HDL Coder Modeling Guidelines (R2015b)

0. Introduction

0.1 About this guide

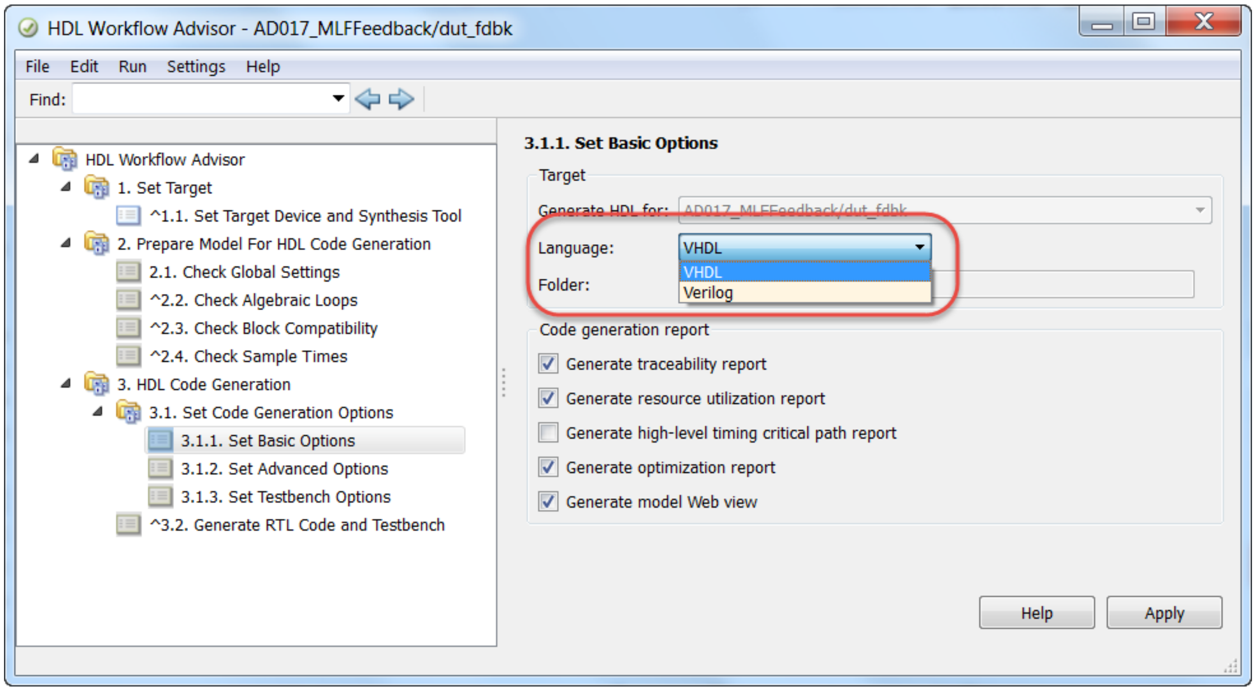
This is a set of recommended guidelines for creating Simulink models, MATLAB function blocks, and Stateflow charts for use with HDL Coder. Because HDL Coder generates code that will target hardware, some amount of hardware architectural guidance must be provided as part of the design. There are additional guidelines for optimizing the speed and area of the design implemented in hardware. Where noted, the guidelines also reflect industry-standard HDL guidelines such as those from STARC.

0.1 Recommended HDL Coder design workflow

MATLAB / Simulink	1.	Reference model design	Design and verify the floating-point functional algorithm.
	2.	Implementation model creation	Add hardware awareness, being mindful of clocking, data types, resource mapping. Use HDL-supported blocks. Typically Simulink is the top-level and primary entry point for this model, though MATLAB function blocks and Stateflow charts may be needed.
Fixed-Point Designer	3.	Fixed-point conversion	Convert floating point data to fixed-point for hardware implementation. Fixed-Point Designer utilizes simulation to provide feedback and guidance on error tolerance and min/max values.
HDL Coder	4.	HDL generation properties and preferences	Set optimization preferences such as pipelining or resource sharing.
	5.	HDL generation readiness check	Checks the model for compliance and consistency with HDL code generation rules.
	6.	HDL generation	Generates VHDL or Verilog.
HDL Verifier + EDA verification	7.	HDL simulation and verification	Validate that the fully-timed bit-accurate HDL still meets functional requirements.
EDA synthesis and place & route	8.	FPGA/ASIC implementation	Implement the generated HDL on the target hardware.

0.2 Target language

HDL Coder generates synthesizable VHDL or Verilog. VHDL is the default. The target language can be set a number of different ways, the most common being Simulink **Configuration Parameters > HDL Code Generation** pane or the Simulink HDL Workflow Advisor as follows:


HDL Workflow Advisor - AD017_MLFFeedback/dut_fdbk
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
    - 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 Testbench Options
    - ^3.2. Generate RTL Code and Testbench

3.1.1. Set Basic Options

Target

Generate HDL for: AD017_MLFFeedback/dut_fdbk

Language: VHDL (VHDL, Verilog)

Folder:

Code generation report

- [✓] Generate traceability report
- [✓] Generate resource utilization report
- [ ] Generate high-level timing critical path report
- [✓] Generate optimization report
- [✓] Generate model Web view

0.3 Definition of terms

Subsystems: Atomic subsystem Variant subsystem Enabled Subsystem Triggered Subsystem Virtual subsystem Non-virtual subsystem

Design concepts: Base rate DUT Registers/Flip-Flops Global reset type Local reset

Signals: Matrix signal Bus signal Vector signal Frame-based signal

Models: Model Variant Model referencing Validation model Generated Model Cosimulation model HDL model parameters HDL block properties HDL-supported blocks Configuration parameter

Implementation: Floating-point to fixed-point conversion Floating-point mapping Sharing Streaming Pipelining DSP slice/block

0.4 Guideline categories

The guidelines are categorized by level of compliance requirements:

	Mandatory	Strongly Recommended	Recommended	Informative
Definition	Not following this rule will result in an error and code cannot be generated.	Code can be generated but it will likely map inefficiently to hardware or may not match the high-level functionality.	Improves quality, readability, or ease of implementation.	Guideline to provide additional information.
Impact	Code generation or logic synthesis cannot be performed.	Poor quality of results	May impact efficiency or ease-of-use downstream.	None

Index of HDL Coder Modeling Guidelines

1.4.4	Use global reset type best suited for your target hardware	Strongly Recommended	FPGA (Altera/Xilinx)	
2.	Block Settings			
2.1	Discontinuities			
2.2	Discrete			
2.2.1	Appropriate use of various types of delay blocks as registers	Recommended	All	1.3.1.3
2.2.2	Map large delays to FPGA block RAM instead of registers to reduce area	Recommended	FPGA (Altera/Xilinx)	
2.3	HDL Operations			
2.3.1	Use a Bit Concat block instead of a Mux block for bit concatenation in VHDL	Mandatory	All	2.1.6.1
2.3.2	Design considerations for RAM Block access	Mandatory	All	
2.3.3	HDL FIFO block usage considerations	Mandatory	All	
2.3.4	Parallel <--> Serial conversion	Informative	All	
2.4	Logic and bit operations			
2.4.1	Logical vs. arithmetic bit shift operations	Informative	All	
2.4.2	Logical Operator, Bitwise Operator, and Bit Reduce for logic operations	Informative	All	
2.4.3	Use Boolean data type for the output of the Compare to Constant/Zero and the Relational Operator blocks	Strongly Recommended	All	
2.5	Lookup tables			
2.5.1	Set the number of Lookup Table data entries to a power of 2 to avoid generation of a division operator (/)	Strongly Recommended	All	
2.5.2	Generating FPGA block RAM from a Lookup Table block	Strongly Recommended	FPGA (Altera/Xilinx)	
2.6	Math operations			
2.6.1	Input vector with Mux block to multi-input adder, multi-input product, and multi-input Min/Max	Strongly Recommended	All	
2.6.2	Set ConstMultiplierOptimization to 'auto' for a Gain block	Strongly Recommended	All	
2.6.3	Use the Bit Shift block or the bitshift function for computations of the power of 2 (ASIC)	Recommended	ASIC	
2.6.4	Use Gain block for computations of the power of 2 (FPGA)	Recommended	FPGA (Altera/Xilinx)	
2.6.5	Use a Gain block for constant multiplication and constant division	Strongly Recommended	All	
2.6.6	Efficient multiplier design for targeting Altera DSP block	Recommended	FPGA (Altera)	
2.6.7	Efficient multiplier design for targeting Xilinx DSP48 slices	Recommended	FPGA (Xilinx)	
2.6.8	Consider speed/area priority and DSP mapping when modeling complex multiplication	Recommended	All	
2.6.9	Model the delay of blocks that will be auto-pipelined (Divide, Sqrt, Trigonometric Function, Cascade Add/Product, Viterbi Decoder)	Recommended	All	
2.6.10	Use Divide blocks in reciprocal mode with a RecipNewton or RecipNewtonSingleRate architecture for more optimal HDL	Strongly Recommended	All	
2.6.11	Consider the additional latency impact of different implementation architectures for the Sqrt and ReciprocalSqrt blocks	Informative	All	

2.6.12	Tradeoffs for Sin/Cos calculation using Trigonometric Function, Lookup Table, Sine/Cosine, and NCO HDL Optimized block	Informative	All	
2.6.13	Use only conj, hermitian, or transpose in a Math Function block	Mandatory	All	
2.6.14	HDL code generation compatible Math Operations for complex number computation	Informative	All	
2.7	Ports and subsystems			
2.7.1	Block settings for Triggered Subsystems/Enabled Subsystems	Mandatory	All	
2.7.2	Proper usage of a Unit Delay Enabled block versus an enabled subsystem with a Delay block	Informative	All	
2.8	Signal attributes			
2.8.1	Rate conversion blocks and usage	Recommended	All	
2.9	Signal routing			
2.9.1	Choosing the right block for extracting a portion of a vector signal	Recommended	All	
2.9.2	Block parameter setting for the Multiport Switch Block	Mandatory	All	
2.9.3	Add 1 to index signals when describing a selector circuit in a MATLAB Function block	Recommended	All	
2.9.4	Use a MATLAB Function block to select indices when extracting portions of a very large constant vector	Recommended	All	
2.9.5	Writing to individual elements of a vector signal using the Assignment block	Mandatory	All	
2.9.6	Proper usage of Goto/From blocks	Mandatory	All	
2.9.7	Ascending bit ordering for 1-D arrays may cause warnings from HDL rule checkers	Informative	All	2.1.6.1
2.10	Source blocks			
2.10.1	Do not use a sample time of inf for a Constant block	Mandatory	All	
2.11	MATLAB Function blocks			
2.11.1	Proper usage of dsp.Delay as a register	Recommended	All	
2.11.2	Update persistent variables at the end of a MATLAB function	Strongly Recommended	All	
2.11.3	Explicitly define data types for constants used in expressions	Mandatory	All	
2.11.4	Use Delay blocks to break feedback loops in MATLAB Function blocks	Mandatory	All	
2.11.5	Do not use logical operators in conditional statements when initializing persistent variables	Recommended	All	
2.11.6	Use X(:)=X+1; when input and output data types are the same in MATLAB code expressions	Recommended	All	
2.11.7	Avoid unintended latch inference by performing arithmetic operations outside of if/else branches	Strongly Recommended	All	2.2.1.1
2.11.8	Avoid generating always @* Verilog code for Xilinx Virtex-4 and 5	Mandatory	FPGA (Xilinx)	
2.11.9	Using MATLAB code for [M, N] matrix operations	Informative	All	
2.11.10	Use a single for loop for element-by-element operations to reduce area	Recommended	All	
2.12	Stateflow			
2.12.1	Choosing Mealy vs Moore for Stateflow state machine type	Strongly Recommended	All	2.11.1.1
2.12.2	Stateflow Chart block configuration	Strongly Recommended	All	

2.12.3	Do not use absolute time for temporal logical logic (after, before and every)	Mandatory	All	
2.12.4	Consider desired state order in generated HDL when naming states	Recommended	All	
2.12.5	Using a chart output as an input via a feedback loop	Recommended	All	
2.12.6	Insert an unconditional transition state to create an else statement in the generated HDL	Strongly Recommended	All	2.7.1.3
2.12.7	Avoid unintended latch inference by performing arithmetic operations outside of truth tables	Strongly Recommended	All	2.2.1.1
2.12.8	Hardware considerations when designing an FSM	Strongly Recommended	All	
2.13	DSP System Toolbox			
2.13.1	Use the DSP System Toolbox Delay block if the number of samples to delay might be 0	Recommended	All	
2.13.2	Changing the phase offset of a Downsample block	Recommended	All	
2.13.3	Use the NCO HDL Optimized block for sine and cosine computation and signal generation	Recommended	All	
2.13.4	Block settings for FIR filter blocks	Informative	All	
2.13.5	IIR Filter blocks	Informative	All	
2.14	Others			
2.14.1	Use case restrictions when importing user-defined HDL code with an HDL Cosimulation block	Mandatory	All	
2.14.2	Define clock and block name to match user-defined HDL settings when using an HDL Cosimulation block	Mandatory	All	
3	Data type settings			
3.1	Basic data type settings			
3.1.1	Use fixed binary point scaling up to 128-bit for fixed-point operations	Mandatory	All	
3.1.2	Trading off rounding error vs processing expense	Strongly Recommended	All	
3.1.3	Restrictions for data type override	Informative	All	
3.2	Simulink data type setting			
3.2.1	Use Boolean for logical data and use ufix1 for numerical data	Mandatory	All	
3.2.2	Define the data type of a Gain block explicitly	Recommended	All	
3.2.3	Restrictions for using enumerated values	Mandatory	All	
3.3	Data type setting for MATLAB code			
3.3.1	Using a fi object in a MATLAB Function block	Strongly Recommended	All	
3.3.2	Use like or cast to inherit data types in MATLAB code	Recommended	All	
3.3.3	Use True/False instead of Boolean data in MATLAB code	Mandatory	All	
3.4	Data type setting for Stateflow charts			
3.4.1	Use a fi object when the Stateflow action language is MATLAB	Mandatory	All	
4	Optimization of speed and area			
4.1	Resource sharing			
4.1.1	Resource sharing requirements	Mandatory	All	
4.1.2	Use StreamingFactor for resource sharing of 1D vector signal processing	Informative	All	
4.1.3	Resource sharing of Gain blocks	Recommended	All	
4.1.4	Resource sharing of Product blocks	Recommended	All	
4.1.5	Resource sharing of subsystems	Recommended	All	

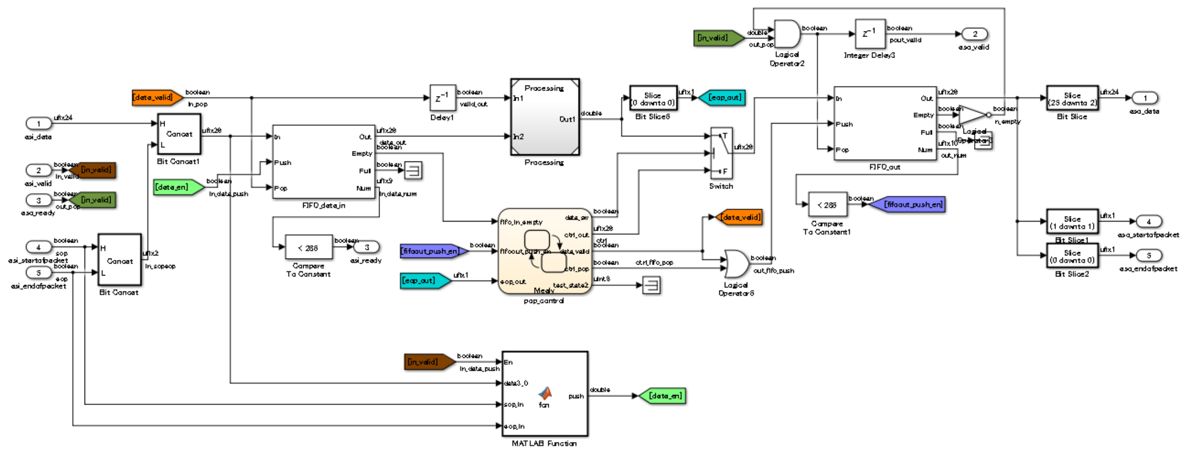
4.2	Pipeline insertion			
4.2.1	Design considerations for pipelining and delay balancing	Recommended	All	
4.2.2	Clock-rate pipelining	Informative	All	
4.2.3	Recommended distributed pipelining settings	Recommended	All	
4.2.4	Apply distributed pipelining to adders, products, min/max, and dot products with vector inputs	Informative	All	
5	Appendix			
5.1	Considerations in HDL code writing for ASIC/FPGA design			
5.2	Synchronous circuit design overview and recommendations			
5.3	Recommended use of registers at outputs of hierarchical structures			
5.4	Follow naming conventions			
5.5	HDL-supported blocks			
5.6	Compatibility check for HDL code generation			
5.7	Setting global clock and reset signals in for HDL code generation			
5.8	Add comments for generating readable HDL code			

1. Architecture Design

1.1 Basic settings

1.1.1 Appropriate use of Simulink, Stateflow, MATLAB Function, BlackBox, Model Reference and HDL Cosimulation block

When creating a hardware implementation model, there are recommended applications for Simulink blocks, MATLAB function blocks and Stateflow charts. These can be mixed within a single subsystem to create a complete model as shown in the following figure:



The recommended application for each type of block is as follows:

- Simulink block: Arithmetic algorithm containing numerical processing or feedback loop.
- MATLAB Function block: Control logic, conditional branch (If/Else statement), simple state machine, and IP written with MATLAB code.
- Stateflow:
  - State chart (Chart, State Transition Table block): Mode logic or state machine which control an output by logic of the past and the present
  - Flow chart (Chart block): Multiple conditional branch (If/Else)
  - Truth table (Truth Table block): Multiple conditional branch (If/Else)

  The algorithm modeled by Stateflow uses logic as main elements, and when complicated operation is included, describe that the calculated result of Simulink block is changed in the logic of Stateflow. Because explicit pipeline processing cannot be described in Stateflow and change of the timing by pipelining insertion is unclear.

- BlackBox: For subsystems that don't need simulation, or that will use imported HDL code. This is an architecture property that can be applied to a subsystem or a referenced model (for example, an interface circuit for an A/D converter, an SDRAM controller, etc.) It is also possible to use the BlackBox property to incorporate handwritten code into a cosimulation model.
- Model reference: For re-using models as sub-blocks in other models. This is useful for partitioning a design to be worked on by multiple engineers in parallel. For more on HDL code generation from a referenced model, see the documentation.

  Note that since a referenced model is treated the same as an Atomic subsystem, an algebraic loop may occur which will prevent HDL code generation. These can be fixed in the design, or possibly by setting the **Minimize algebraic loop occurrences** in the **Model Referencing** pane of Configuration Parameters.

- HDL Cosimulation: For simulating HDL code for the DUT in Mentor® Questa® or ModelSim®, or Cadence® Incisive®, connected to the Simulink environment via HDL Verifier.

1.1.2 Use the `hdlsetup` command to set model configuration parameters and HDL model properties

`hdlsetup('modelname')` sets the parameters of the model specified by `modelname` to common default values for HDL code generation.

