms process data serially and generate video data as a serial stream of pixel data and control signals. To learn about the streaming pixel protocol, see "Streaming Pixel Interface" (Vision HDL Toolbox) in the Vision HDL Toolbox documentation.

To generate an IP core with AXI4-Stream Video interfaces, in your DUT interface, implement these signals:

- Pixel Data
- Pixel Control Bus

The **Pixel Control Bus** is a bus that has these signals:

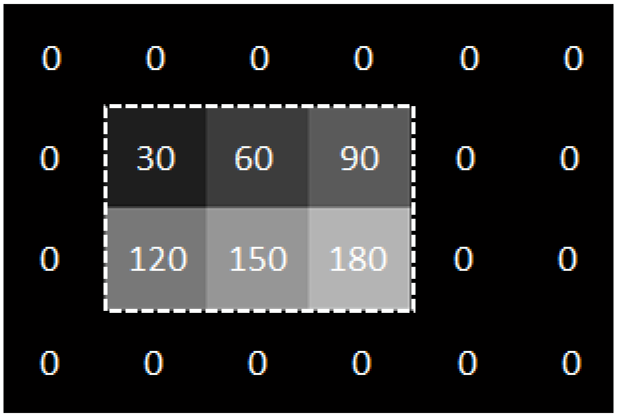
- `hStart`
- `hEnd`
- `vStart`
- `vEnd`
- `valid`

The signals **hStart** and **hEnd** represent the start of an active line and the end of an active line respectively. The signals **vStart** and **vEnd** represent the start of a frame and the end of a frame.

You can optionally model the backpressure signal, **Ready**, and map it to the AXI4-Stream Video interface.

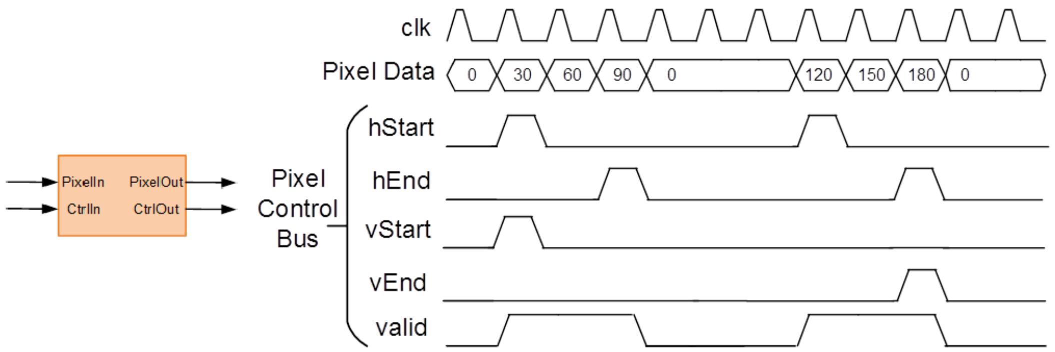
Protocol Signals and Timing Diagrams

This figure is a 2–by–3 pixel image. The active image area is the rectangle with a dashed line around it and the inactive pixels that surround it. The pixels are labeled with their grayscale values.



Pixel Data and Pixel Control Bus

This figure shows the timing diagram for the **Pixel Data** and **Pixel Control Bus** signals that you model at the DUT interface.



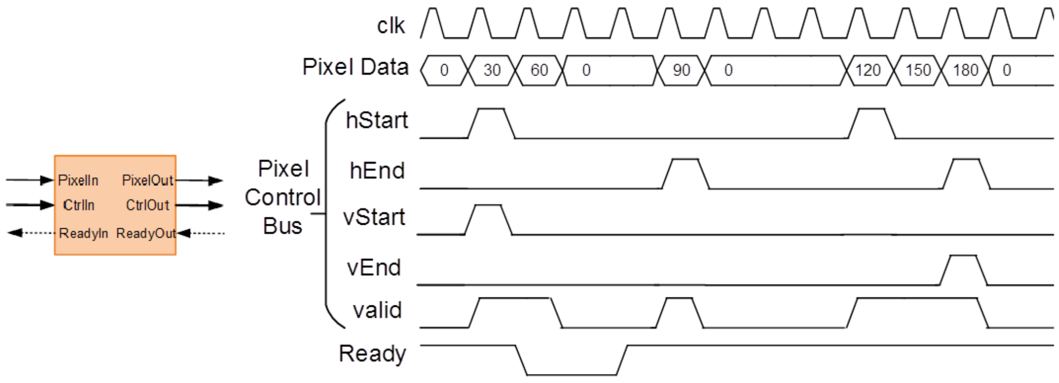
The **Pixel Data** signal is the primary video signal that is transferred across the AXI4-Stream Video interface. When the **Pixel Data** signal is valid, the **valid** signal is asserted.

The **hStart** signal becomes high at the start of the active lines. The **hEnd** signal becomes high at the end of the active lines.

The **vStart** signal becomes high at the start of the active frame in the second line. The **vEnd** signal becomes high at the end of the active frame in the third line.

Optional Ready Signal

This figure shows the timing diagram for the **Pixel Data**, the **Pixel Control Bus**, and the **Ready** signal that you model at the DUT interface.



When you map the DUT ports to an AXI4-Stream Video interface, you can optionally model the backpressure signal, **Ready**, and map it to the AXI4-Stream Video interface.