Example: myhdlsetup.m

1.1.3 Avoid using double-byte characters

Double-byte characters, which are used for Japanese and Chinese characters, are typically not supported by downstream logic synthesis and simulation tools. Therefore HDL code generation does not support them in model and block names.

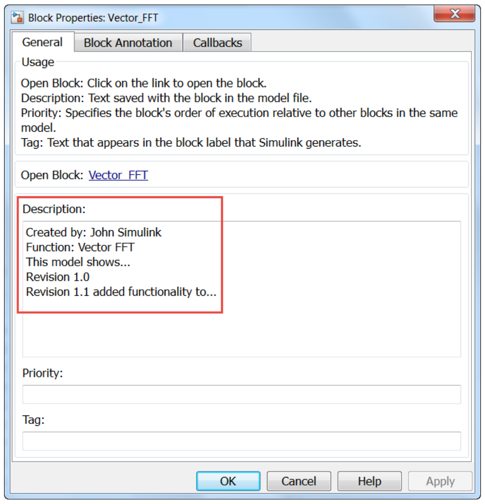
It is also recommended to avoid using double-byte characters in comments as well, since comments are propagated into the generated code. It is good practice to use English for comments.

1.1.4 Consider resource sharing impact during model creation

See 4.1 Resource Sharing.

1.1.5 Document block name, block features, authors, etc., in subsystem block properties

To improve management of the generated HDL, it is good practice to document reference information in the subsystem block properties since these will be generated as comment headers in the HDL. For example:


Block Properties: Vector_FFT

General | Block Annotation | Callbacks

Usage

Open Block: Click on the link to open the block.
Description: Text saved with the block in the model file.
Priority: Specifies the block's order of execution relative to other blocks in the same model.
Tag: Text that appears in the block label that Simulink generates.

Open Block: Vector_FFT

Description:

Created by: John Simulink
Function: Vector FFT
This model shows...
Revision 1.0
Revision 1.1 added functionality to...

Priority:

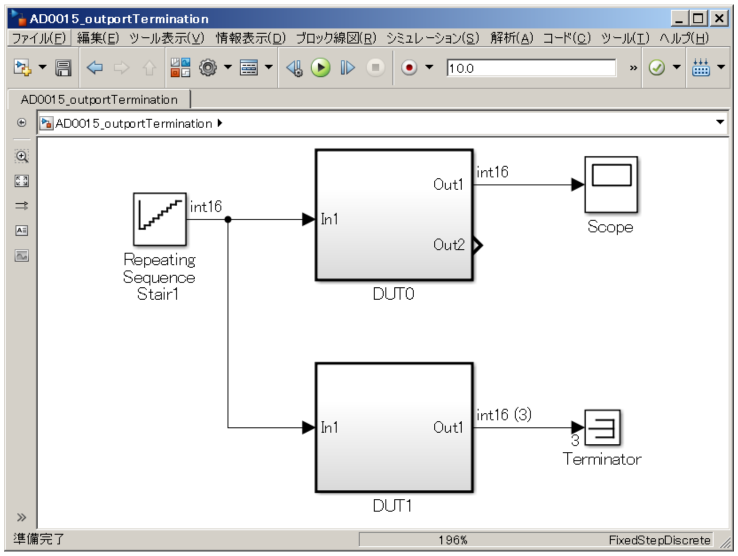
Tag:

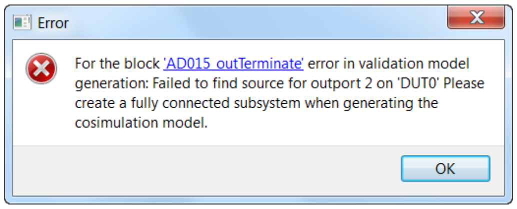
This generates code with a header that looks like this:

```
-- Simulink subsystem description for vector_fft_implementation_example/Vector_FFT:
--
-- Created by: John Simulink
-- Function: Vector FFT
-- This model shows...
-- Revision 1.0
-- Revision 1.1 added functionality to...
--
-- -------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY Vector_FFT IS
```

1.1.6 Terminate unconnected block outputs with Terminator blocks

HDL code generation will fail and generate an error when output ports of blocks are unconnected. For output blocks that are intentionally not connected to downstream logic, connect them to a Terminator block. The following illustrates:


AD0015_outportTermination
ファイル(F) 編集(E) ツール表示(V) 情報表示(D) ブロック線図(R) シミュレーション(S) 解析(A) コード(C) ツール(T) ヘルプ(H)
10.0
int16
Repeating Sequence Stair1
In1
Out1
Out2
DUT0
int16
Scope
In1
Out1
DUT1
int16 (3)
3
Terminator
準備完了
196%
FixedStepDiscrete


Error

For the block 'AD015_outTerminate' error in validation model generation: Failed to find source for outport 2 on 'DUT0' Please create a fully connected subsystem when generating the cosimulation model.

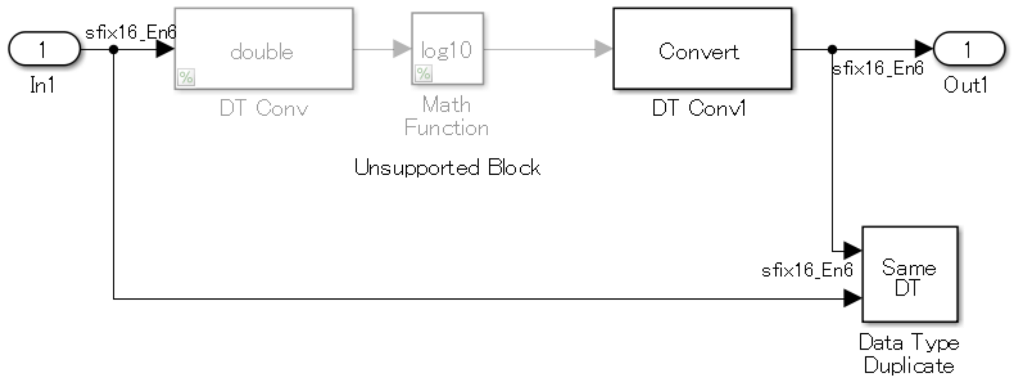
OK

Example: AD015_outTerminate.slx

1.1.7 Proper usage of commenting out blocks

Code generation will fail if a block has been tagged as a “comment through” pass-through.

Code can be generated for a block that is commented out. The generated code will assign a constant value of 0 to the signal that would have been connected to its output. For instance the following example:



Generates the following HDL:

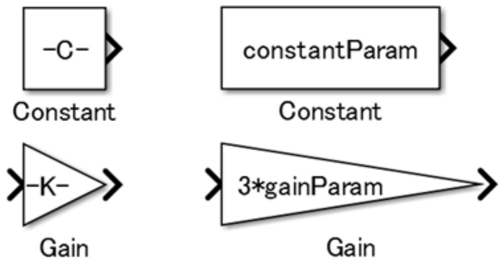
VHDL generated from the subsystem containing a commented-out block	Verilog generated from the subsystem containing a commented-out block

```
ENTITY Generated IS
   PORT( In1  :  IN    std_logic_vector(15 DOWNTO 0);
         Out1 :  OUT   std_logic_vector(15 DOWNTO 0)
         );
 END Generated;

ARCHITECTURE rtl OF Generated IS

  SIGNAL TmpGroundAtDT_DupIn1_out1 : signed(15 DOWNTO
```
	```
	module Generated
	           (
	            In1,
	            Out1
	           );

	  input   signed [15:0] In1;
	  output  signed [15:0] Out1;
	```

`0);`
 	
`  BEGIN`
`    TmpGroundAtDT_DupIn1_out1 <= to_signed(16#0000#, 16);`
 	
`     Out1 <= std_logic_vector(TmpGroundAtDT_DupIn1_out1);`
 	
` END rtl;`
	`   wire signed [15:0] TmpGroundAtDT_DupIn1_out1;`
	
	`   assign TmpGroundAtDT_DupIn1_out1 = 16'sb0000000000000000;`
	
	`   assign Out1 = TmpGroundAtDT_DupIn1_out1;`
	
	`endmodule`

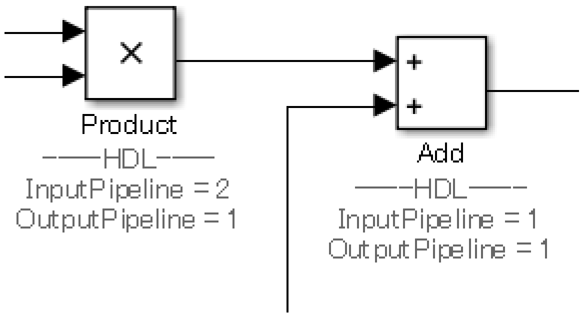
1.1.8 Adjust sizes of constant and gain blocks so that parameters can be identified

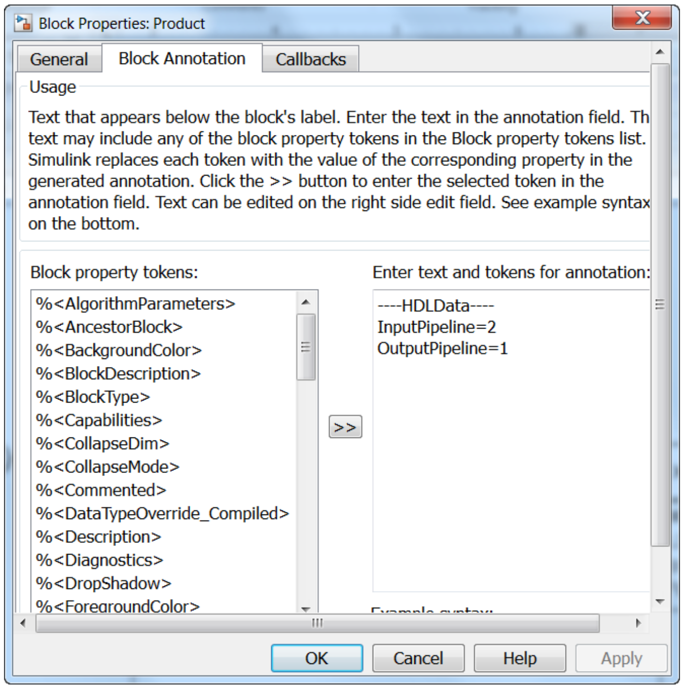
For constant blocks and gain blocks that use parameter values, in order to increase readability it is good practice to adjust the size of the block so that the parameter value can be displayed. For instance:



1.1.9 Display parameters that will affect HDL code generation

Certain block parameters such as pipelining and resource sharing can significantly affect HDL code generation. Therefore if these parameters are set, it is good practice to display them in the Simulink diagram. It also helps to use delimiters such as "--------" to separate the annotation from the block name. For example:




Block Properties: Product

General | Block Annotation | Callbacks

Usage

Text that appears below the block's label. Enter the text in the annotation field. Th
text may include any of the block property tokens in the Block property tokens list.
Simulink replaces each token with the value of the corresponding property in the
generated annotation. Click the >> button to enter the selected token in the
annotation field. Text can be edited on the right side edit field. See example syntax
on the bottom.

Block property tokens:

```
%<AlgorithmParameters>
%<AncestorBlock>
%<BackgroundColor>
%<BlockDescription>
%<BlockType>
%<Capabilities>
%<CollapseDim>
%<CollapseMode>
%<Commented>
%<DataTypeOverride_Compiled>
%<Description>
%<Diagnostics>
%<DropShadow>
%<ForegroundColor>
```

Enter text and tokens for annotation:

```
----HDLData----
InputPipeline=2
OutputPipeline=1
```

Example: BS013_blockAnnotation.slx, showHdlBlockParams.m

The sample file showHdlBlockParams.m attaches a delimiter and annotation automatically to the block to which the following HDL block properties are set:

- BalanceDelays
- DistributedPipelining
- ConstrainedOutputPipeline
- InputPipeline, OutputPipeline
- StreamingFactor
- SharingFactor

In order to attach an annotation of the above property via the command-line:

```
>> showHdlBlockParams (<blockname> and 'on -- ')
```

To delete the annotation of the above property:

```
>> showHdlBlockParams (<blockname>, 'off')
```

1.1.10 Change block parameters by using find_system and set_param

The functions `find_system` and `set_param` can be used together to batch modify the parameters of specific blocks. The following is an example script that detects Constant blocks with a **Sample time** of inf and batch modifies it to -1:

```
        modelname = ‘sfir_fixes’

        % Detect all the Constant blocks in a model.
        blockConstant = find_system(bdroot, 'blocktype', 'Constant')
```

```
% Sampling time detects the Constant block used as [inf],
% and changes sampling time into [-1].

for n = 1:numel(blockConstant)
    sTime = get_param(blockConstant{n}, 'SampleTime')
    if strcmp(lower(sTime), 'inf')
        set_param(blockConstant{n}, 'SampleTime', '-1')
    end
end
```

1.2 Subsystem and Model Hierarchy

1.2.1 When the DUT is not at the top level of the model, set the DUT as a non-virtual subsystem

When the DUT (the target subsystem for code generation) exists in a lower hierarchy from the top, HDL Coder converts it to a model reference. Because the execution sequence of a referenced model is equivalent to an Atomic subsystem, when the DUT is a virtual subsystem this conversion may change its operation.

Therefore set the DUT as a non-virtual subsystem before verification and code generation. Subsystem types to which it can be set are: Atomic Subsystem; model reference; Variant Subsystem; and a variant model.

A conditionally-executed subsystem (Enabled Subsystem, Triggered Subsystem) cannot be specified as a DUT. In order to use one as a top-level, create an Atomic Subsystem one level of hierarchy up from it.

If there is a feedback loop out of an atomic subsystem that results in an algebraic loop, it will result in a code generation error. This can be fixed in the design by setting the **Minimize algebraic loop occurrences** in the **Model Referencing** pane of Configuration Parameters.

1.2.2 Type of subsystem and hierarchical design for a DUT

Considerations for a DUT subsystem:

- Because a difference in simulation results may occur, it is good practice to make the DUT a non-virtual subsystem, e.g. an Atomic subsystem, when it is not at the top level of the model.
- When generating code from a subsystem which is low in the design hierarchy, make the HDL block property of the subsystem into the default configuration.
- If the DUT is lower in the design hierarchy, because it gets converted to a reference model, a new Simulink model with references to the validation and co-simulation models will be generated in the target directory for code generation.
- The subsystem which is at the top of the hierarchy for code generation cannot be set as a BlackBox.
- Connect outputs with no fanout to a Terminator block.
- Don't place a comment through and a comment out block into the DUT.

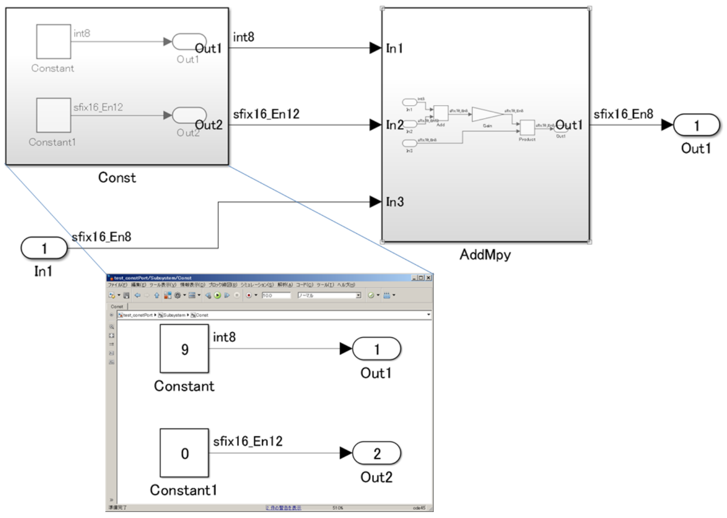
Guidelines for usage of various subsystems:

- Virtual subsystem
  - A subsystem is virtual if the block is neither conditionally executed nor atomic.
  - Don’t use it as a DUT. Use this type of subsystem in lower levels of the hierarchy where you want to divide a generated file.
- Atomic Subsystem
  - Make the DUT an Atomic Subsystem.
  - Use Atomic Subsystems to generate a single HDL file for identical instances of subsystems in lower levels of hierarchy.

        - To enable resource sharing in a subsystem unit, make all target subsystems into Atomic Subsystems.
- Variant Subsystem (*Example : AD004_variantSubDivide.slx and AD020_variantChildOp.slx*)
    - Use this block to change the behavior of a subsystem by using a MATLAB variable without having to modify the subsystem itself.
    - The file name and instance name for the generated code will be unique to the active configuration at the time of code generation.
    - It cannot be set as the top hierarchy for a DUT.
    - It cannot be a target of resource sharing.
- Model Referencing (*Example: AD002_modelRefDivide.slx and AD019_modelRefChild.slx*)
    - Use this block to unify a model composed of smaller partitions. It also enables incremental code generation.
    - A reference model can be set as the DUT for code generation. A directory will be created using the reference model name and the HDL code will be placed under it.
    - When using a continuous block in a testbench, make the DUT into a reference model.
    - The preference of the block parameter [Model argument values] (for this instance) of a reference model is not equivalent to a Code Generation.
- Model Variants (*Example: AD003_modelVariantDivide.slx and AD021_varianChildRecip.slx*)
    - This block is the same use-case as a reference model, except its behavior can be changed by using a MATLAB variable.
    - Because it becomes the model name set as the file name/instance name generated being effective, if an effective subsystem is changed, it will be cautious of the file name/instance name generated changing.
    - Similar to a Variant Subsystem, it cannot be set as a code generation target subsystem.

1.2.3 Do not connect constant blocks to ports directly crossing subsystem boundaries

For an example such as the following, where a constant is directly connected to the output port of a subsystem:



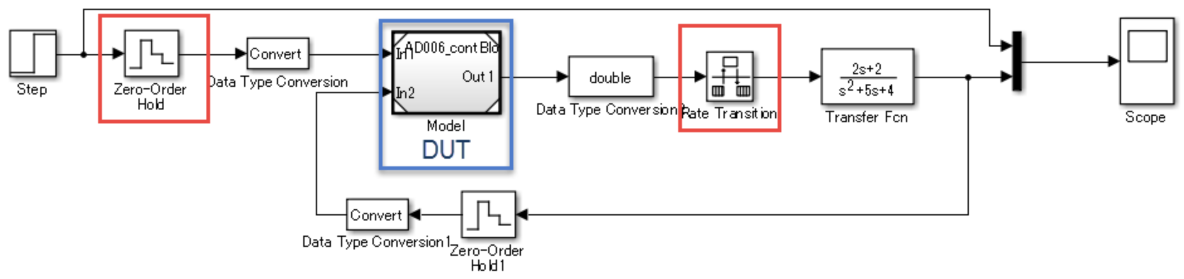
Logic synthesis may optimize away the constant, resulted in an unconnected output port.

1.2.4 For testbenches that use blocks in continuous solver mode, make the DUT a model reference with a discrete solver.

Some testbenches may include parts that require continuous solvers, such as the Continuous library and Simscape. Since the lower levels of hierarchy inherit the solver settings by default, convert the DUT subsystem to a referenced model and that points to a model that uses a discrete solver.

You will need to insert sample time conversion blocks such as Rate Transition or Zero-Order Hold at the boundaries of the DUT to convert the input and output signals.