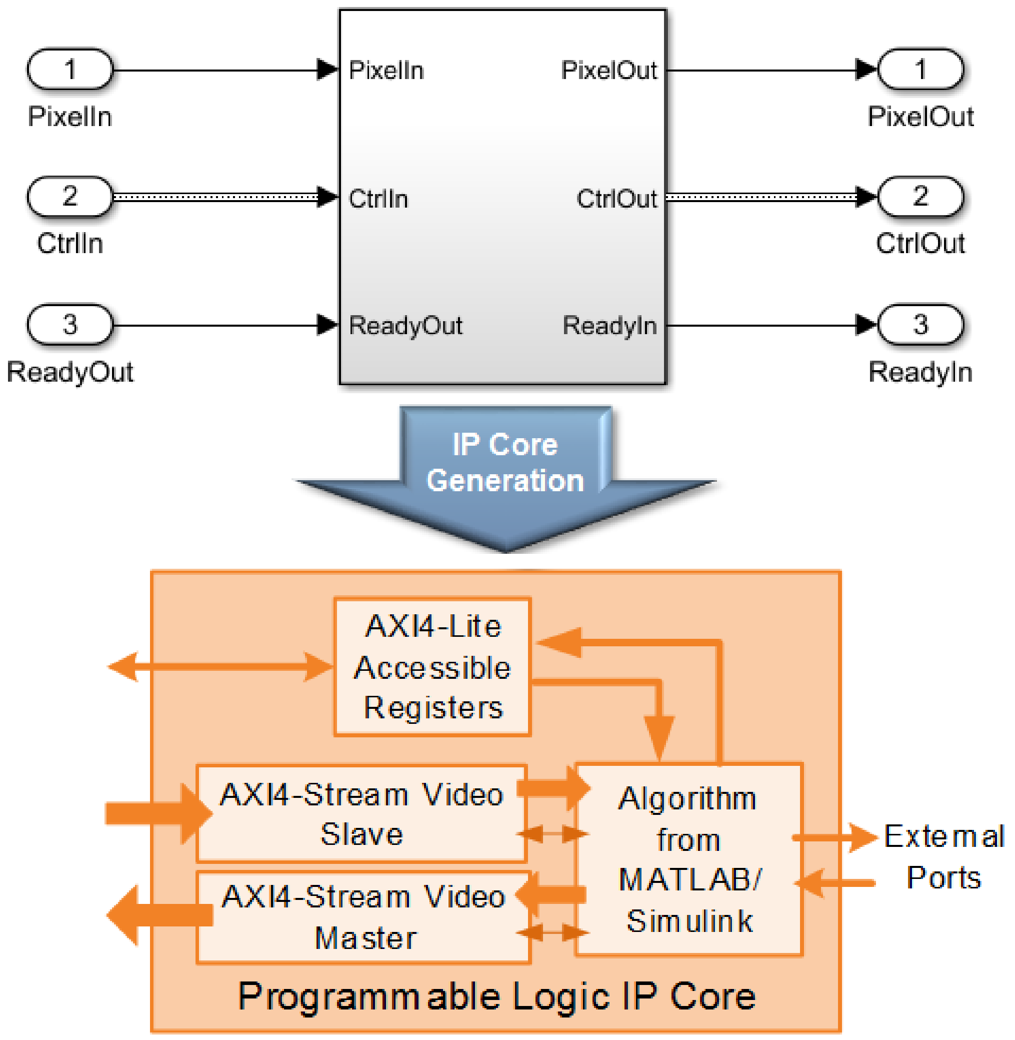
In a Slave interface, with the **Ready** signal, you can apply back pressure. In a Master interface, with the **Ready** signal, you can respond to back pressure.

If you model the **Ready** signal in your AXI4-Stream Video interfaces, your Master interface must deassert its **valid** signal one cycle after the **Ready** signal is deasserted.

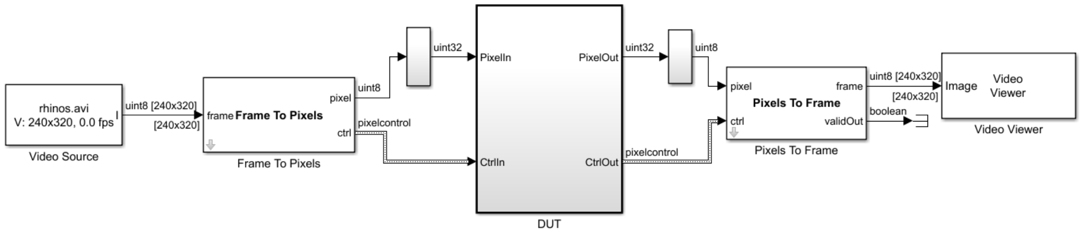
If you do not model the **Ready** signal, HDL Coder generates the associated backpressure logic.

Model Data and Control Bus Signals

You can model your video algorithm with **Pixel Data** and **Pixel Control Bus** signals at the DUT ports and map the signals to AXI4-Stream Video interfaces. You can optionally model the backpressure signal, **Ready**, and map it to the AXI4-Stream Video interface.



This figure shows an example of a top-level Simulink model with a Video Source input.

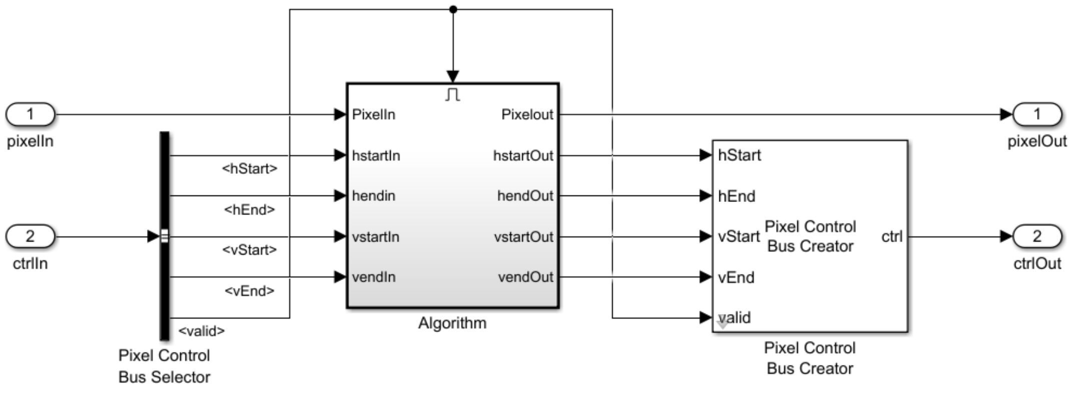


The Frame To Pixels and Pixels To Frame blocks perform the conversion between the video frames and the **Pixel Data** and **Pixel Control Bus** at the DUT interface. To use these blocks, you must have the Vision HDL Toolbox installed.

See also Frame To Pixels and Pixels To Frame in the Vision HDL Toolbox documentation.

Pixel Data and Pixel Control Bus Modeling

This figure shows how to model the **Pixel Data** and **Pixel Control Bus** signals inside the **DUT** subsystem.



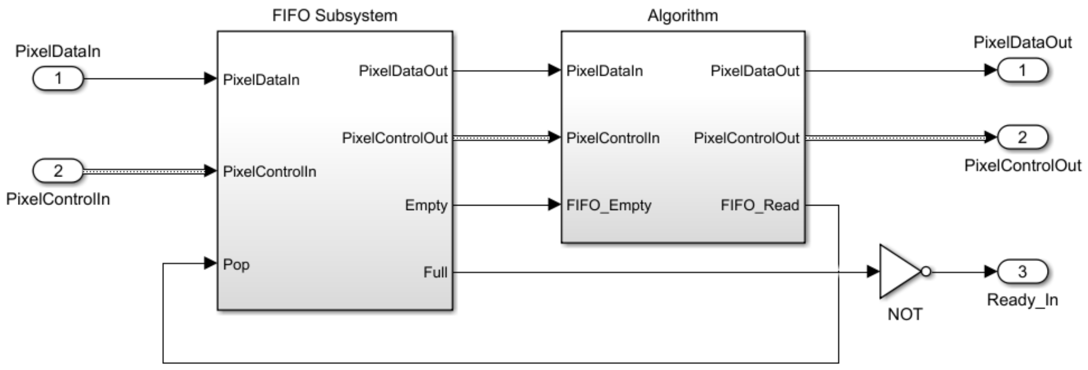
You can directly connect the **valid** signal from the **Pixel Control Bus** to the Enable port. If you do not have the Vision HDL Toolbox software, replace the Pixel Control Bus Selector and Pixel Control Bus Creator blocks with the Bus Selector and Bus Creator blocks respectively.

Ready Signal Modeling

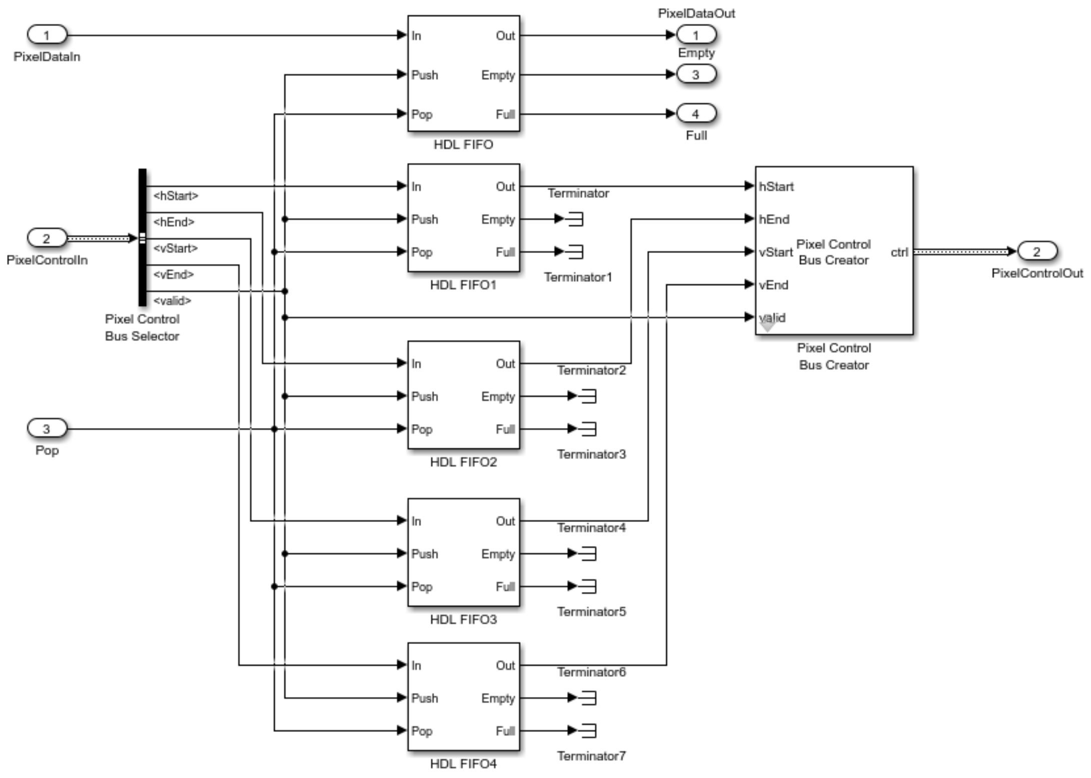
The AXI4-Stream Video interfaces in your DUT can optionally include a **Ready** signal.