Examples: AD005_continuous.slx, AD006_contBlock.slx



1.2.5 Generate re-usable HDL code from identical subsystems

When there are two or more subsystems (including library blocks) which perform the same function, by default an HDL file for each of these virtual subsystems will be generated. In order to generate a single HDL file, convert this subsystem to an Atomic Subsystem. This will make the generated files easier to manage. See Generate Reusable Code from Atomic Subsystems in the product documentation for an example.

1.2.6 Generate parameterized HDL code for gain and constant blocks

When using several masked subsystems for which only the Constant or Gain parameters differ, you can reduce the number of generated HDL files by setting **Generate parameterized HDL code from masked subsystem** to “on” in the **HDL Code Generation** pane.

Example: AD012_HDLParameter.slx

1.2.7 Insert handwritten code for a block into the generated code for the DUT

Some cases, such as re-using pre-verified RTL IP, require insertion of existing code for a block into the DUT. In order to ease this process, create the block in Simulink in order to be plug-in-compatible with the generated code. This includes the following:

- Name the block the same name as the VHDL entity or Verilog module
- Define the same inputs and outputs, including the same types, sizes, and names
- Define the same clock, reset, and clock enable. Note that only one clock, reset, and clock enable per block is allowed.
- The block can only be single-rate

Then set the block to be a Black Box in order to disable code generation. See the documentation for details.

1.2.8 Only use numerical values and string data types for mask parameters for user-defined subsystems

By using parameterized values for masked subsystems, you can generate parameterized HDL code. But code can only be generated if the mask parameters are numerical values or strings. Using objects, types, or Simulink API commands such as `add_block`, `add_line`, etc. will result in a code generation error.

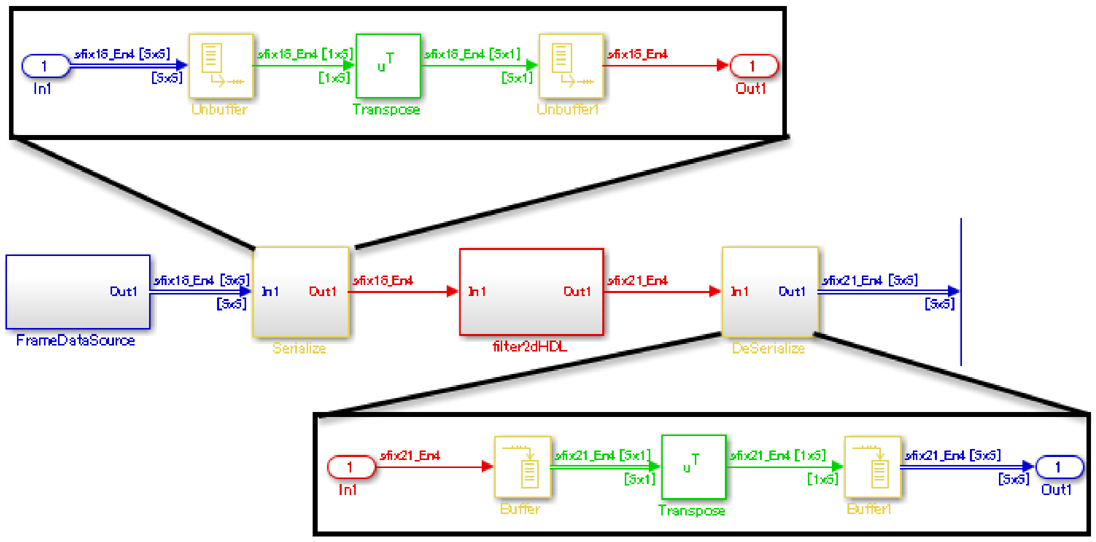
1.3 Signal types

1.3.1 Serialize 2D matrix signals into a 1D signal before it enters an HDL subsystem, and vice versa for the output

Because hardware interfaces are one-dimensional, any multi-dimensional matrix must be serialized into a hardware-friendly one-dimensional scalar or vector. For image/video signals, use the Frame-To-Pixels block in the Vision HDL Toolbox product.

Example: AD001_matrix.slx

This example model performs serialization of two-dimensional data at the input, and then deserialization at the output.



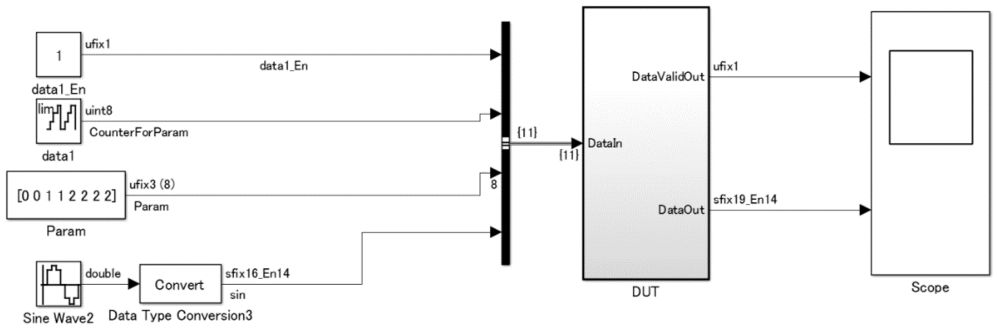
- Serialize: Converts the two-dimensional matrix data of size MxN into scalar data. It consists of two Unbuffer blocks and a Transpose block. The first Unbuffer block divides data into a line writing direction, and divides the 5x5 matrix data into four 1x5 vectors. Note that the sampling time is set to one-fourth since there will be four output vectors for each input sample. The Transpose block that follows transposes the 1x5 vector into a 5x1 vector. The last Unbuffer block divides the 5x1 vector data into scalar data for input to the image processing subsystem (filter2HDL).
- DeSerialize: Converts the scalar data output from the filter2HDL subsystem into a two-dimensional matrix. It consists of two Buffers and one Transpose block. This converts the scalar data to lines then to the MxN matrix, in an inverse operation from the Serialize block. Note that the output sample time ends up being the same as the sample time on the input to the Serialize block.

In addition, two-dimensional processing is supported within the MATLAB code and the MATLAB Function block. For details, refer to 2.11.10 Use MATLAB code for [M and N] matrix operation.

1.3.2 Using a signal bus to improve readability

When a DUT has many input or output signals, to improve readability, create a bus signal using a Bus Creator block to create a single structure for the input or output

Example: AD022_bus.slx



The bus signal can be a structure of different data types or a vector signal of the same data types. However, since the amount of blocks that support bus signals is limited, when performing signal conditioning, it is necessary to extract the desired signal from a bus using a Bus Selector block. The HDL code corresponding to the signal line in a bus signal serves as a signal name in an input or output port as the generated code shown below.

VHDL generated from the model containing a bus signal

```
 ENTITY DUT IS
   PORT( clk                       :   IN    std_logic;
       reset                       :   IN    std_logic;
       clk_enable                  :   IN    std_logic;
       DataIn_data1_En             :   IN    std_logic;
       DataIn_CounterForParam      :   IN
std_logic_vector(7 DOWNTO 0);  -- ufix8
       DataIn_Param                :   IN
vector_of_std_logic_vector3(0 TO 7);  -- ufix3 [8]
       DataIn_sin                  :   IN
std_logic_vector(15 DOWNTO 0);  -- sfix16_En14
       ce_out                      :   OUT   std_logic;
       DataValidOut                :   OUT   std_logic;
-- ufix1
       DataOut                     :   OUT
std_logic_vector(18 DOWNTO 0)  -- sfix19_En14
       );
 END DUT;
 .
 .
 .
   SIGNAL Bus_Creator2_out1_Param          :
vector_of_std_logic_vector3(0 TO 7);  -- ufix3 [8]
.
 .
 .
    -- <S1>/Enabled Subsystem
    u_Enabled_Subsystem : Enabled_Subsystem
       PORT MAP( clk => clk,
               reset => reset,
               enb => clk_enable,
               In1_CounterForParam =>
std_logic_vector(CounterForParam),  -- uint8
               In1_Param => Bus_Creator2_out1_Param,
-- ufix3 [8]
               In1_sin => std_logic_vector(sin),  --
sfix16_En14
               Enable => data1_En,  -- ufix1
               Out1 => Enabled_Subsystem_out1  --
sfix19_En14
                 );

   -- <S1>/Bus Selector5
   DataValidOut <= Delay_out1_data1_En;

   DataOut <= Enabled_Subsystem_out1;

 .
 .
 .
```

Verilog generated from the model containing a bus signal

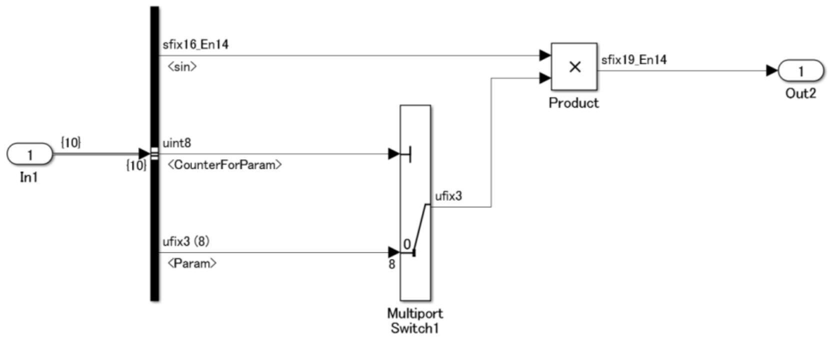
```
 module DUT
       (
        clk,
        reset,
        clk_enable,
        DataIn_data1_En,
        DataIn_CounterForParam,
        DataIn_Param_0,
        DataIn_Param_1,
        DataIn_Param_2,
        DataIn_Param_3,
        DataIn_Param_4,
        DataIn_Param_5,
        DataIn_Param_6,
        DataIn_Param_7,
        DataIn_sin,
        ce_out,
        DataValidOut,
        DataOut
       );
 .
 .
 .
wire [2:0] Param [0:7];  // ufix3 [8]
 .
 .
 .
   // <S1>/Enabled Subsystem
   Enabled_Subsystem   u_Enabled_Subsystem
(.clk(clk),
                        .reset(reset),
                        .enb(clk_enable),
                        .In1_CounterForParam(CounterF
orParam),  // ufix8
                      .In1_Param_0(Param[0]),
                      .In1_Param_1(Param[1]),
                      .In1_Param_2(Param[2]),
                      .In1_Param_3(Param[3]),
                      .In1_Param_4(Param[4]),
                      .In1_Param_5(Param[5]),
                      .In1_Param_6(Param[6]),
                      .In1_Param_7(Param[7]),
                      .In1_sin(sin),  // sfix16_En14
                      .Enable(data1_En),  // ufix1
                      .Out1(Enabled_Subsystem_out1)
// sfix19_En14
                        );

   assign DataOut = Enabled_Subsystem_out1;
   assign ce_out = clk_enable;
 .
```

	. .

The blocks that support the **Input a bus signal** property for HDL generation are:

- Bus Creator
- Bus Selector
- MATLAB Function
- Delay, Memory
- Zero-order Hold
- Rate Transition (both up and down)
- Signal Specification
- From/Goto
- Switch
- Multi-Port Switch
- Stateflow
- Model Reference
- Vision HDL toolbox blocks (accept a pixel control bus for their control input)

To extract and use a signal from a Simulink bus signal to a block other than those listed above, use a Bus Selector block:

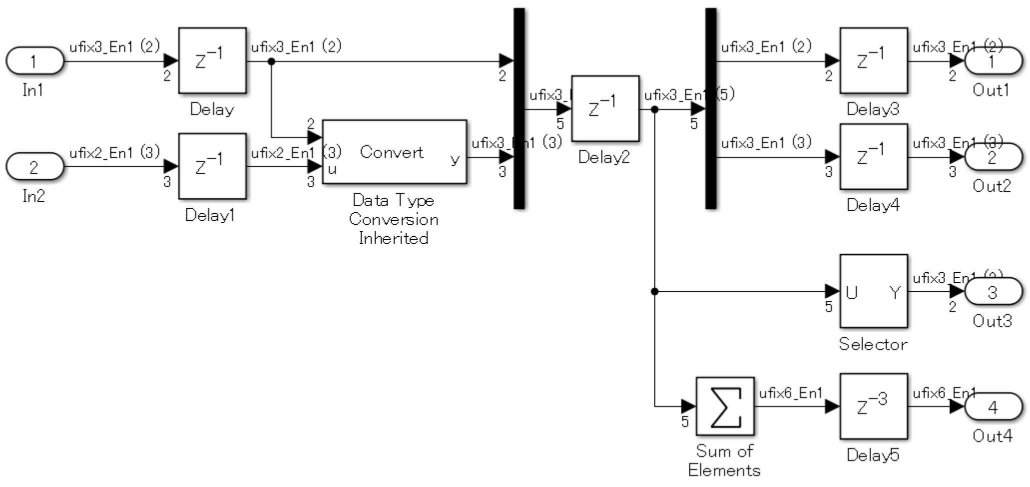


For more on HDL code generation support of bus structures, see [Signal and Data Type Support]() in the documentation.

1.3.3 Design considerations for vector signals

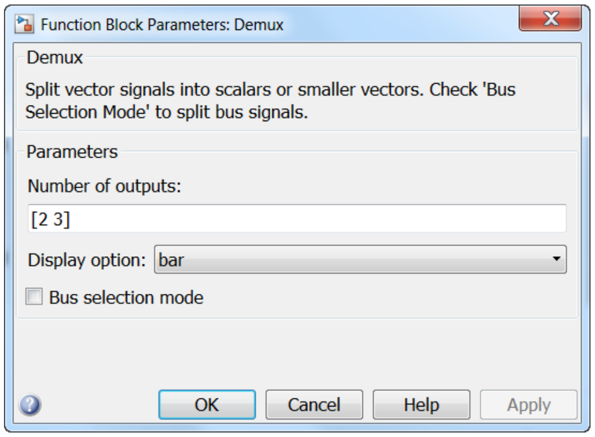
In order to process a group of signals with the same attributes as a vector signal, use a [Mux block]().

Example: AD014_vector.slx



The signals to combine into a vector can be different dimensions. However they must be of the same data type. If they are not the same data type, use a Data Type Conversion block to avoid an error.

When separating and choosing the signal of a desired number of element from a vector signal, use a Demux block or a Selector block. In the following figure, two branches separate the vector signal of the number of element 5 into the vector signal of the number of elements 2 and 3 in the Demux block.



Function Block Parameters: Demux

Demux

Split vector signals into scalars or smaller vectors. Check 'Bus Selection Mode' to split bus signals.

Parameters

Number of outputs:

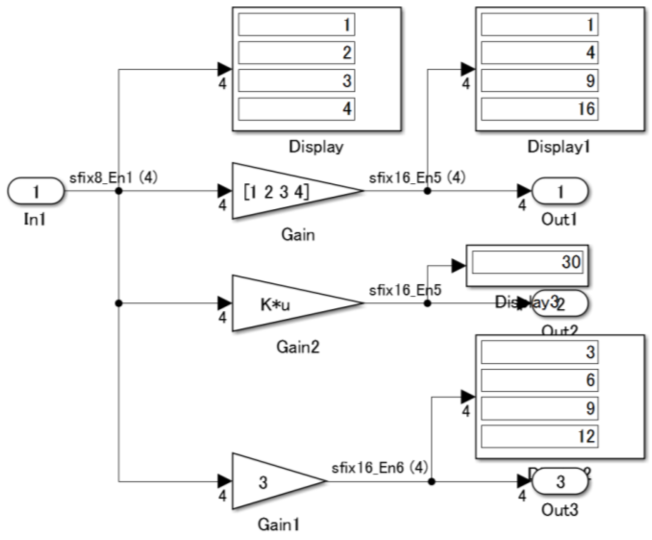
[2 3]

Display option: bar

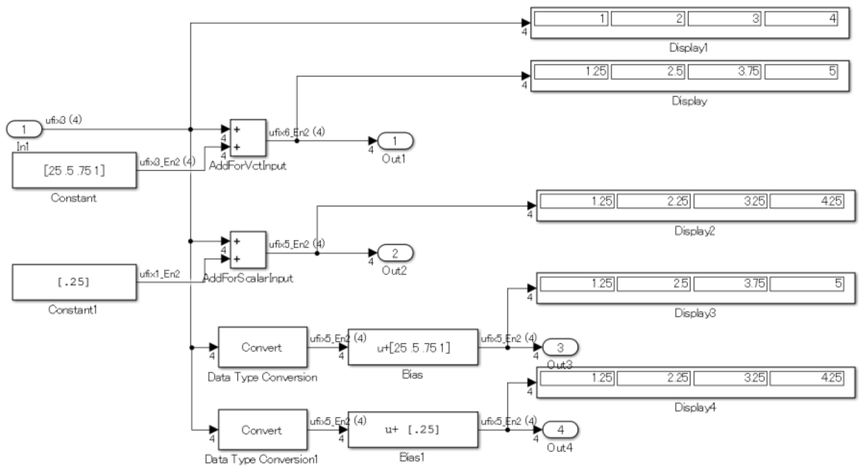
☐ Bus selection mode

Methods for performing operations using vector signals are as follows:

- Various methods can be used for constant multiplication using a Gain block :
  - 1 block of Gain(s): Multiplication for every element of a vector [A*a, B*b, C*c, and D*d]
  - 2 blocks of Gain(s): Matrix multiplication A*a+B*b+C*c+D*d of a vector
  - 3 blocks of Gain(s): Multiplication of a vector and a scalar [A*a, B*a, C*a, and D*a]



- Various methods can be used for performing constant addition/subtraction using either a constant and an Add block, or a Bias block.
  - Addition of a vector signal and a scalar [A+a, B+a, C+a, and D+a]
  - Addition of a vector signal and a vector [A+a, B+b, C+c, and D+d]



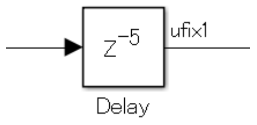
- When addition and subtraction of the elements of a vector signal are performed, Refer to [2.6.1 Input vector with Mux block to multi-input adder, multi-input product, and multi-input Min/Max](#).

1.3.4 One-dimensional vectors created by Delay, Mux, and Constant blocks generate HDL with ascending bit order

In order to assure compatibility with MATLAB vector signals, Simulink by default will create one-dimensional vectors specified in ascending order from LSB to MSB when using Delay and Mux blocks. This goes against convention in VHDL and Verilog and will trigger a warning from HDL rule checkers. You can either ignore the rule checker warning and ensure that all of your vector connections match in their ordering, or make the following changes to make sure that all vectors conform to the [MSB:LSB] convention.

Example: BS054_downto.slx

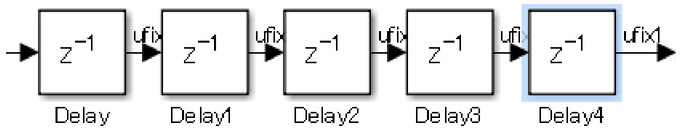
1. Delay block with delay value greater than 1 :



Generated VHDL:

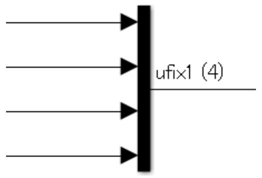
```
-- <S1>/Delay
Delay_process : PROCESS (clk, reset)
     BEGIN
       IF reset = '1' THEN
         Delay_reg <= (OTHERS => '0');
       ELSIF clk'EVENT AND clk = '1' THEN
         IF enb = '1' THEN
           Delay_reg(0) <= In1;
           Delay_reg(1 TO 4) <= Delay_reg(0 TO 3);
         END IF;
       END IF;
     END PROCESS Delay_process;
```

This can be addressed by using single-delay blocks in series:



The resulting VHDL will use scalar signals instead of vector signals.