For example, you can have a FIFO in your DUT to store some video data before processing the signals. Use a **FIFO Subsystem** that contains HDL FIFO blocks to store the **Pixel Data** and the **Pixel Control Bus** signals. To apply the backpressure to the upstream component, model the **Ready** signal based on the FIFO Full signal.

This figure shows how to model the **Ready** signal inside the **DUT** subsystem.



The **FIFO Subsystem** block uses HDL FIFO blocks for the **Pixel Data** and for the **Pixel Control Bus** signals.

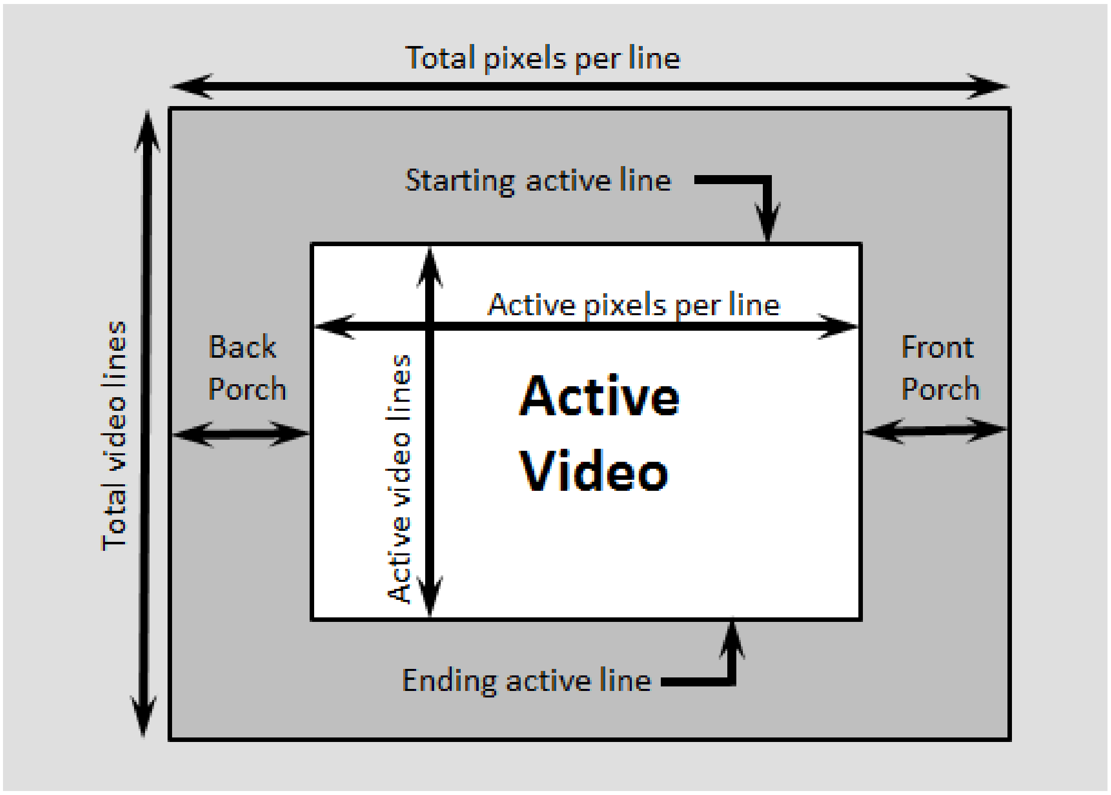


Disable delay balancing for the **Ready** signal path. If you enable delay balancing, the coder can insert one or more delays on the **Ready** signal.

Video Porch Insertion Logic

Video capture systems scan video signals from left to right and from top to bottom. As these systems scan, they generate inactive intervals between lines and frames of active video. This inactive interval is called a video porch. The horizontal porch consists of inactive cycles between the end of one line and the beginning of next line. The vertical porch consists of inactive cycles between the ending active line of one frame and the starting active line of next frame.

This figure shows a video frame with the horizontal porch split into a front and a back porch.



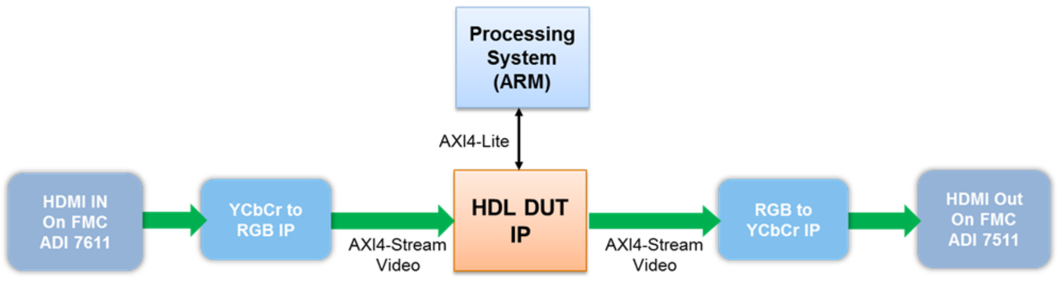
The AXI4-Stream Video interface does not require a video porch, but Vision HDL Toolbox algorithms require a porch for processing video streams. If the incoming pixel stream does not have a sufficient porch, HDL Coder inserts the required amount of porch to the pixel stream. By using the AXI4-Lite registers in the generated IP core, you can customize these porch parameters for each video frame:

- Active pixels per line (Default: 1920)
- Active video lines: (Default: 1080)
- Horizontal porch length (Default: 280)
- Vertical porch length (Default: 45)

Default Video System Reference Design

You can integrate the generated HDL IP core with AXI4-Stream Video interfaces into the `Default video system` reference design.

This figure is a block diagram of the `Default video system` reference design architecture.



You can use this `Default video system` reference design architecture with these target platforms:

- `Xilinx Zynq ZC702 evaluation kit`
- `Xilinx Zynq ZC706 evaluation kit`
- `ZedBoard`

To use the `Default video system` reference design, you must install the Computer Vision System Toolbox™ Support Package for Xilinx Zynq-Based Hardware.

Restrictions

When you map the DUT ports to AXI4-Stream Video interfaces:

- The video algorithm must be single rate.
- The DUT port mapped to the **Pixel Data** signal must use a scalar data type.
- The DUT can have at most one AXI4-Stream Video Master channel and one AXI4-Stream Video Slave channel.
- Xilinx Zynq-7000 must be your target platform.

- You must use Xilinx Vivado as your synthesis tool.
- **Processor/FPGA synchronization** must be `Free running`.

More About

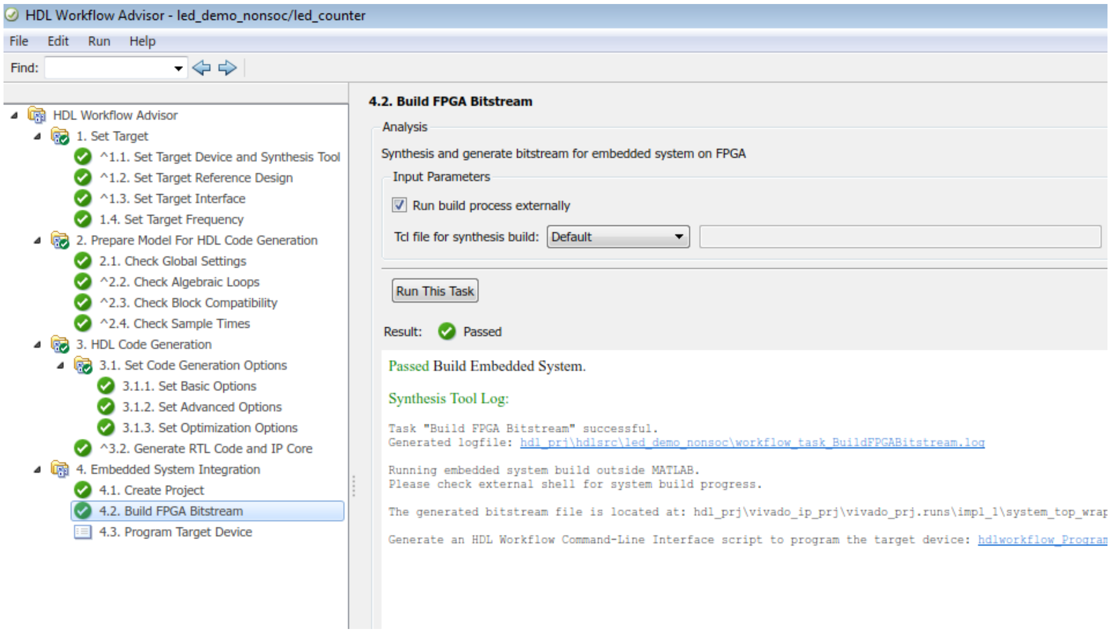
- “Model Design for AXI4-Stream Interface Generation” on page 27-11
- “Streaming Pixel Interface” (Vision HDL Toolbox)

IP Core Generation Workflow for Standalone FPGA Devices

In this section...

You can generate a reusable HDL IP core for any supported Xilinx or Altera FPGA device. The workflow produces an IP core report that displays the target interface configuration and the coder settings that you specify. See "Custom IP Core Generation" on page 26-2.

You can optionally build your own custom reference designs and integrate the generated IP core into the reference design. The workflow does not require the Embedded Coder software, because you need not generate the embedded code that is run on the processor. This means that the workflow does not have a **Generate Software Interface Model** task.


HDL Workflow Advisor - led_demo_nonsoc/led_counter
File Edit Run Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis Tool
^1.2. Set Target Reference Design
^1.3. Set Target Interface
1.4. Set Target Frequency
2. Prepare Model For HDL Code Generation
2.1. Check Global Settings
^2.2. Check Algebraic Loops
^2.3. Check Block Compatibility
^2.4. Check Sample Times
3. HDL Code Generation
3.1. Set Code Generation Options
3.1.1. Set Basic Options
3.1.2. Set Advanced Options
3.1.3. Set Optimization Options
^3.2. Generate RTL Code and IP Core
4. Embedded System Integration
4.1. Create Project
4.2. Build FPGA Bitstream
4.3. Program Target Device
Run This Task

4.2. Build FPGA Bitstream

Analysis

Synthesis and generate bitstream for embedded system on FPGA

Input Parameters

☑ Run build process externally

Tcl file for synthesis build: Default

Result: ✓ Passed

Passed Build Embedded System.