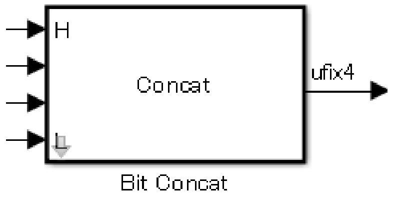
2. Mux block



Generated VHDL:

```
SIGNAL Mux_out1          : std_logic_vector(0 TO 3);  -- ufix1 [4]
-- <S1>/Mux
Mux_out1(0) <= In1;
Mux_out1(1) <= In1;
Mux_out1(2) <= In1;
Mux_out1(3) <= In1;
```

This can be addressed by using the Bit Concat block from the **HDL Coder > HDL Operations** library:



Generated VHDL:

```
 Out2                    :   OUT   std_logic_vector(3 DOWNTO 0);  -- ufix4
```

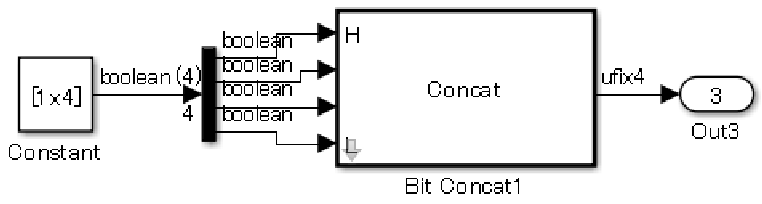
```
        -- <S1>/Bit_Concat
        Bit_Concat_out1 <= unsigned'(In1 & In1 & In1 & In1);
        Out2 <= std_logic_vector(Bit_Concat_out1);
```

3. Constant block

   Generated VHDL:

```
        Out3                    :   OUT   std_logic_vector(0 TO 3)  -- boolean [4]
              -- <S1>/Constant

        Constant_out1(0) <= '0';
        Constant_out1(1) <= '1';
        Constant_out1(2) <= '0';
        Constant_out1(3) <= '1';
```

This can be addressed by using a Demux together with the Bit Concat block from the **HDL Coder > HDL Operations** library:



Generated VHDL:

```
        Out3                              :   OUT   std_logic_vector(3 DOWNTO 0)  -- ufix4

       -- <S1>/Constant
       Constant_out1(0) <= '0';
       Constant_out1(1) <= '1';
       Constant_out1(2) <= '0';
       Constant_out1(3) <= '1';

       -- <S1>/Demux
       Constant_out1_0 <= Constant_out1(0);
       Constant_out1_1 <= Constant_out1(1);
       Constant_out1_2 <= Constant_out1(2);
       Constant_out1_3 <= Constant_out1(3);

       -- <S1>/Bit_Concat1
       Bit_Concat1_out1 <= unsigned'(Constant_out1_0 & Constant_out1_1 & Constant_out1_2 &
Constant_out1_3);
       Out3 <= std_logic_vector(Bit_Concat1_out1);
```

1.3.5 Manually write HDL control logic for 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.

However since Simulink cannot simulate the behavior of bidirectional ports, you will need to manually write HDL control logic in the black box in order to properly verify system-level behavior.

1.4 Clock and Reset

In Simulink, global signals such as clock, clock enable and reset are not explicitly modeled. Instead, they are created during code generation. You represent clock cycles in a Simulink model using sample time.

For a single-rate model, 1 sample time in Simulink maps to 1 clock cycle in HDL. You can use a relative mapping (e.g. 1 second in Simulink = 1 HDL clock) or an absolute mapping (e.g. 10e-9 second in Simulink = one 10 ns clock in HDL), depending on your preference and design requirement.

1.4.1 Creating a frequency-divided clock from the Simulink model's base sample rate

You can assign a frequency-divided clock rate for HDL code generation to be a multiple of the Simulink base sample rate. For instance in a case where the Simulink base rate is 1 MHz and your target hardware will run at 50 MHz, you can assign a global oversampling factor of 50 in the **HDL Code Generation > Global Settings** pane in Configuration Parameters.

1.4.2 Use master-clock division or a clock multiple for proper multi-rate model operation

For a multi-rate model, the fastest sample time maps to 1 clock cycle in HDL. Blocks operating at slower sample times use the same clock in HDL, but are gated with clock enable signals that are active once every N clock cycles. You can also specify HDL Coder to generate multiple synchronous clock signals. Each of the clock signals corresponds to one rate in Simulink.

Note: Some optimization settings (e.g. sharing factor) and alternative block architecture (e.g. Newton-Raphson square root) introduces additional sample rates not present in the original model. In those cases, the fastest generated sample time is mapped to 1 HDL clock

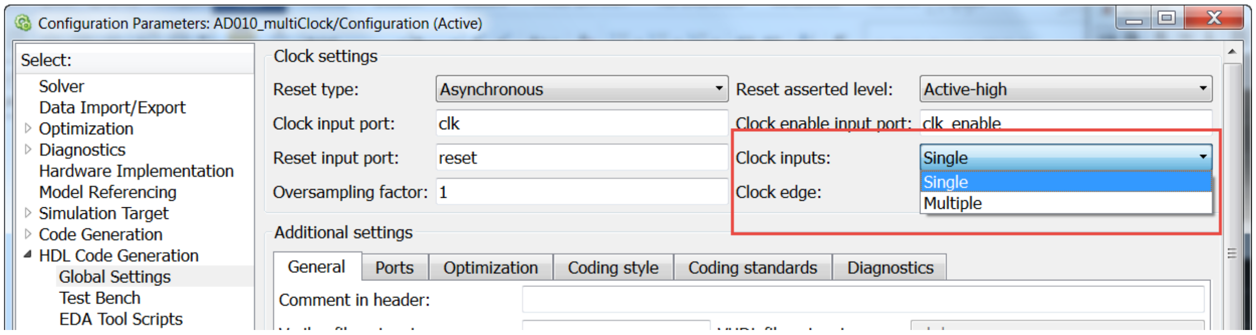
In order to model multi-rate clocks in Simulink, use the following blocks:

- Simulink > Signal Attributes > Rate Transition
- DSP System Toolbox > Signal Operations > Upsample, Downsample, Repeat
- HDL Coder > HDL Operations > HDL FIFO

For a Rate Transition block, select the block parameters **Ensure data integrity during data transfer** and **Ensure deterministic data transfer (maximum delay).** The output sample rate needs to be an integer multiple of the input – for input sample time Ts1 = 4 and output sampling time Ts2 = 12 can be used, however Ts1 = 3 and Ts2 = 4 will produce an error.

There are two ways to generate a clock signal for a DUT that has multiple sample rates:

Example: AD010_multiClock.slx


Configuration Parameters: AD010_multiClock/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:

Additional settings

Comment in header:

Single: use a single clock, and clock enables for lower rates. This preference is simpler because only a single clock signal is needed for all registers, however this can result in more power dissipation since the fastest clock is connected to every register in the design. To reduce power, map the clock enable to a gated clock in your HDL design.

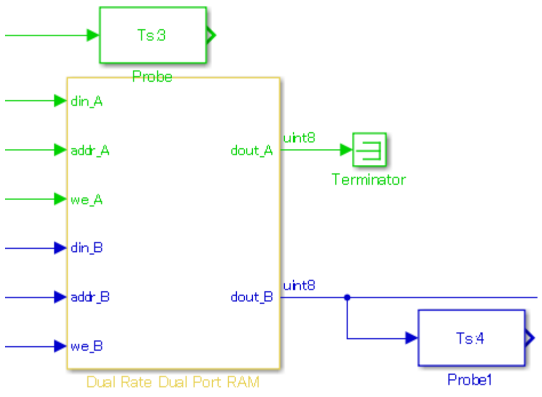
Multiple: generate clock ports for every sample rate in the DUT. This requires more work because you will need to connect each of the clock, clock enable, and reset ports externally. But power can be reduced in registers connected to the slower clock signals.

1.4.3 Use Dual Rate Dual Port RAM for non-integer multiple sample times in a multi-rate model

Using Rate Transition and Upsample/Downsample blocks to create multi-rate models requires that the clock rates be integer multiples of the base rate. To create a multi-rate model with clocks that are non-integer multiples, a Dual Rate

Dual Port RAM block can be used. Take care to manage address control as described in 2.3.2 Design considerations for RAM block access.

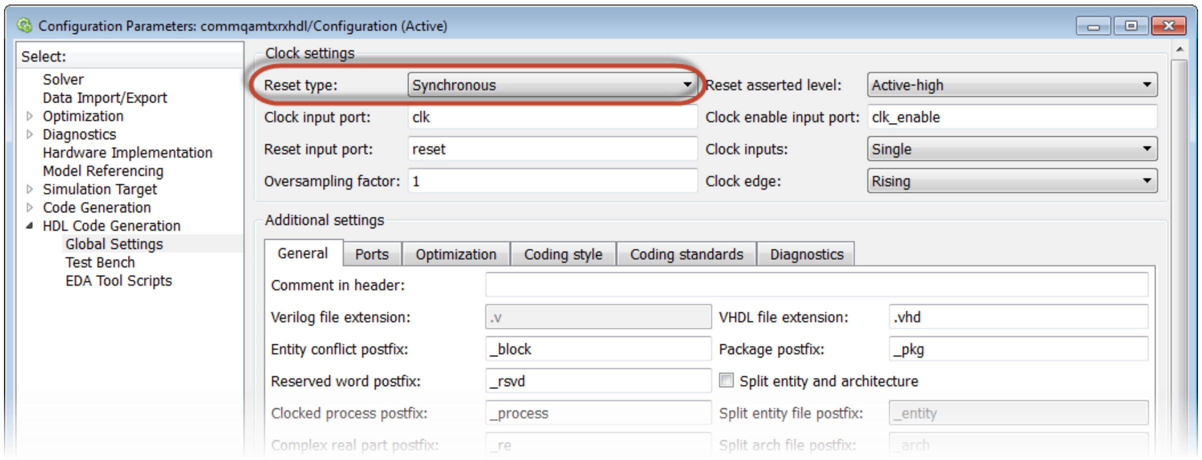
Example: AD009_RTDRRAM.slx



1.4.4 Use global reset type best suited for your target hardware

While a synthesis tool can faithfully implement either synchronous or asynchronous reset logic, matching the reset type to the underlying FPGA architecture will result in better resource utilization and performance.

- For Xilinx FPGA devices, use *synchronous* global reset.
- For Altera FPGA devices, use *asynchronous* global reset.
- The **Reset type** setting can be found in the **Global Settings** pane of the HDL Coder UI, or in HDL Workflow Advisor **3.1.2 Set Advanced Options**.


Configuration Parameters: commqamtxrxhdl/Configuration (Active)

Clock settings

Reset type:	Synchronous	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 and architecture	
Clocked process postfix:	_process	Split entity file postfix:	_entity
Complex real part postfix:	_re	Split arch file postfix:	_arch

2. Block Settings

2.1 Discontinuities

2.2 Discrete

2.2.1 *Appropriate use of various types of delay blocks as registers*

Example: BS005_delay.slx

1. The chart below shows the delay blocks that are usable for HDL code generation along with the parameters that can be set

Block name	Initial condition value type	ResetType support	Local reset port	Local enable port
Unit Delay	Scalar only	Supported	Unsupported	Unsupported
Delay *Specify number of samples to delay*	Possible to set each value with a vector value	Supported	Supported	Supported
Unit Delay Enabled	Scalar only	Supported	Unsupported	Supported [2]
Unit Delay Resettable	Scalar only	Unsupported	Supported [1] [2]	Unsupported
Unit Delay Enabled Resettable	Scalar only	Unsupported	Supported [1] [2]	Supported [2]
Tapped Delay	Possible to set each value with vector value. [3]	Supported	Unsupported	Supported

Footnotes:

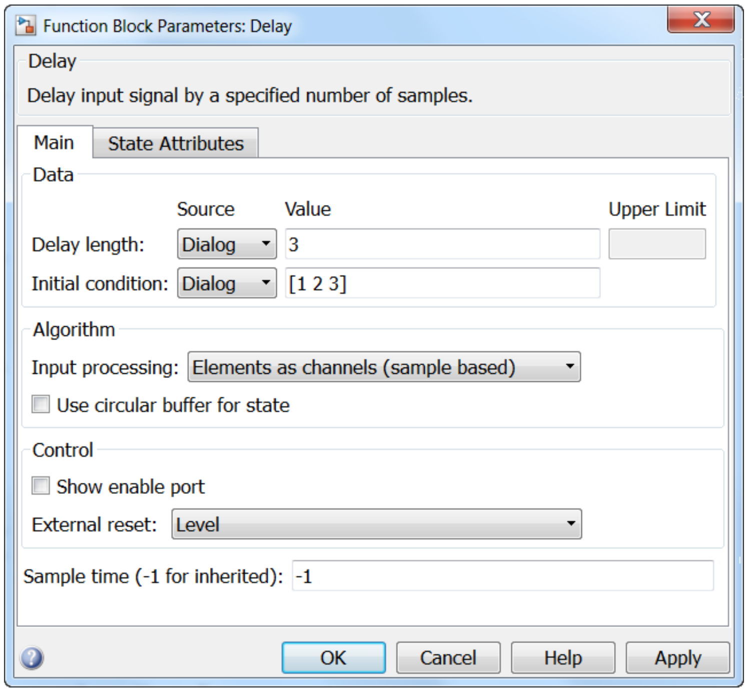
1. Use the `SoftReset` block parameter to specify whether to generate hardware-friendly synchronous reset logic, or local reset logic that matches the Simulink simulation behavior.

2. Input a Boolean signal into a Reset or an Enable port.

3. The number of elements in the **Initial condition** must match the **Number of delays parameter**, otherwise it will result in an error during HDL code generation.

2. Setting parameters for the Delay and Unit Delay blocks:

- For **Delay length**, set **Source** to `Dialog` and enter a scalar integer greater than 0 in the **Value** field.
- For **Initial Condition** set **Source** to `Dialog`. When the delay length is greater than 1, you can use a vector to set the initial value of each register.
- For **External reset**, set to `Level` to use a local reset port.
- Set **Input processing** to `Elements as channels (sample based)`.
- Set **Sample time** to `-1` (inherit). Changing sample time is not supported for code generation.



Delay

Delay input signal by a specified number of samples.

Data

	Source	Value	Upper Limit
Delay length:	Dialog	3	
Initial condition:	Dialog	[1 2 3]	

Algorithm

Input processing: Elements as channels (sample based)

☐ Use circular buffer for state

Control

☐ Show enable port

External reset: Level

Sample time (-1 for inherited): -1

3. Setting parameters for the Unit Delay Enabled block and Unit Delay Enabled Resettable block

- When the input to the Enable port is true (not 0), the output and state will be updated. When the input to the R (Reset) port is true, the reset condition will be applied and the state initialized.
- When the input signal is a vector signal, the initial value to each channel can be set by setting vector value for the **Initial condition** parameter.
- The E (Enable) port and the R (Reset) port require a Boolean signal as an input.

4. Setting parameters for the Tapped Delay block

   - This block acts as a model that connects multiple delay blocks as a shift register. When **Include current input in output vector** is un-checked and the **Number of delays** is set to 1, it will act as a Unit Delay block. In that case, if the **Initial condition** is set to multiple elements (vector), HDL code generation is unsupported.

5. Set up global default reset parameters

   To set up global synchronous or asynchronous reset, go to the pulldown menu **Code > HDL Code > Options…** and select **Global Settings** under **HDL Code Generation**. There you can set **Reset type** to Synchronous or Asynchronous, and also specify **Reset asserted level** to Active-high or Active-low.

6. Generating registers with no reset

   To specify register logic without reset, go into the block's **HDL Block Properties…** and specify `ResetType` as `none`**.** If there is no reset logic, mismatches between Simulink and the generated HDL can occur for a number of samples at the beginning of simulation before the register is loaded.

7. 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.

`SoftReset` setting	Type of reset	Matches Simulink simulation?
on	Synchronous reset	No
`off` [default]	Asynchronous reset	Yes

This property is available for the Unit Delay Resettable block or Unit Delay Enabled Resettable block. It is recommended practice to set this to `off` so that the generated HDL matches the Simulink simulation behavior.

8. Using RAM-based mapping instead of a shift register (FPGA targets only)

   To specify that a delay maps to RAM rather than a shift register, under the block's **HDL Block Properties…**, set `UseRAM` to on. For more information on when to use this setting, refer to 2.2.2 Map large delays to FPGA block RAM instead of registers to reduce area.