Synthesis Tool Log:

```
Task "Build FPGA Bitstream" successful.
Generated logfile: hdl_prj\hdlsrc\led_demo_nonsoc\workflow_task_BuildFPGABitstream.log

Running embedded system build outside MATLAB.
Please check external shell for system build progress.

The generated bitstream file is located at: hdl_prj\vivado_ip_prj\vivado_prj.runs\impl_1\system_top_wrap

Generate an HDL Workflow Command-Line Interface script to program the target device: hdlworkflow_Program
```

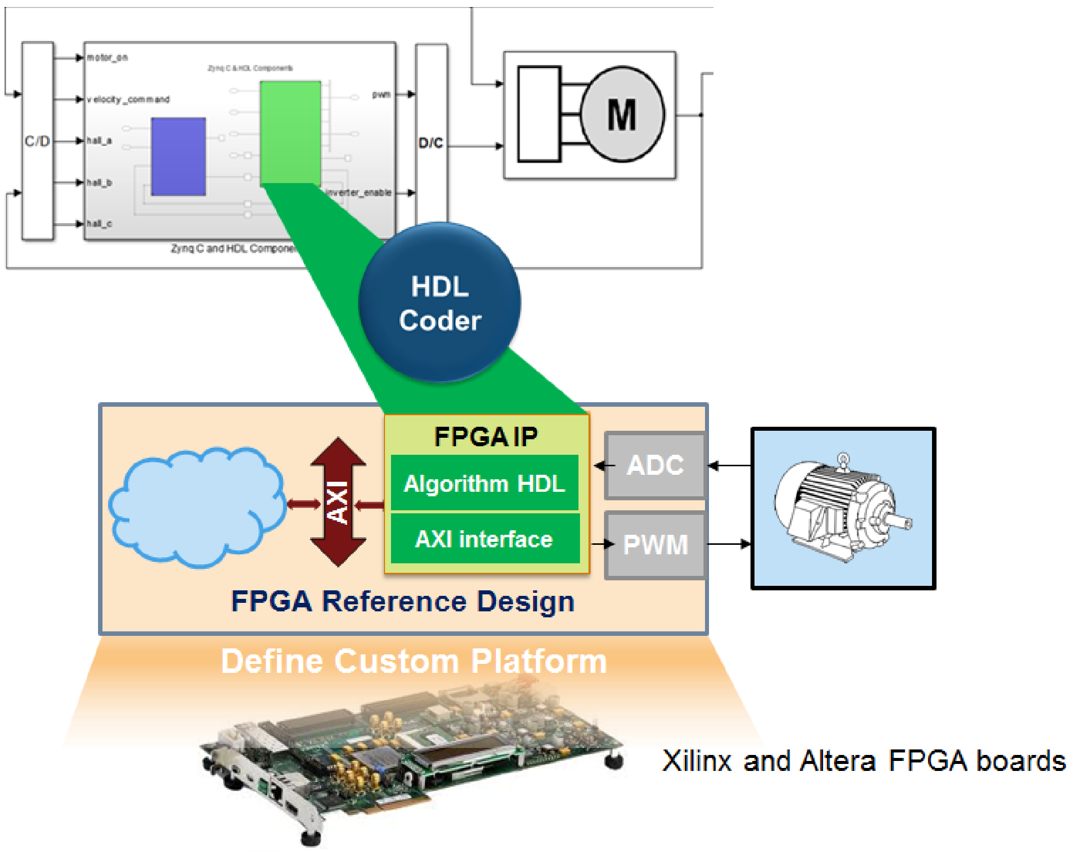
The workflow for the FPGA boards has these features:

- **Set Target Reference Design** task. Populates the reference design, its tool version, and the parameters that you specify.
- **Set Target Interface** task. Map your DUT ports to the interfaces on the target platform.
- **Set Target Frequency** task. Specifies the **Target Frequency (MHz)** to modify the clock module in the reference design to produce a clock signal with that frequency.
- **Generate RTL Code and IP Core** task. Generates a reusable and sharable IP core. The IP core packages the RTL code, a C header file, and the IP core definition files.
- **Create Project** task. Creates a project for integrating the IP core into the predefined reference designs.

You can generate an IP core with an optional AXI4 or an AXI4-Lite interface.

Targeting FPGA Reference Designs with AXI4 Interface

This figure shows how HDL Coder generates an IP core with an AXI4 interface and integrates the IP core into the FPGA reference design. See “Board and Reference Design Registration System” on page 27-17.



Use the HDL Coder generated AXI4-Lite interface to connect the IP core with an AXI4 or AXI4-Lite Master device such as:

- MicroBlaze processor.
- Nios II processor.
- PCIe Endpoint that connects to an external processor.
- JTAG Master.

When you connect the HDL IP core to a processor such as the MicroBlaze, you must integrate the handwritten C code to run on the processor. The generated IP core report displays the register address mapping information. To find the register offsets in the IP core register space, use this mapping information. To get the memory address of each register, add the register offset to the base address that you specify in your reference design. You can also find the register offsets in the C header file in the generated IP core folder.

Targeting FPGA Reference Designs Without AXI4 Interface

In the reference design definition function, you can create your own custom reference designs without the AXI4 slave interface. See also addAXI4SlaveInterface (hdlcoder.ReferenceDesign).

When creating a custom reference design, to target a standalone FPGA board, use the `EmbeddedCoderSupportPackage` method of the `hdlcoder.ReferenceDesign` class:

```
hRD.EmbeddedCoderSupportPackage = hdlcoder.EmbeddedCoderSupportPackage.None;
```

See EmbeddedCoderSupportPackage (hdlcoder.ReferenceDesign).

Board Support

HDL Coder supports these FPGA boards with the `IP Core Generation` workflow:

- `Xilinx Kintex-7 KC705 development board`
- `Arrow DECA MAX 10 FPGA evaluation kit`

Using these boards, you can integrate the generated IP core into the `default system` reference design. By default, this reference design does not have an AXI4 slave interface. Optionally, you can add the interface in the reference design definition function.

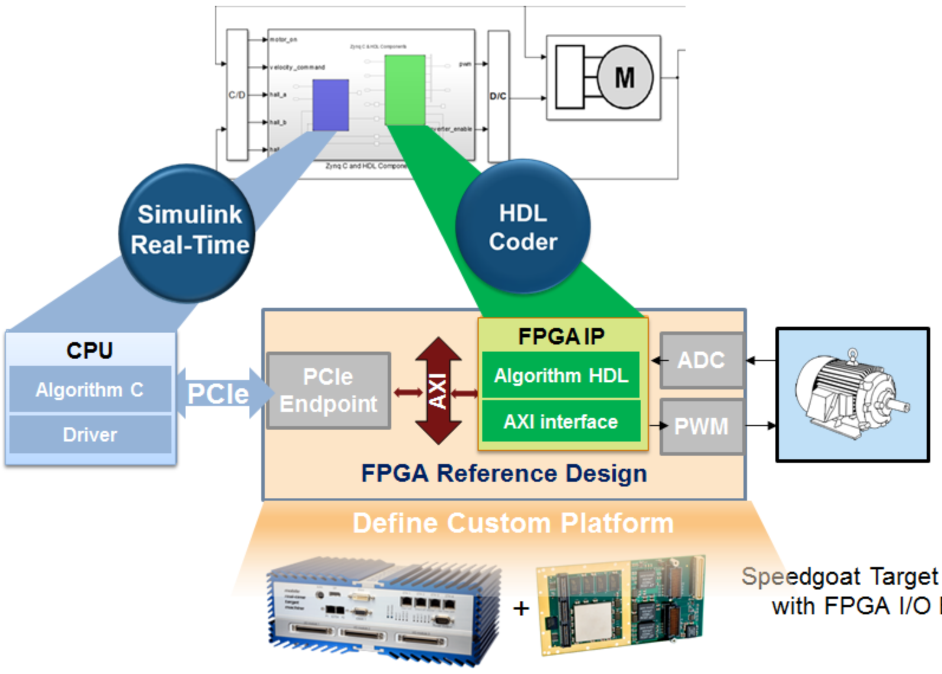
Related Examples

-

IP Core Generation Workflow for Speedgoat Boards

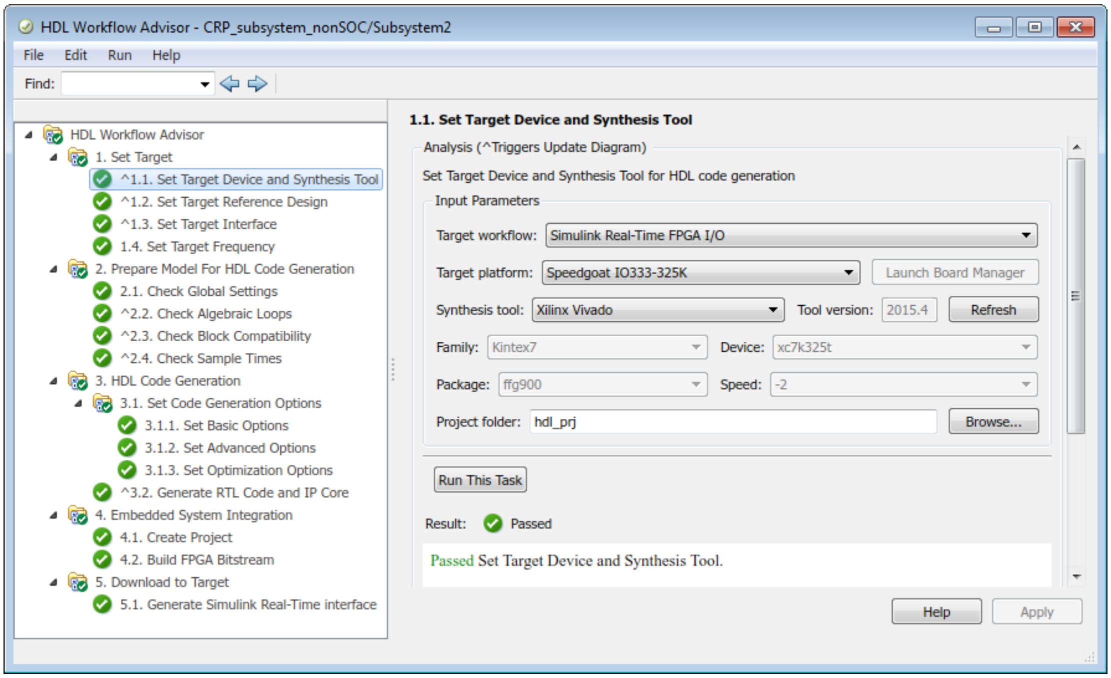
For the Speedgoat boards that support Xilinx Vivado, HDL Coder uses the `IP Core Generation` workflow infrastructure to generate a reusable HDL IP core. The workflow produces an IP core report that displays the target interface configuration and the coder settings that you specify. You can integrate the IP core into a larger design by adding it in an embedded system integration environment. See “Custom IP Core Generation” on page 26-2.