Generated code examples:

	VHDL	Verilog

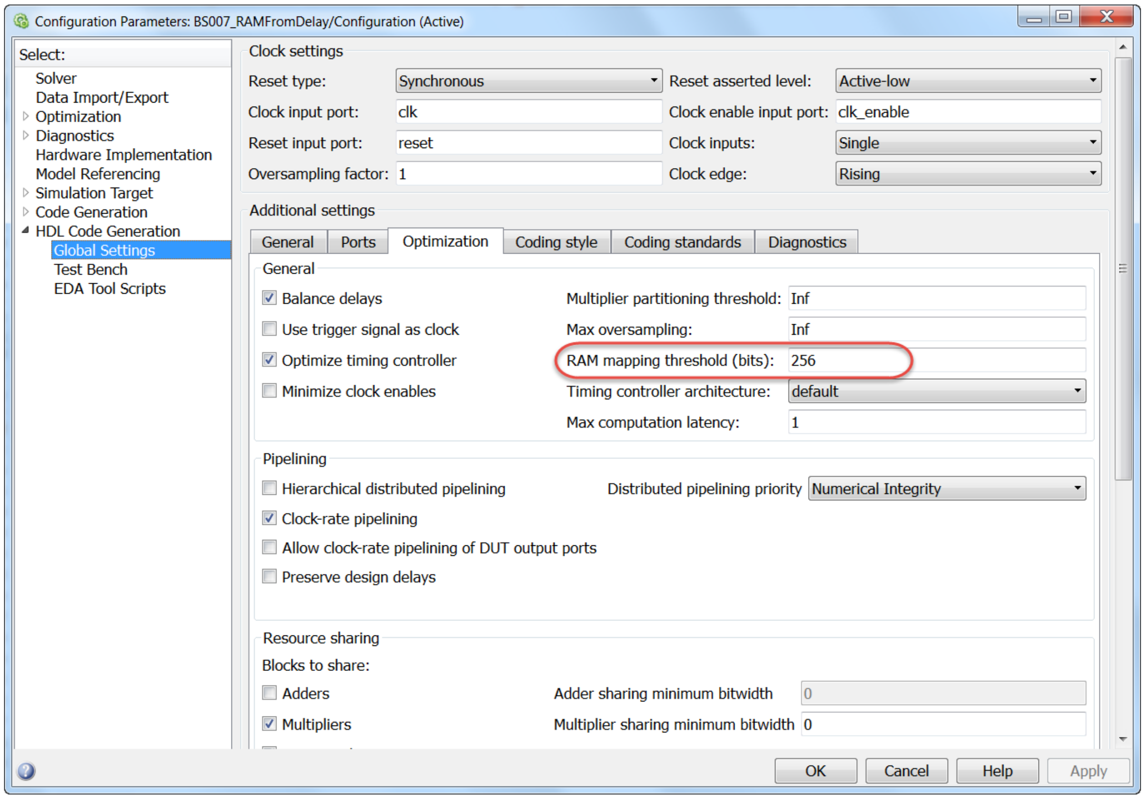
Register with no reset	`-- <S1>/Delay_wtout_rst`
	`Delay_wtout_rst_process : PROCESS (clk`
	`BEGIN`
	`  IF clk'EVENT AND clk = '1' THEN`
	`    Delay_wtout_rst_out1 <= In1_signed;`
	`  END IF;`
	`END PROCESS Delay_wtout_rst_process;`
		`// <S1>/Delay`
		`always @(posedge clk)`
		`  begin : Delay_process`
		`    Delay_out1 <= In1;`
		`   end`

Register with synchronous reset	`-- <S1>/Delay`
	`Delay_process : PROCESS (clk)`
	`BEGIN`
	`  IF clk'EVENT AND clk = '1' THEN`
	`    IF reset = '1' THEN`
	`      Delay_out1 <= to_signed(16#0001#, 16);`
	`    ELSE`
	`      Delay_out1 <= In1_signed;`
	`    END IF;`
	`  END IF;`
	`END PROCESS Delay_process;`
		`// <S1>/Delay`
		`always @(posedge clk)`
		`  begin : Delay_process`
		`    if (reset == 1'b1) begin`
		`      Delay_out1 <= 16'sb0000000000000001;`
		`    end`
		`    else begin`
		`      Delay_out1 <= In1;`
		`    end`
		`  end`

Register with asynchronous reset	`-- <S1>/Delay`
	`Delay_process : PROCESS (clk, reset)`
	`BEGIN`
	`  IF reset = '1' THEN`
	`    Delay_out1 <= to_signed(16#0001#, 16);`
	`  ELSIF clk'EVENT AND clk = '1' THEN`
	`    Delay_out1 <= In1_signed;`
	`  END IF;`
	`END PROCESS Delay_process;`
		`// <S1>/Delay`
		`always @(posedge clk or posedge reset)`
		`  begin : Delay_process`
		`    if (reset == 1'b1) begin`
		`      Delay_out1 <= 16'sb0000000000000001;`
		`    end`
		`    else begin`
		`      Delay_out1 <= In1;`
		`    end`
		`  end`

2.2.2 Map large delays to FPGA block RAM instead of registers to reduce area

By default, Delay blocks map to registers. For large amounts of delay, this can be costly in terms of FPGA resources. Often it is more efficient to map to the device's block or distributed RAM instead. In order to specify that a Delay block maps to RAM when it is larger than a specified threshold, under **HDL Block Properties…**, set `UseRam` to on. The `RAMMappingThreshold` can be set under the pulldown menu **Code > HDL Code > Options…** as shown here:


Configuration Parameters: BS007_RAMFromDelay/Configuration (Active)

Select:
- Solver
- Data Import/Export
- Optimization
- Diagnostics
- Hardware Implementation
- Model Referencing
- Simulation Target
- Code Generation
- HDL Code Generation
  - Global Settings
  - Test Bench
  - EDA Tool Scripts

Clock settings

Reset type: Synchronous
Reset asserted level: Active-low
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

General | Ports | Optimization | Coding style | Coding standards | Diagnostics

General

- [✓] Balance delays
- [ ] Use trigger signal as clock
- [✓] Optimize timing controller
- [ ] Minimize clock enables

Multiplier partitioning threshold: Inf
Max oversampling: Inf
RAM mapping threshold (bits): 256
Timing controller architecture: default
Max computation latency: 1

Pipelining

- [ ] Hierarchical distributed pipelining
- [✓] Clock-rate pipelining
- [ ] Allow clock-rate pipelining of DUT output ports
- [ ] Preserve design delays

Distributed pipelining priority: Numerical Integrity

Resource sharing

Blocks to share:

- [ ] Adders
- [✓] Multipliers

Adder sharing minimum bitwidth: 0
Multiplier sharing minimum bitwidth: 0

In this case, if the product `DelayLength * WordLength * ComplexLength` is greater than or equal to 256, it will be mapped to RAM.

- `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 the size of the RAM or ROM in your design is small, your synthesis tool may map the generated code to distributed RAM resources in the FPGA fabric, instead of block RAMs for better hardware performance. The threshold is tool-dependent, and can usually be configured within the synthesis tool.

Example: BS007_RAMFromDelay.slx

This example implements a 4096-sample delay in one Delay block with `UseRam=off` and one with `UseRam=on`. With the input signal being 16 bits wide, it requires a RAM size of 65,536. The following table shows the difference in resource usage when targeting an XC6vlx240t device:

Delay block configuration	Delay_RAM (`UseRAM = on`)			Delay_FF (`UseRAM = off`)		
	Slice FF	Slice LUT	RAMB36E1	Slice FF	Slice LUT	RAMB36E1
16bit x 4096word	41	36	2	4127	3845	0

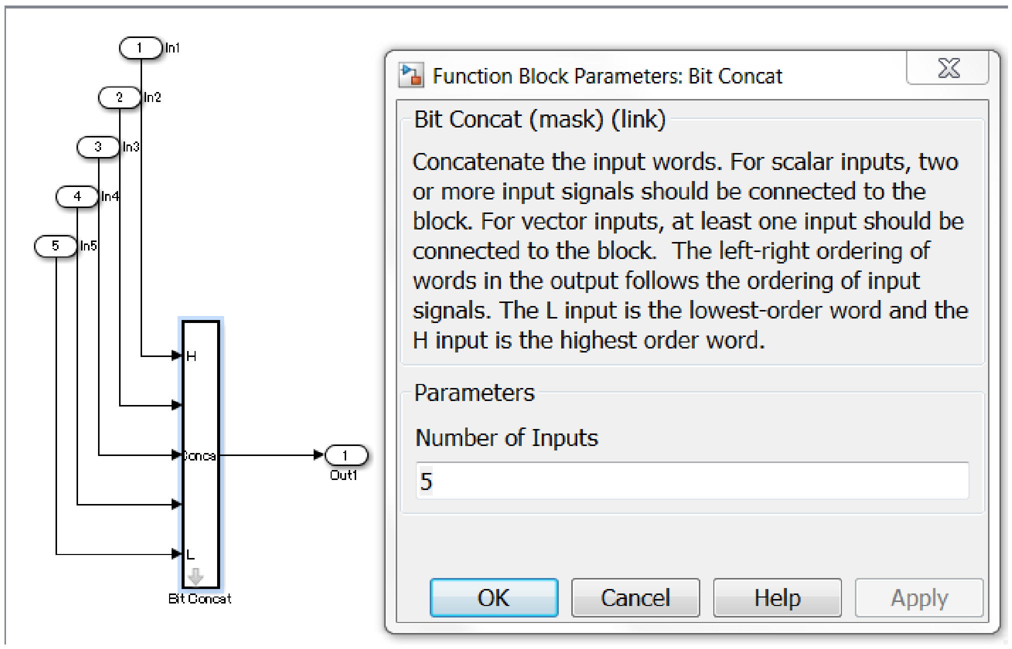
2.3 HDL Operations

2.3.1 *Use a Bit Concat block instead of a Mux block for bit concatenation in VHDL*

A Mux block or a Bit Concat block can be used in Simulink to combine signals into a single vector. However when combining scalar signals, a Mux block will create a VHDL declaration of `std_logic_vector(0 to n)`. This will result in violations or warnings from most HDL rule checkers since the typical bit order convention is `(n downto 0)`.

For this reason, it is recommended practice to use a Bit Concat block when creating a vector from Boolean or ufix1 signals.

The following figure shows the bit ordering for a Bit Concat block:



Function Block Parameters: Bit Concat

Bit Concat (mask) (link)

Concatenate the input words. For scalar inputs, two or more input signals should be connected to the block. For vector inputs, at least one input should be connected to the block. The left-right ordering of words in the output follows the ordering of input signals. The L input is the lowest-order word and the H input is the highest order word.

Parameters

Number of Inputs

5

VHDL generated from the Bit Concat block (correct)

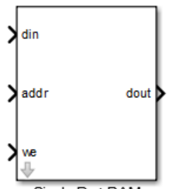
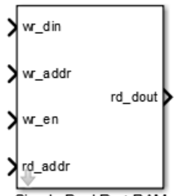
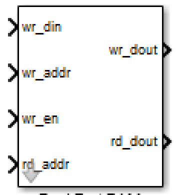
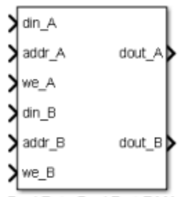
```
ENTITY DutBitconcat IS
  PORT( In1   :   IN    std_logic;
        In2   :   IN    std_logic;
        In3   :   IN    std_logic;
        In4   :   IN    std_logic;
        In5   :   IN    std_logic;
       Out1   :   OUT   std_logic_vector(4 DOWNTO 0)
      );
 END DutBitconcat;


ARCHITECTURE rtl OF DutBitconcat IS

  -- Signals
  SIGNAL Bit_Concat_out1 : unsigned(4 DOWNTO 0);   --
ufix5

 BEGIN
   -- <S2>/Bit Concat
   Bit_Concat_out1 <= unsigned'(In1 & In2 & In3 & In4 &
In5);

   Out1 <= std_logic_vector(Bit_Concat_out1);

 END rtl;
```

VHDL generated from the Mux block (incorrect)

```
ENTITY DutMux IS
  PORT( In1     :   IN    std_logic;
        In2     :   IN    std_logic;
        In3     :   IN    std_logic;
        In4     :   IN    std_logic;
        In5     :   IN    std_logic;
       Out1     :   OUT   std_logic_vector(0 TO 4)
         );
 END DutMux;


ARCHITECTURE rtl OF DutMux IS

  -- Signals
  SIGNAL Mux_out1                                         :
std_logic_vector(0 TO 4);  -- boolean [5]

 BEGIN
   -- <S3>/Mux
   Mux_out1(0) <= In1;
   Mux_out1(1) <= In2;
   Mux_out1(2) <= In3;
   Mux_out1(3) <= In4;
   Mux_out1(4) <= In5;

   Out1 <= Mux_out1;

 END rtl;
```

2.3.2 Design considerations for RAM Block access

There are four different RAM blocks available for use:

Circuit size Smaller	Single-port RAM	 Single Port RAM	Supports sequential read and write operations. If you want to model RAM that supports simultaneous read and write operations at different addresses, use the Dual Port RAM or Simple Dual Port RAM.
↓	Simple Dual Port RAM	 Simple Dual Port RAM	Supports simultaneous read and write operations, and has a single output port for read data. You can use this block to generate HDL code that maps to RAM in most FPGAs. The Simple Dual Port RAM is similar to the Dual Port RAM, but the Dual Port RAM has both a write data output port and a read data output port.
↓	Dual Port RAM	 Dual Port RAM	Supports simultaneous read and write operations, and has both a read data output port and write data output port. You can use this block to generate HDL code that maps to RAM in most FPGAs. If you do not need to use the write output data, wr_dout, you can achieve better RAM inference with synthesis tools by using the Simple Dual Port RAM block.
Larger	Dual Rate Dual Port RAM	 din_A addr_A we_A din_B addr_B we_B dout_A dout_B Dual Rate Dual Port RAM	Supports simultaneous read and write operations to different addresses at two clock rates. Port A of the RAM can run at one rate, and port B can run at a different rate (set Clock Inputs to "Multiple"). In high-performance hardware applications, you can use this block to access the RAM twice per clock cycle. If you generate HDL code, this block maps to a dual-clock dual-port RAM in most FPGAs.

RAM design considerations:

- You can set global RAM configuration parameters as needed under **HDL Code Generation > Global Settings > Coding Style**:
    - `Initialize all RAM blocks`: when on (default), all RAM signal outputs will initialize to '0' in simulation, and HDL code generation will create the initialization logic to do the same in HDL simulation. There are cases with FPGA synthesis tools where too large an initialization loop will generate an error messages. In those cases, set this parameter to `off` or remove the loop restriction in your FPGA synthesis tool.
    - `RAM Architecture`: Select `RAM with clock enable` (default) to generate a RAM that is connected to the global clock enable signal. Altera FPGA devices do not support RAM with clock enable, so this setting can be turned off when targeting those.
- In a Dual Rate Dual Port RAM block, concurrent access to the same address is forbidden, so you must add logic to prevent that from happening. See this implemented in *Example BS048_DRDPRAM.slx*. This example design sets the address of port B to 255 when addresses coincide with each other, and disables

write enable signal of port B when both are in write mode, giving priority to port A. Also to change sample rates in this example, adjust the parameters of the Repeat and Downsample blocks in the AddWeCheck subsystem.

2.3.3 HDL FIFO block usage considerations

The HDL FIFO block stores a sequence of input samples in a first in, first out (FIFO) register.

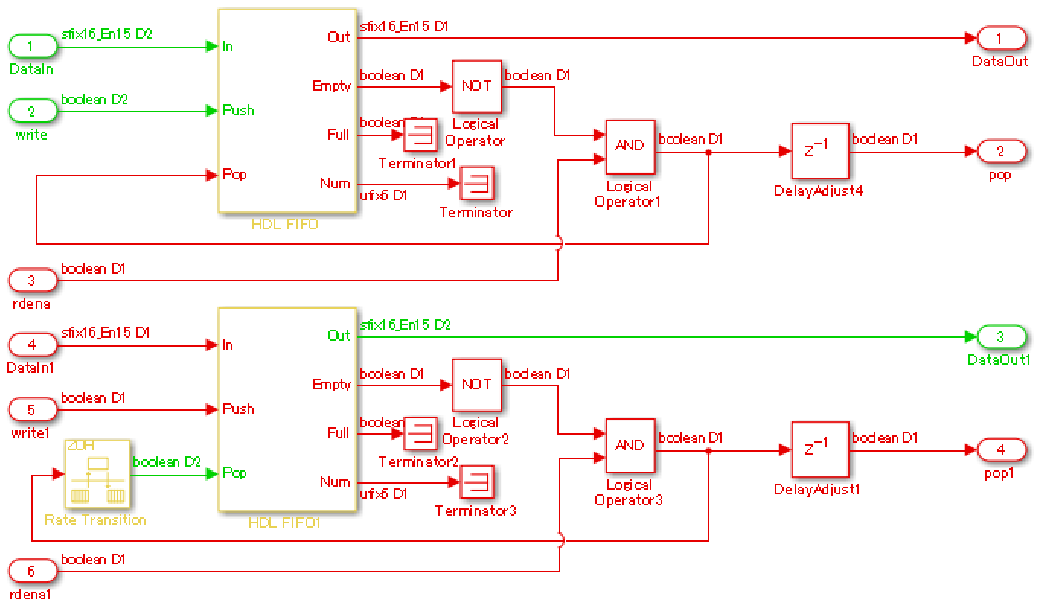
- Inputs (`In`, `Push`) and outputs (`Out`, `Pop`) can run at different sample times. Enter the ratio of output sample time to input sample time. Use a positive integer or `1/N`, where `N` is a positive integer. The default is `1`.

For example:

  - If you enter `2`, the output sample time is twice the input sample time, meaning the outputs run slower
  - If you enter `1/2`, the output sample time is half the input sample time, meaning the outputs run faster

The `Full`, `Empty`, and `Num` signals run at the faster rate.

In the following figure, when using the control output of FIFO in an input, perform a rate transition if needed. The configuration of the rate transition in *Example BS044_HDLFIFO.slx* is shown in the following figure:



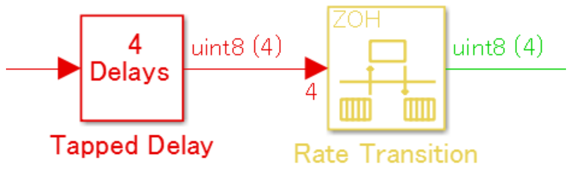
2.3.4 Parallel <--> Serial conversion

Starting with the 2014b release, the HDL Coder/HDL Operations library includes Serializer1D and Deserializer1D blocks. These are the recommended method for performing parallel-to-serial and serial-to-parallel conversion.

Prior to R2014b, the following are manual methods for converting a signal between parallel and serial processing:

Example: BS027_serial2Parallel.slx

- Serial-to-parallel: Use a Rate Transition block followed by a Tapped Delay block as shown:



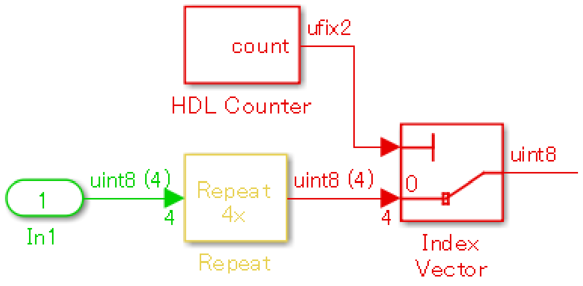
Set the Tapped Delay block parameters as follows:

- **Number of delays:** set to be equal to the amount of parallel outputs
- **Order output vector starting with:** `Oldest`
- **Include current input in output vector:** not selected

Set the Rate Transition block parameters as follows:

- **Output port sample time options:** Multiple of input port sample time
- **Sample time multiple(>0):** set to be equal to the amount of parallel outputs

- Parallel-to-serial : Use a Repeat block followed by an Index Vector block as shown :



Set the Repeat block parameters as follows:

- **Repetition count:** set to be equal to the amount of parallel inputs

  For more on trading off speed vs area when using a Repeat block, see 2.8.1 Rate conversion blocks and usage

Set the Index Vector block parameters as follows:

- **Data port order:** `Zero-based contiguous`

2.4 Logic and bit operations

2.4.1 Logical vs. arithmetic bit shift operations

The shift operation provided in multiple Simulink blocks and MATLAB functions, the left logical shift is the same as left arithmetic shift, however the right logical shift is different from the right arithmetic shift. The following table summarizes the differences:

Block/function name	Parameter/operation	Verilog	VHDL	Explanation
HDL Operations/ Bit Shift [1]	Shift Left Logical/ logical left shift	`<<<` [2]	`sll`	In the case of signed data and a positive value, the input signal does

				not maintain a sign bit. 0 goes into the empty bit on LSB side.
	Shift Right Logical/ logical right shift	`>>`	`srl`	Don't maintain a sign bit. 0 goes into the empty bit on MSB side. Because the sign bit shifts to right-hand side, a negative value turns into a positive value.
	Shift Right Arithmetic/ Arithmetic right shift	`>>>`	`SHIFT_RIGHT`	When an input is a signed data type, a sign bit is maintained and other bits shift to the right.
Shift Arithmetic `bitshift` function	Positive value (shift right) / Arithmetic right shift	`>>>`	`SHIFT_RIGHT`	When an input is a signed data type, a sign bit is maintained and other bits shift to the right.
	Negative value (shift left) / Arithmetic left shift	`<<<`	`sll` [3]	0 goes into the empty bit on LSB side. In the case of a positive value, an input signal does not maintain a sign bit for signed data. Don't check overflow and underflow.
`bitsll` function	None/logical left shift	`<<<` [2]	`sll`	In the case of a positive value, an input signal does not maintain a sign bit for signed data. Don't check overflow and underflow.
`bitsrl` function	None/logical right shift	`>>`	`srl`	Don't maintain a sign bit.
`bitsra` function	None/arithmetic right shift	`>>>`	`SHIFT_RIGHT`	Maintain a sign bit, when an input signal is a signed data type.

Footnotes:

1. Because the Bit Shift block uses the `bitsll`, `bitsrl`, and `bitsra` functions inside, the code generated from these functions is the same.
2. In MATLAB and Verilog, because the operation of an arithmetic left shift and a logical left shift is the same, it is not an issue that a logical left shift model generates code that uses an arithmetic left shift.
3. In VHDL, an arithmetic left shift (`SHIFT_LEFT`) and logical shift (`sll`) are the same.

The difference between a logical right shift and an arithmetic right shift is whether it holds the sign bit or not. For signed data types, this is the MSB. Although in a logical right shift, the sign bit is shifted to the right and a 0 goes into the MSB, in an arithmetic right shift the MSB is maintained and also shifted to the right.

Example:

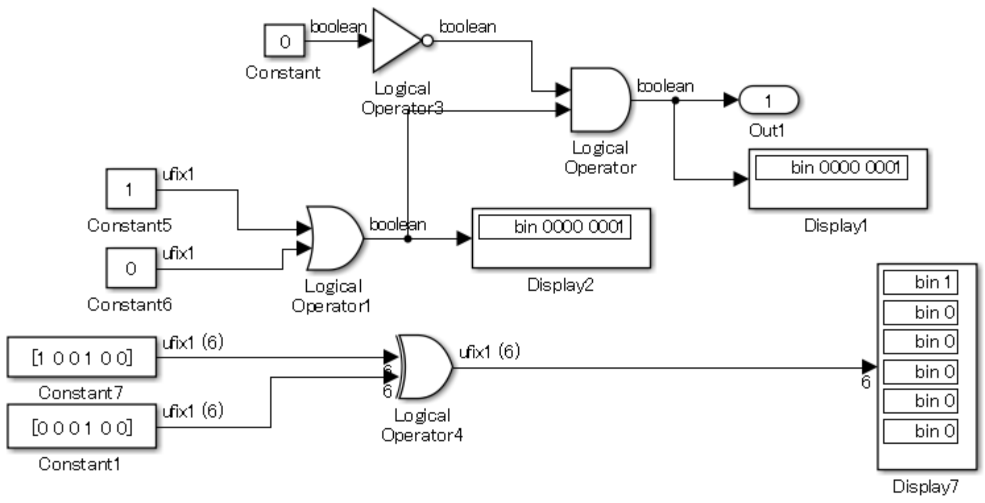
```
>> A = fi([], 1, 4, 0, 'bin','1011');
>> B = bitsrl(A, 2)  %  logic right shift
B =
     2
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 4
        FractionLength: 0

>> B.bin
ans =
0010

>> C=bitsra(A, 2)  %  arithmetic right shift
C =
    -2
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 4
        FractionLength: 0

>> C.bin
ans =
1110
```

2.4.2 Logical Operator, Bitwise Operator, and Bit Reduce for logic operations

Example: BS017_logical.slx

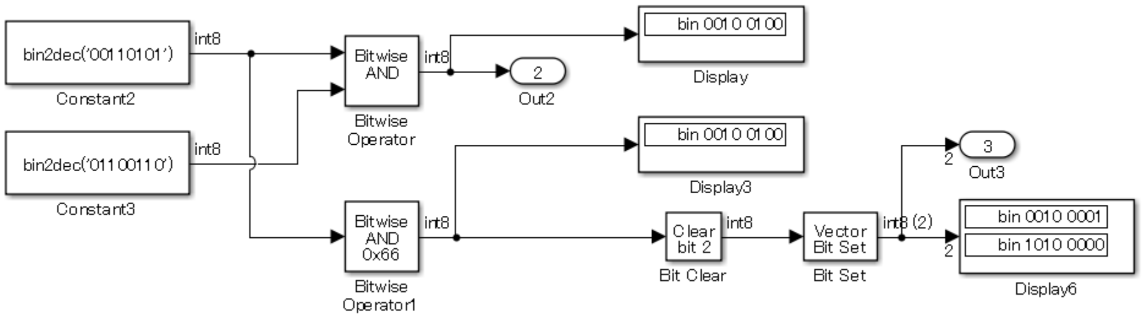
- Single-bit logic operations

For single-bit logic operations using `Boolean` or `ufix1` data types, use a Logical Operator block. Set the **Icon shape** to `distinctive` to view it as a logical circuit symbol. Note that this block also supports vectors of `Boolean` or `ufix1`.



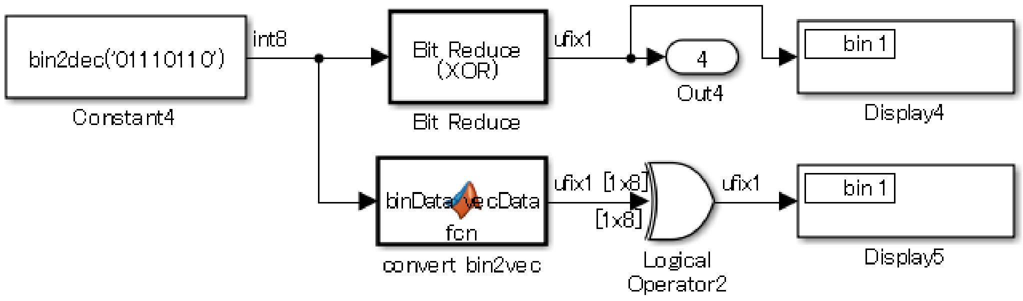
- Bitwise operations on two or more bits

For bitwise operations using `integer` or `ufix1` data types, use the Bitwise Operator block.



- Reduction operation

To perform a bit-by-bit reduction operation on a vector of `Boolean` or `ufix1` that returns a 1-bit value, use the Bit Reduce block. In the following example, the MATLAB function block uses a bitslice functions to convert the 8-bit input to a vector of 8 1-bit `ufix1` elements:



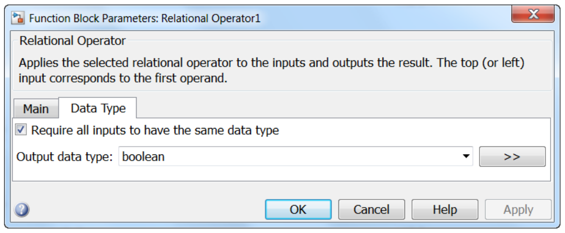
MATLAB Function block:

```
function vecData = fcn(binData)
%#codegen

nt = numerictype(binData);
vecData = fi(zeros([1 8]), 0,1,0);

% for n = 1:8  % Can't use for loop
%     vecData(n) = bitsliceget(binData, n,n);
% end
vecData(1) = bitsliceget(binData, 1,1);
vecData(2) = bitsliceget(binData, 2,2);
vecData(3) = bitsliceget(binData, 3,3);
vecData(4) = bitsliceget(binData, 4,4);
vecData(5) = bitsliceget(binData, 5,5);
vecData(6) = bitsliceget(binData, 6,6);
vecData(7) = bitsliceget(binData, 7,7);
vecData(8) = bitsliceget(binData, 8,8);
```

2.4.3 Use Boolean data type for the output of the Compare to Constant/Zero and the Relational Operator blocks

The output of the Compare to Constant, Compare to Zero, and Relational Operator blocks can be set to either `boolean` or `uint8`. Since the result of these operations will be 0 or 1, it is more efficient to output a `boolean` type since only the LSB will be connected in the generated HDL.

For a Relational Operator block, it is good practice to make sure that both inputs are of the same data type to avoid unintended truncation of bits (for instance a sign bit) that could cause simulation mismatches in the HDL:


Function Block Parameters: Relational Operator1

Relational Operator

Applies the selected relational operator to the inputs and outputs the result. The top (or left) input corresponds to the first operand.

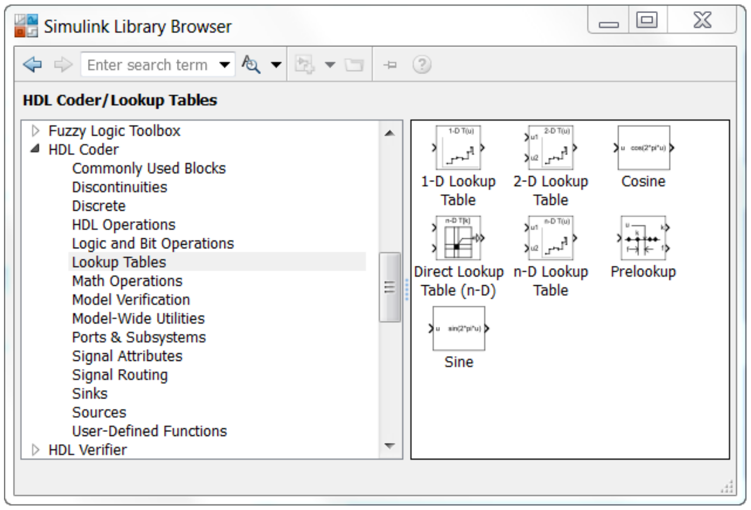
☑ Require all inputs to have the same data type

Output data type: boolean

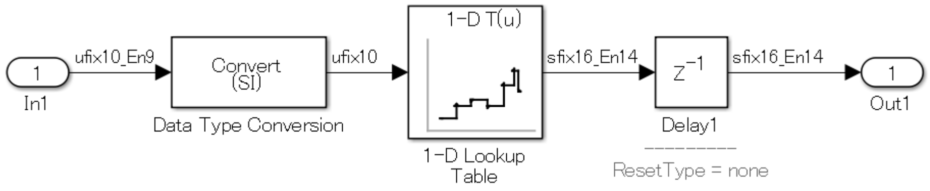
2.5 Lookup tables

2.5.1 Set the number of Lookup Table data entries to a power of 2 to avoid generation of a division operator (/)

For the blocks in the Simulink HDL Coder library that use lookup tables:



- Set the number of data points (the **Breakpoints** parameter) in a Lookup Table block to be a power of 2. This will prevent generation of a divide operator "/" in the HDL, which would require extra logic.
- Use a Data Type Conversion block with the **Input and output to have equal** parameter set to `Stored Integer` to convert input data into an integer data type (e.g. `fixdt(0, 10, 0)`) and integer data (e.g. [0 : 1 : (2^10-1)]) for the **Breakpoints** parameter to avoid generating a divide operator.

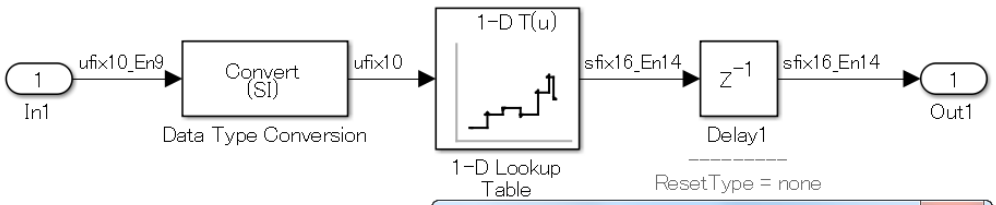


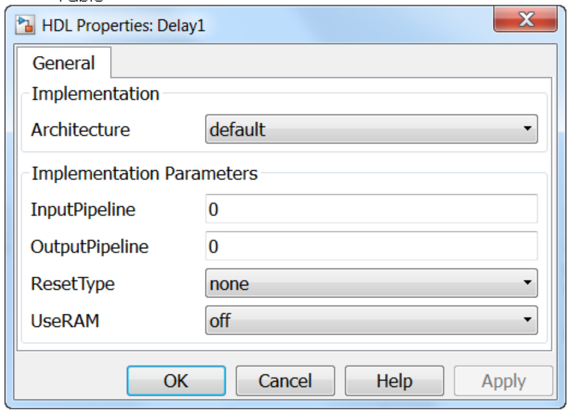
Example: BS020_LUT.slx

2.5.2 Generating FPGA block RAM from a Lookup Table block

In order to target a Lookup Table block to the Block RAM of an FPGA, put in a Delay (Unit) block after the Lookup Table, and set `ResetType` as none.

Example: BS021_LUTBRAM.slx




HDL Properties: Delay1

General

Implementation

Architecture: default

Implementation Parameters

InputPipeline	0
OutputPipeline	0
ResetType	none
UseRAM	off

The following table illustrates the difference in resource utilization between mapping to Block RAM and not, using an Altera Cyclone IV E, EP4CE115F23C7 device:

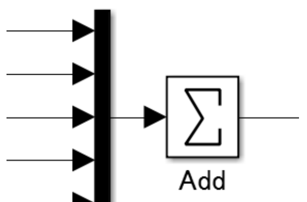
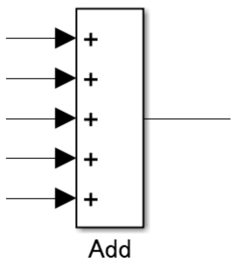
	Lookup Table mapped to Block RAM	Lookup Table mapped to logic
Model input/output data type	10-bit input 16-bit output	10-bit input 16-bit output
Circuit resources	Logic elements : 19 Memory bits : 12,288	Logic elements : 589 Total registers : 26 Memory bits : 0
Fmax(1200mV 85C)	389.71 MHz	181.72 MHz

2.6 Math operations

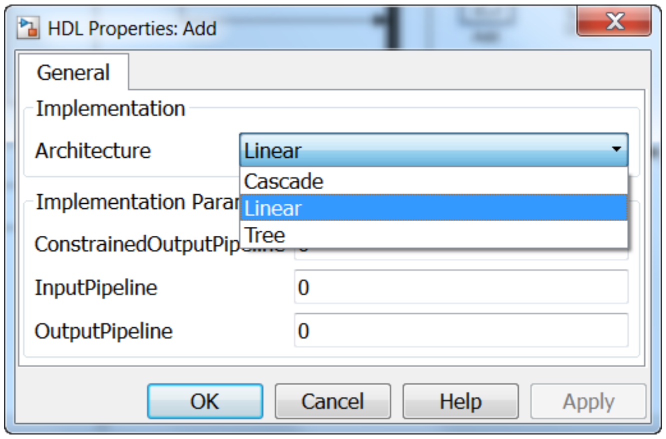
2.6.1 Input vector with Mux block to multi-input adder, multi-input product, and multi-input Min/Max

When using a multi-input adder, multi-input product, or multi-input min/max with 3 or more inputs, use a mux block to create a vector for the input to the operation.

Example: AD011_multinAddProd.slx

Correct: Adding a vector signal input	Incorrect: -- Adding scalar signal inputs
	

You can still change the **Architecture** in **HDL Block Properties…** as shown:



Implementation

Architecture: Linear

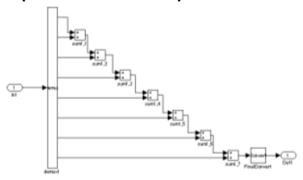
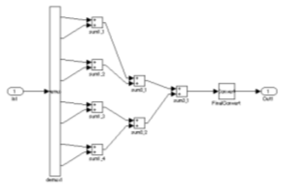
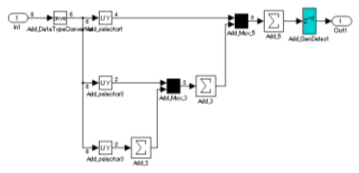
Cascade
Linear
Tree

Implementation Parar

ConstrainedOutputPip

InputPipeline: 0

OutputPipeline: 0

	Linear	Tree	Cascade
Features	• Large circuit • Requires one operator per input	• Smaller circuit than Linear • Pipeline insertion is possible between adders (see 4.2.3) • Requires one operator per input	• Resources can be shared to reduce area • Requires faster clock speed (see table below)
Example generated circuit	7 paths with 8-input adders 	3 paths with 8-input adders 	

Number of resources and over-clocking required for Linear/Tree/Cascade adder / product:

Number of inputs	Number of operators in Linear/Tree	Cascade		
		Over-clock	Number of operators (Multiplication or addition)	For control Number of adders
3	2	3	1	1
4	3	3	2	1
5	4	4	2	1
6	5	4	2	2
7	6	4	3	2
8	7	5	3	2
9	8	5	3	2
10	9	5	3	3
11	10	5	4	3
12	11	6	4	3
13	12	6	4	3
14	13	6	4	3
15	14	6	4	4
16	15	6	5	4
17	16	7	5	4
18	17	7	5	4
19	18	7	5	4
20	19	7	5	4

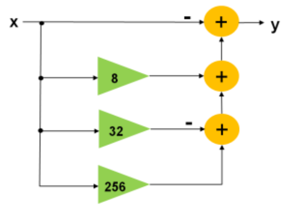
Because one sample of latency will be added, it is good practice to add a Delay (Unit) to the output of the original model as described in 2.6.9 Model the delay of blocks that will be auto-pipelined (Divide, Sqrt, Trigonometric Function, Cascade Add/Product, Viterbi Decoder).

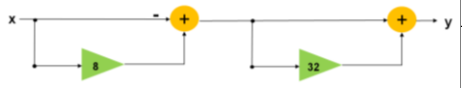
2.6.2 Set ConstMultiplierOptimization to 'auto' for a Gain block

For most optimal resource usage when using a Gain block, in the blocks HDL Block Properties…, set `ConstMultiplierOptimization` to `'auto'`. This will allow HDL Coder to choose between the best optimization techniques for the number of adders required by the multiplication operation. This setting tries to avoid using multipliers since they tend to be resource-intensive on the device. The following example highlights the differences:

Example: BS004_gainProperty.slx

HDL block property	Implementation method	Generated code example

`csd`	Resize the input data in parallel, then subtract and add the result.
	
		`-- <S1>/Gain`
		`-- CSD Encoding(231): 1001'01001'; Cost (Adders) = 3`
		` Gain_mul_temp <= ((resize(Delay_out1 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 21) - resize(Delay_out1 & '0' & '0' & '0' & '0' & '0', 21)) + resize(Delay_out1 & '0' & '0' & '0', 21)) - resize(Delay_out1, 21);`
		` Gain_out1 <= Gain_mul_temp(19 DOWNTO 0);`
		
		`// CSD Encoding (231):1001'01001'; Cost (Adders) = 3`
		`assign Gain_mul_temp  = (($signed({Delay_out1, 8'b00000000}) - $signed({Delay_out1, 5'b00000})) + $signed({Delay_out1, 3'b000})) - Delay_out1;`
		` assign Gain_out1 = Gain_mul_temp[19:0];`

`fcsd`	Multiple cascaded additions of the input data together with its resized data.
	