This figure shows how the software generates an IP core with an AXI interface and integrates the IP core into the FPGA reference design.



HDL Coder supports the `Speedgoat IO333—325K` board with the `Simulink Real-Time FPGA I/O` workflow. This workflow uses the `IP Core Generation` workflow infrastructure and has these key features:

- Support for Xilinx Vivado as the synthesis tool.
- Generates a reusable and sharable IP core. The IP core packages the RTL code, a C header file, and the IP core definition files.
- Creates a project for integrating the IP core into the Speedgoat reference design.
- Generates an FPGA bitstream and downloads the bitstream to the target hardware.


HDL Workflow Advisor - CRP_subsystem_nonSOC/Subsystem2
File
Edit
Run
Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis Tool
^1.2. Set Target Reference Design
^1.3. Set Target Interface
1.4. Set Target Frequency
2. Prepare Model For HDL Code Generation
2.1. Check Global Settings
^2.2. Check Algebraic Loops
^2.3. Check Block Compatibility
^2.4. Check Sample Times
3. HDL Code Generation
3.1. Set Code Generation Options
3.1.1. Set Basic Options
3.1.2. Set Advanced Options
3.1.3. Set Optimization Options
^3.2. Generate RTL Code and IP Core
4. Embedded System Integration
4.1. Create Project
4.2. Build FPGA Bitstream
5. Download to Target
5.1. Generate Simulink Real-Time interface
Launch Board Manager
Refresh
Browse...
Run This Task
Help
Apply

1.1. Set Target Device and Synthesis Tool

Analysis (^Triggers Update Diagram)

Set Target Device and Synthesis Tool for HDL code generation

Input Parameters

Target workflow: Simulink Real-Time FPGA I/O

Target platform: Speedgoat IO333-325K

Synthesis tool: Xilinx Vivado — Tool version: 2015.4

Family: Kintex7 — Device: xc7k325t

Package: ffg900 — Speed: -2

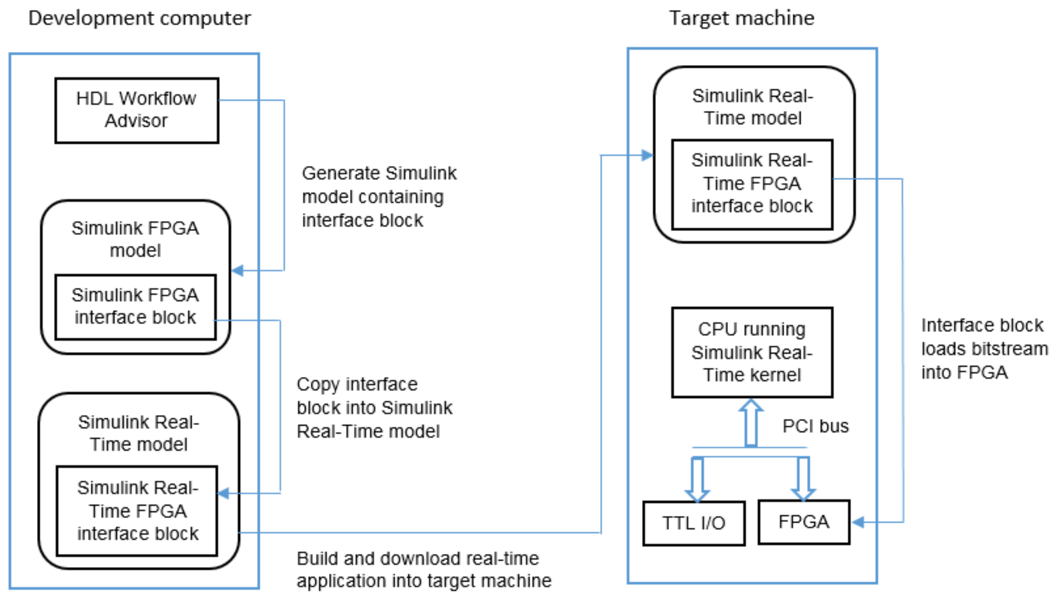
Project folder: hdl_prj

Result: ✓ Passed

Passed Set Target Device and Synthesis Tool.

To learn more about the board and its interfaces, see Speedgoat IO333.

After building the FPGA bitstream, the workflow generates 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.


Development computer
HDL Workflow Advisor
Generate Simulink model containing interface block
Simulink FPGA model
Simulink FPGA interface block
Copy interface block into Simulink Real-Time model
Simulink Real-Time model
Simulink Real-Time FPGA interface block
Build and download real-time application into target machine
Target machine
Simulink Real-Time model
Simulink Real-Time FPGA interface block
CPU running Simulink Real-Time kernel
PCI bus
TTL I/O
FPGA
Interface block loads bitstream into FPGA

To learn more about the workflow, see “FPGA Programming and Configuration” (Simulink Real-Time) in the Simulink Real-Time documentation.

More About

- “FPGA Programming and Configuration” (Simulink Real-Time)
- “Custom IP Core Generation” on page 26-2

External Websites

- www.speedgoat.com/support/iomodules