		` -- <S1>/Gain1`
		` -- FCSD for 231 = 33 X 7; Total Cost = 2`
		` -- CSD Encoding (33) : 0100001; Cost (Adders) = 1`
		` Gain1_factor <= resize(Delay_out1 & '0' & '0' & '0' & '0' & '0', 21) + resize(Delay_out1, 21);`
		` -- CSD Encoding (7) : 1001'; Cost (Adders) = 1`
		` Gain1_mul_temp <= resize(Gain1_factor & '0' & '0' & '0', 21) - Gain1_factor;`
		` Gain1_out1 <= Gain1_mul_temp(19 DOWNTO 0);`
		
		` // FCSD for 231 = 33 X 7; Total Cost = 2`
		` // CSD Encoding (33) : 0100001; Cost (Adders) = 1`
		` assign Gain1_factor = $signed({Delay_out1, 5'b00000}) + Delay_out1;`
		` // CSD Encoding (7) : 1001'; Cost (Adders) = 1`
		` assign Gain1_mul_temp = $signed({Gain1_factor, 3'b000}) - Gain1_factor;`
		` assign Gain1_out1 = Gain1_mul_temp[19:0];`

`auto`	Auto-select `csd` or `fcsd`, depending on which uses the fewest adders.	Same as `csd` or `fcsd`

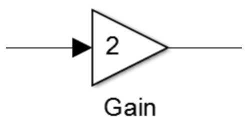
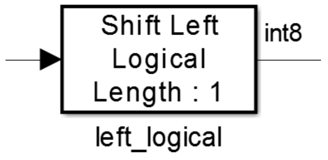
`none`	Operator (*)	` -- <S1>/Gain3`
		` Gain3_mul_temp <= to_signed(2#011100111#, 9) * Delay_out1;`
		` Gain3_out1 <= Gain3_mul_temp(19 DOWNTO 0);`
		
		` assign Gain3_mul_temp = 231 * Delay_out1;`
		` assign Gain3_out1 = Gain3_mul_temp[19:0];`

2.6.3 Use the Bit Shift block or the bitshift function for computations of the power of 2 (ASIC)

When performing a constant multiplication of the power of 2, it is more efficient in ASIC hardware to use a bit shift operation instead of performing a multiplication (there is little difference in FPGA hardware). Therefore in Simulink, it is best to utilize the Bit Shift block rather than a Gain or Product block. In MATLAB you can use the `bitshift` function.

Take care to pay attention to right-shift functionality when dividing by a power of 2, as described in section 2.4.1.

Examples: BS010_MLFGain.slx, BS053_gainShift.slx

Gain block	Bit Shift block
	
VHDL generated from a Gain block	VHDL generated from a Bit Shift block

`SIGNAL in1      : signed(7 DOWNTO 0);  -- int8`
`SIGNAL gcast    : signed(15 DOWNTO 0);  -- sfix16_En4`
`SIGNAL out1     : signed(7 DOWNTO 0);  -- int8`
 	
`gcast <= resize(in1 & '0' & '0' & '0' & '0' & '0' & '0',`
`16);`
`out1 <= gcast(11 DOWNTO 4);`
	`SIGNAL in1      : signed(7 DOWNTO 0);  -- int8`
	`SIGNAL out1     : signed(7 DOWNTO 0);  -- int8`
	
	`out1 <= in1 sll 2;`

Verilog generated from a Gain block	Verilog generated from a Bit Shift block

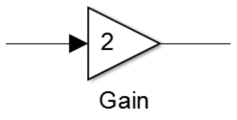
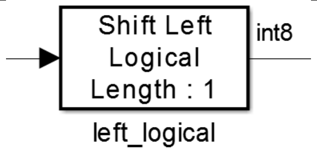
`wire signed [7:0] in;  // int8`
`wire signed [15:0] gcast;  // sfix16_En4`
`wire signed [7:0] out1;  // int8`
 	
`assign gcast = {{2{in1[7]}}, {in1, 6'b000000}};`
`assign out1 = gcast[11:4];`
	`wire signed [7:0] in;  // int8`
	`wire signed [7:0] out1;  // int8`
	
	`assign out1 = in1 <<< 2;`

2.6.4 Use Gain block for computations of the power of 2 (FPGA)

When performing a constant multiplication of the power of 2 on an FPGA, since the hardware utilizations are similar, it is safer to use a Gain block in Simulink or the * operator in MATLAB due to the bit shift differences described in section 2.4.1.

For best results, set **Output data type** to `Inherit` to avoid bit truncation and set the **Parameter data type** to be the minimum bit width required for a gain operation. For example if the **Gain** parameter is 2, set the **Parameter data type** to `fixdt(0,1,-1)`.

Examples: BS010_MLFGain.slx, BS053_gainShift.slx

Gain block	Bit Shift block
	
VHDL generated from a Gain block	VHDL generated from a Bit Shift block

`SIGNAL in1      : signed(7 DOWNTO 0);  -- int8`
`SIGNAL gcast    : signed(15 DOWNTO 0);  -- sfix16_En4`
`SIGNAL out1     : signed(7 DOWNTO 0);  -- int8`
 	
`gcast <= resize(in1 & '0' & '0' & '0' & '0' & '0' & '0',`
`16);`
`out1 <= gcast(11 DOWNTO 4);`
	`SIGNAL in1      : signed(7 DOWNTO 0);  -- int8`
	`SIGNAL out1     : signed(7 DOWNTO 0);  -- int8`
	
	`out1 <= in1 sll 2;`

Verilog generated from a Gain block	Verilog generated from a Bit Shift block

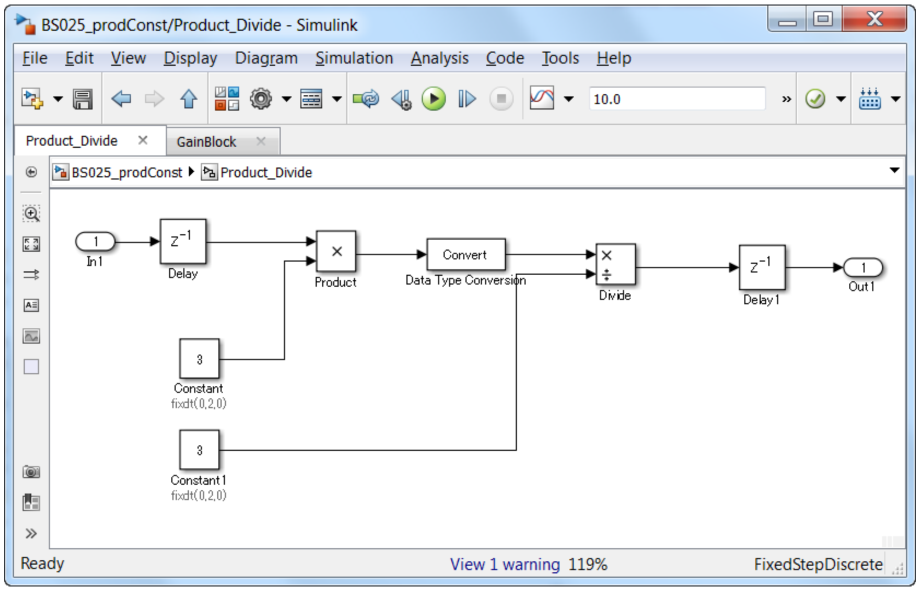
`wire signed [7:0] in;  // int8`
`wire signed [15:0] gcast;  // sfix16_En4`
`wire signed [7:0] out1;  // int8`
 	
`assign gcast = {{2{in1[7]}}, {in1, 6'b000000}};`
`assign out1 = gcast[11:4];`
	`wire signed [7:0] in;  // int8`
	`wire signed [7:0] out1;  // int8`
	
	`assign out1 = in1 <<< 2;`

2.6.5 Use a Gain block for constant multiplication and constant division

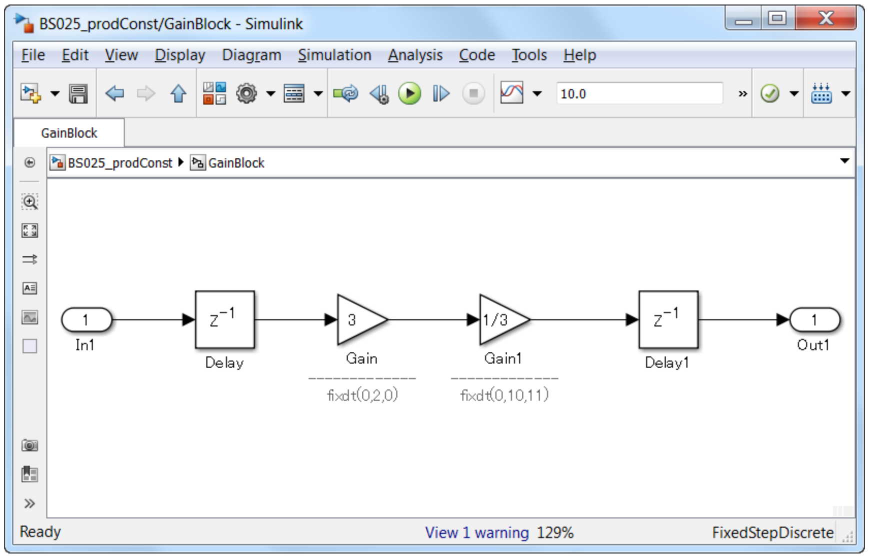
When a one of the factors in a multiplication operation is a constant, it is more efficient in hardware to use a Gain block instead of a Product block.

Similarly when the divisor is a constant in a division operation, it is more efficient in hardware to use a Gain block with the **Gain** parameter being the reciprocal of the divisor.

Example: BS025_prodConst.slx



And for optimal hardware resource usage with a Gain block, refer to the settings described in section 2.6.2.


BS025_prodConst/GainBlock - Simulink
File Edit View Display Diagram Simulation Analysis Code Tools Help
10.0
GainBlock
BS025_prodConst ▸ GainBlock
1
In1
z^{-1}
Delay
3
Gain
fixdt(0,2,0)
1/3
Gain1
fixdt(0,10,11)
z^{-1}
Delay1
1
Out1
Ready
View 1 warning 129%
FixedStepDiscrete

When using the Gain block for division, because the **Gain** parameter is an inverse ratio of the original divisor, be sure to properly set the fixed-point setting in **Parameter data type**. If the following MATLAB commands are executed, optimization of fixed-point scaling is performed automatically and accuracy can be checked:

```
>> format long
>> A=1/3;
>> B=fi(A, 0, 10)
B =
   0.333496093750000

          DataTypeMode: Fixed-point: binary point scaling
```

```
              Signedness: Unsigned
              WordLength: 10
          FractionLength: 11
>> err = (A-double(B))/A
err =
    -4.882812500000555e-04
```

2.6.6 Efficient multiplier design for targeting Altera DSP block

Altera's Cyclone IV series offers an Embedded Multiplier, which is a configuration of Register->Product->Register. This can be used as an 18-bit output for the product of two 9-bit inputs, or a 36-bit output for the product of two 18-bit inputs. Consult Altera's web site for more information on the Embedded Multipliers in Cyclone IV Devices.

The DSP block on the Stratix series (- IV) and Arria series (- II) devices is composed of two Half DSP blocks. Each of these Half DSP blocks contain a register bank before and after four 18x18-bit products, three adder stages, and a compute element. Consult Altera's web site for more information on the Arria II DSP architecture and the Stratix IV DSP architecture.

The internal resources of this Half DSP block can be used in various modes. The number of multipliers which can be used per compute mode is shown in the following table.

Arithmetic contents	Bit width of product inputs / output	Arithmetic operations per block	1st addition/subtraction step	2nd addition/subtraction step
A*B	9x9 / 18 12x12 / 24 18x18 / 36 36x36 / 72 54x54 floating point / 108	8 6 4 2 2	- - - - -	- - - - -
A*B+C*D	18x18 / 36	4	Addition/subtraction	-
(A*C-B*D) +(A*D-B*C)j	18x18 / 36	2	Addition/subtraction	
(A*B+C*D) +(E*F+G*H)	18x18 / 36	2	Addition/subtraction	Addition
A*B+C*D	18x36 / 55	2	-	Addition

The DSP block configuration changed in the Stratix V, Arria V, and Cyclone V series devices.

2.6.7 Efficient multiplier design for targeting Xilinx DSP48 slices

- Circuit architecture and input/output data type of a DSP slice

The Xilinx DSP48 and DSP48E slices are composed of a register bank→product→register bank→adder. The DSP48A and DSP48E1 have a pre-adder and are composed of register bank→adder→register bank→product→register bank→adder→register bank. So for designs that require a register before and after a multiplier and subsequent adder, only one DSP slice is required. The configurations with the pre-adder are useful for algorithms such as a symmetrical FIR filter.

Implementation with minimal resources and maximum accuracy can be attained by ensuring that the input/output bit width matches a DSP slice configuration. For example inputs that are 9x9 or 18x18 can map to one DSP slice, whereas larger bit width inputs would require two or more slices, depending on the device and synthesis options.

The following table summarizes the features of the various 7 Series DSP48 slices with links to their user guides:

DSP Slice name	Typical Device	Input bit width	Multiplication output	Pre-adder (bit width)
DSP48	Virtex-4	18x18	48	None

DSP48A	Spartan6	18x18	48	Yes (18x18)
DSP48E	Virtex-5	25x18	48	None
DSP48E1	Virtex-6 Virtex-7	25x18	48	Yes (25x25)

For more information on the DSP48 architecture, see the Xilinx XtremeDSP 48 Slice data sheet.

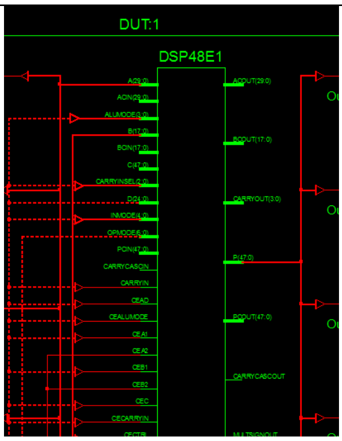
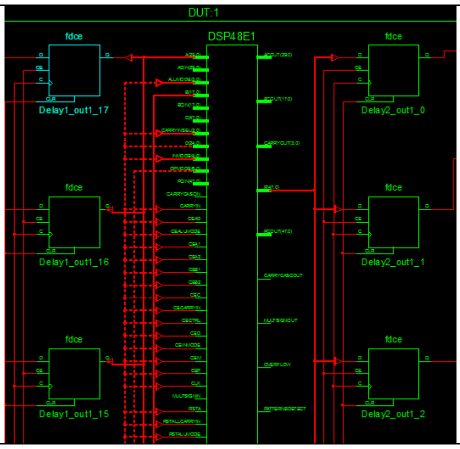
Additionally, because saturate and round are arithmetic operations, performing those outside of the multiplier will help ensure mapping to a DSP48 slice.

- Reset type preferences

  In order to map the registers before and after a multiplier to the registers in a DSP48 slice, set **HDL Code Generation>Global Settings>Reset type** to `Synchronous`. If asynchronous reset it used, it will not be able to utilize the registers inside the DSP slices, resulting in inefficient hardware as shown in the following comparison:

Reset type	Synchronous	Asynchronous
Synthesized result	Efficient and high-speed, because registers inside the DSP slice are used	Because registers inside the DSP slice are not used, it is slow and inefficient

Post-mapping number of resources	Registers = 0
	
	DSP48 = 1
		Registers = 72 (18+18+36)
		
		DSP48 = 1

Results in the Xilinx ISE Technology Map Viewer		 DUT:1 DSP48E1 fdce Delay1_out1_17 Delay1_out1_16 Delay1_out1_15 Delay2_out1_0 Delay2_out1_1 Delay2_out1_2

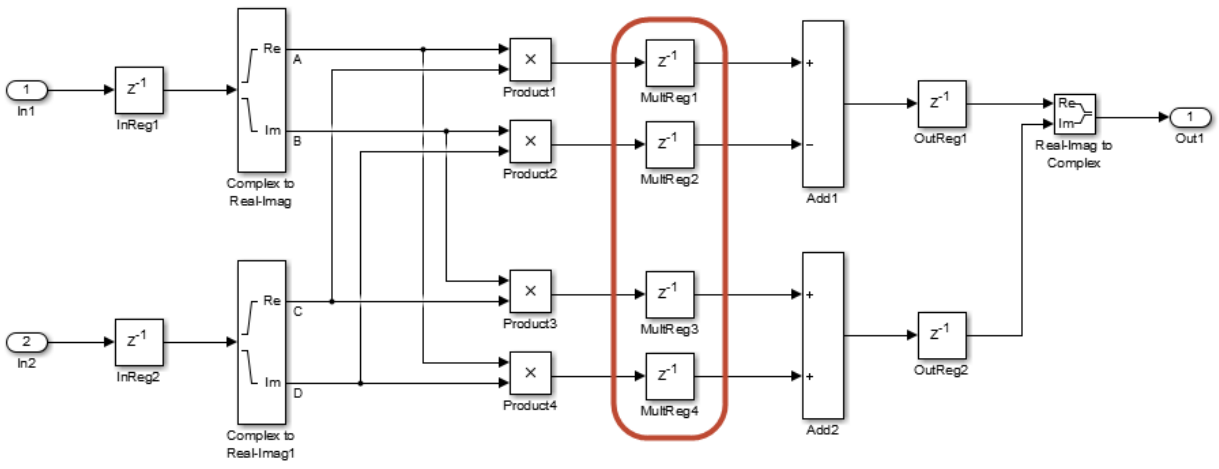
- *Example BS057_multAdd2DSP48E1.slx* demonstrates how to configure the inputs and outputs of a pre-add→product→add with Delay blocks so that it will map to one DSP48E1 slice.

2.6.8 Consider speed/area priority and DSP mapping when modeling complex multiplication

A complex multiply is a series of multiply and multiply-add operations:

$$(A + Bj)*(C+Dj) = (A*C - B*D) + (A*D + B*C)j$$

Instead of using the Simulink product block, consider elaborating the real and imaginary components in order to accommodate the bit growth from the adder, and to pipeline the multiply-add operations, as shown in the figure below.



- Pipelining guidelines:
    - Xilinx ISE and Vivado can map up to 2 input registers, 1 multiplier output register and 1 adder output register into the DSP48 block. You may get better speed performance by using 2 multiplier output registers, and let the synthesis tool re-time the extra level as appropriate.
    - Altera Quartus II can map 1-2 input register and 1 adder output register into the DSP block, depending on the device family. Most families do not have pipeline register between the multiplier and the adder; if you use one in your design, Quartus II may re-time it to the adder output in order to map the adder inside the DSP block.

The following table compares the approach of manual register distribution versus using input/output pipelining or Delay blocks and using automated distributed pipelining. For details, refer to section 4.2.1.

Optimization priority	Configure registers manually	Use distributed pipelining with inserted Delay block or input/output pipelining
Number of multiplications	HDL_complex_Multiplier_01 block (three multiplications)	None
Speed	HDL_complex_Multiplier_02 block (four multiplications)	HDL_Complex_Multiplier_03 block (one Product block, four multiplications)

Example: BS056_complexMultiplier.slx

The following table compares synthesis results for 18-bit inputs:

Target device	Block configuration	Synthesis result
Xilinx Virtex-6 XC6VLX240T-1FFG1156	HDL_complex_Multiplier_01 (Three multiplications)	Slice Registers: 331 Slice LUTs: 202 DSP48E1: 3 Fmax: 163MHz
	HDL_complex_Multiplier_02 HDL_complex_Multiplier_03 (Four multiplications)	DSP48E1: 4 Fmax: 263 MHz
Altera Stratix IV EP4SGX230KF40C2	HDL_complex_Multiplier_01 (Three multiplications)	Registers: 365 Combinational ALUTs: 260 DSP block 18bit: 8 Fmax: 254 MHz
	HDL_complex_Multiplier_02 HDL_complex_Multiplier_03 (Four multiplications)	Registers: 290 Combinational ALUTs: 75 DSP block 18bit: 8 Fmax: 329 MHz

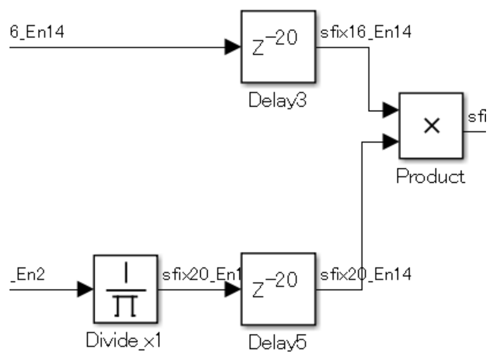
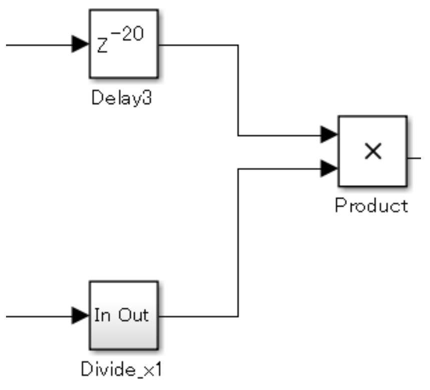
2.6.9 Model the delay of blocks that will be auto-pipelined (Divide, Sqrt, Trigonometric Function, Cascade Add/Product, Viterbi Decoder)

Delay differences have a large effect on system operation, especially for algorithms that iterate through feedback loops. For the Divide, Sqrt, and Trigonometric Function blocks, the correct amount of registers will be automatically inserted during code generation. Thus it is best practice to estimate these delays when creating the model in Simulink so that the model has the proper output latency when used in the system.

When using these blocks, insert the delay immediately after the block so that HDL Coder will incorporate it into the generated module for that block. Refer to the product documentation for predicting the amount of delay to insert:

- Divide (when **HDL Block Properties > Architecture** is `RecipNewton` or `RecipNewtonSingleRate`)
- Sqrt (when **HDL Block Properties > Architecture** is `RecipNewton` or `RecipNewtonSingleRate`)
- Trigonometric Function (when **Block Parameters (Trigonometry) > Approximation method** is `CORDIC`)

Example: BS0014_Latency.slx

Simulink model which uses a Divide block	The generated verification model
 6_En14 z^{-20} sfix16_En14 Delay3 sfi Product _En2 sfix20_En1 z^{-20} sfix20_En14 Divide_x1 Delay5	 The pre-inserted delay of 20 is incorporated inside the generated divider.

Inserted delay amounts based on each block's setting are shown in the following table.

Block name	HDL block property/Architecture	Number of additional latency
Divide	`RecipNewtonSingleRate`	4*Iteration+8 (an input is a signed) 4*Iteration+6 (an input is a unsigned)
	`RecipNewton`	Iteration+5 (an input is a signed) Iteration+3 (an input is a unsigned)
Sqrt	`SqrtNewtonSingleRate`	Iteration+3
	`SqrtNewton`	4*Iteration+6
ReciprocalSqrt	`RecipSqrtNewtonSingleRate`	17
	`RecipSqrtNewton`	5
Trigonometric Function	`SinCosCordic, Trigonometric`	Number of occurrence+1
Add, Product	`Cascade`	1
Communications System toolbox / Error Detection and Correction / Convolution / Viterbi Decoder	In the case of Register-based Traceback `TracebackStagesPerPipeline`	Tracebackdepth/TracebackStagesPerPipeline+12 (when Register-based Traceback is chosen) Tracebackdepth*3+14 (when RAM-based Traceback is chosen)

2.6.10 Use Divide blocks in reciprocal mode with a RecipNewton or RecipNewtonSingleRate architecture for more optimal HDL

When generating HDL from the Divide block, it is best to use it in reciprocal mode with a Product block, and to specify the architecture for more accurate system simulation

For best results out of your synthesis tool, use the Divide block in reciprocal mode and connect it to a Product block. The divide operator in HDL (/) will require manual intervention downstream in synthesis in order to specify an implementation architecture and pipelining parameters. It is best to specify this during algorithm design so that the number of iterations and latency is modeled during system simulation.

Set the following parameters on the Divide block:

Parameter	Setting
Block Parameters > **Main > Number of inputs**	`/`
Block Parameters > **Signal attributes > Integer rounding mode**	`Zero`
Block Parameters > **Signal attributes > Saturate on integer overflow**	<Selected>
HDL Block Properties > **Architecture**	`ReciprocalRsqrtBasedNewton` for smaller area `ReciprocalRsqrtBasedNewtonSingleRate` for higher frequency

For RecipNewton and RecipNewtonSingleRate architectures, the amount of delay (number of registers) that will be added automatically according to the following rules (where “n” is the number of iterations):

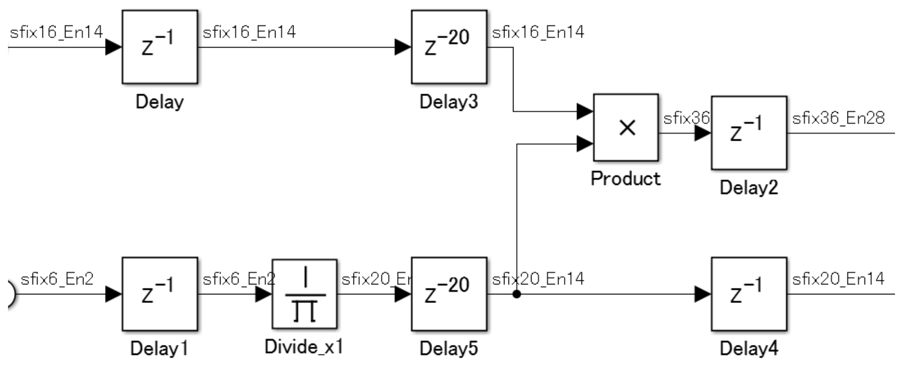
`ReciprocalRsqrtBasedNewtonSingleRate`	Signed input: Latency = 4*n + 8 Un-signed input: Latency = 4*n + 6
`ReciprocalRsqrtBasedNewton`	Signed input: Latency = n + 5 Un-signed input: Latency = n + 3

Example: *BS0003_Divide.slx*

Modeling a signed `RecipNewtonSingleRate` reciprocal function with 3 iterations, which results in a latency of 20:

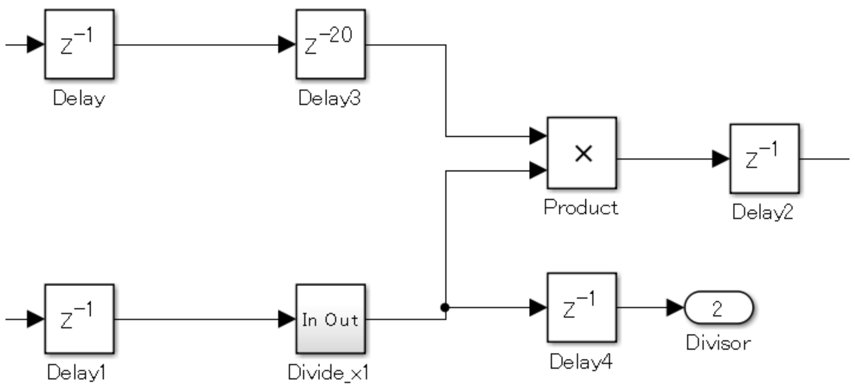
Original model

The delay of 20 is inserted explicitly into the model so that its output latency is correctly modeled for system simulation.



Generated model

The 20 pipeline stages are included inside in the HDL code for the Divide block so they are not visible at this level. During downstream synthesis the registers will be distributed throughout the Divide block.



2.6.11 Consider the additional latency impact of different implementation architectures for the Sqrt and ReciprocalSqrt blocks

Example: BS039_sqrt.slx

Note that the inputs to the Sqrt and ReciprocalSqrt blocks need to be unsigned fixed point integer data types. You can choose different implementation architectures under **HDL Block Properties…>Architecture**. The following table compares the additional latency inserted for the architecture choices:

Block	Architecture	Other settings	Implementation	Additional latency (number of samples)
Sqrt	`SqrtFunction`	None	Multiply/Add [3]	0
	`SqrtBitset`	`UseMultiplier` = on/off [1]		0
	`SqrtNewton`	`Iterations = 2` [2]	Newton method [4]	Iterations+3 = 5
	`SqrtNewton SingleRate`	`Iterations = 2` [2]	Newton method [5]	4*Iterations+6 = 14
Reciprocal Sqrt	`RecipSqrtNewton`	None	Newton method [4]	5
	`RecipSqrtNewton SingleRate`	None	Newton method [5]	17

Footnotes:

1. The `UseMultiplier` setting does not affect the implementation.
2. Setting `Iterations` to more than 2 does not improve accuracy, so set to 2.
3. For `UseMultiplier=on`, it uses a multiply/add for both `SqrtFunction` and `SqrtBitset`. `UseMultiplier=off` requires extra processing for `SqrtBitset`.
4. When `SqrtNewton` is chosen, it uses the Newton approximate calculation method, which uses 3x over-clocking
5. When `SqrtNewtonSingleRate` is chosen, it uses the Newton approximate calculation method in a single clock cycle

2.6.12 Tradeoffs for Sin/Cos calculation using Trigonometric Function, Lookup Table, Sine/Cosine, and NCO HDL Optimized block

There are four methods of performing Sin/Cos calculation. The logic area and pipeline depth will differ depending on the frequency.

	Trigonometric Function block	Lookup Table block	Sine / Cosine block (Lookup Tables library)	DSP System Toolbox>Signal Operations>NCO HDL Optimized
Feature	Uses CORDIC approximation. A	The memory for one period is consumed.	The memory for 1/4 period is consumed.	Although it is a block for generating a signal,

	pipeline delay is added. This is recommended if delay is permitted.	No additional delay is generated.	No additional delay is generated.	it can also be used for Sin/Cos calculation.
Calculation technique	sin, cos, cos+jsin, sincos	Dependent on formula entered in **Table data** field	sin, cos, exp(j), sincos	sin, cos, exp(j), sincos
Limitations	**Approximation method** set to `CORDIC` [1]. Delay of the number of occurrences+1 is added.	Same as general HDL restrictions for a Lookup Table block.	Either `Speed` or `Precision` are valid settings for **Internal rule priority for lookup table**	Delay of six samples is added. Set **Phase increment** to `0` and **Phase offset source** to `Input port`.
Latency (delay)	Number-of-occurrences +1 sample	0	0	Six samples
Post-synthesis Frequency (Cyclone IV target)	About 200 MHz for 12 bit outputs	At least 250 MHz for 12 bit outputs	About 60 MHz for 12 bit outputs.	About 260 MHz for 12 bit outputs
Post-synthesis Area	For large bit widths, it is relatively small compared to other techniques	Inefficient because all table data must be defined for one period	Efficient because table data for 1/4 period is defined and chosen with a switch.	Efficient because table data for 1/4 period is used. [2]

Footnotes:

1. CORDIC is an algorithm of an approximate calculation, and because it calculates by repetitive operations referencing small Lookup Table, adding/subtracting, and shifting, operation is possible at few circuit resources.
2. By selecting the block parameter **Enable look up table compression method**, the Lookup Table data for 1/4 cycle is further compressible. Refer to the documentation for this block for details.

Example: BS0035_SinCos.slx

2.6.13 Use only conj, hermitian, or transpose in a Math Function block

Only the following `Function` parameters are supported for HDL code generation from a Math Function block:

- `conj` : calculate a complex conjugate
- `hermitian` : because two-dimensional vectors are not supported by HDL code generation, use this function for row vector <=> column vector conversion
- `transpose` : calculate a complex conjugate transpose

Note: `reciprocal` also supports HDL code generation, however it is best practice to use the Divide block for a reciprocal function since it is easier to set up code generation parameters.

Example: BS050_mathFunc.slx

2.6.14 HDL code generation compatible Math Operations for complex number computation

The following table summarizes both simulation as well as HDL code generation support for blocks in the Math Operations library

Block name	Simulation with a complex input	HDL code generation with a complex input
Abs	Unsupported	Unsupported
Add	✓	✓
Assignment	✓	Unsupported
Bias	✓	Unsupported

Complex to Real-Image	✓	✓
Decrement Real World	✓	✓
Decrement Stored Integer	✓	✓
Divide	Unsupported	Unsupported
Dot Product	✓	✓
Gain	✓	✓ (real number coefficient and complex modulus)
Increment Real World	✓	✓
Increment Stored Integer	✓	✓
Magnitude-Angle to Complex	Only Real for input	Only Real for input
Math Function - conj [1]	✓	✓
Math Function - transpose [1]	✓	✓
Math Function - hermitian [1]	✓	✓
Matrix Concatenate	✓	Vector only
MinMax	Unsupported	Unsupported
Product	✓	✓
Product of Elements	Only 2 inputs supported	Only 2 inputs supported
Real-Imag to Complex	Only Real for input	Only Real for input
Reciprocal	Unsupported	Unsupported
Reciprocal Sqrt	Unsupported	Unsupported
Reshape	✓	Scalar and vector only
Sign	Unsupported	Unsupported
Sqrt	Unsupported	Unsupported
Subtract	✓	✓
Sum	✓	✓
Sum of Elements	✓	✓
Trigonometric Function	Unsupported	Unsupported
Unary Minus	✓	Signed data only
Vector Concatenate	✓	Supported

Footnotes:

1. In **HDL Block Properties…** use the default value `Math` for `Architecture`.

2.7 Ports and subsystems

2.7.1 Block settings for Triggered Subsystems/Enabled Subsystems

A triggered subsystem is a subsystem that receives a control signal via a Trigger block. The triggered subsystem executes for one cycle each time a trigger event occur. In order to generate HDL code from a triggered subsystem, the block must be set with the preferences as detailed in Restrictions for HDL code generation from a triggered subsystem.

An enabled subsystem is a subsystem that receives a control signal via an Enable block. The enabled subsystem executes at each simulation step where the control signal has a positive value. In order to generate HDL code from an enabled subsystem, the block must be set with the preferences as detailed in Restrictions for HDL code generation from an enabled subsystem.

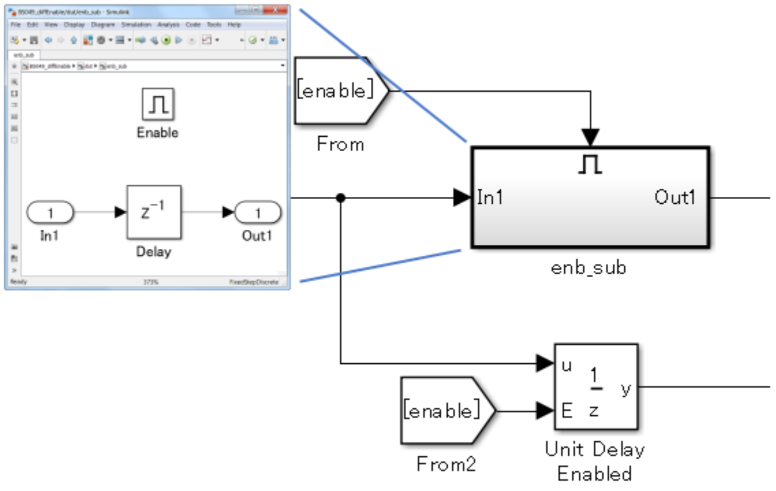
Example: BS028_triggeredEnabled.slx

2.7.2 Proper usage of a Unit Delay Enabled block versus an enabled subsystem with a Delay block

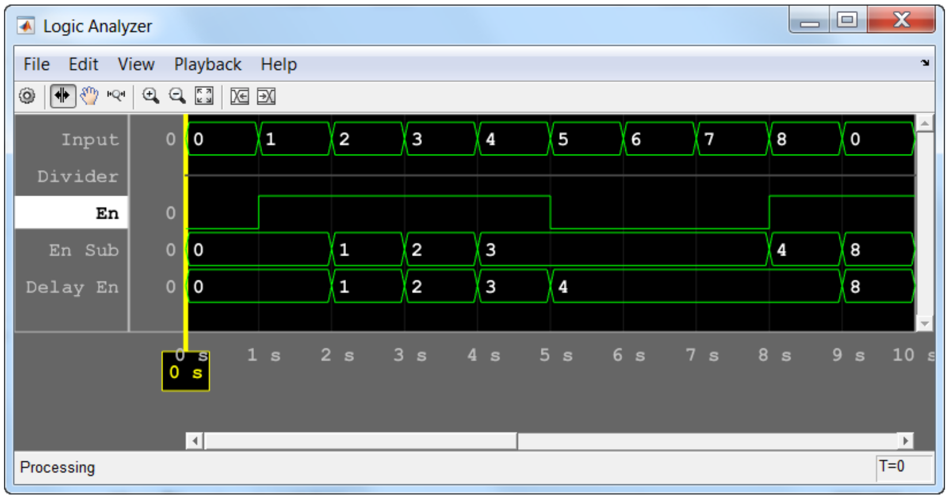
In the Simulink/Discrete library there is a Unit Delay Enabled block, which has the same behavior as a Unit Delay block, with the addition of an enable input. When the enable input is active (high), the signal input will be propagated to the output after a 1 sample delay.

Similar behavior can be modeled by using a Unit Delay block inside an enabled subsystem. This could be useful in modeling multi-sample delays. However the behavior between these two modeling methods differs slightly. In the following example, both are modeled with the same input signal and enable:

Example: BS049_diffEnable.slx



The resulting waveforms illustrate the difference in behavior with both modeling a 1 sample delay. “En Sub” is the output from the enabled subsystem and “Delay En” is the output from the Unit Delay Enabled block:



Whereas the enabled subsystem shuts down when enable becomes inactive (low), the Unit Delay Enabled block stops sampling the input when enable becomes inactive, but it still outputs the delayed input that was in its queue. Therefore take care to use the modeling semantic that is best suited for your application.

2.8 Signal attributes

2.8.1 Rate conversion blocks and usage

There are multiple methods for modeling rate transitions, which can impact your design’s final timing and resource requirements in hardware. It is important to understand the impact of different techniques, including when to add a register for synchronous designs.

Raising the sample rate

There are a few blocks that can be used to raise the sample rate, depending on the needs of your design:

Block	Generates bypass register?	Generates zero-padding?	Parameters

Repeat	No	No	
Rate Transition	Yes	No	**Ensure data integrity during data transfer** = `true` **Ensure deterministic data transfer (maximum delay)** = `true`
Upsample	Yes	Yes	**Input processing** = `Elements as channels (sample based)` **Rate options** = `Allow multirate processing`

Rate Transition and Upsample each generate a register, and Upsample also generates logic to insert zero-padding logic, so be aware of these hardware impacts when using them.

A Delay block should be added to the design after each if the input and output clocks are not synchronous to each other. This will add an extra 1-sample delay at the output rate to ensure synchronization of the signal.

Lowering the sample rate

Either a Rate Transition or a Downsample block can be used to lower the sample rate. Since they both generate the same HDL code, it is easier to use a Rate Transition block since its parameters for HDL code generation are more easily set.

- Rate Transition block

  Set **Ensure data integrity during data transfer** and **Ensure deterministic data transfer (maximum delay)** to `true`. The **Initial condition** parameter is not used when raising the sample rate. This block does not insert a register, so when the input and output clocks are not synchronous to each other you will need to insert a Delay block for synchronization.

Example: BS029_upDownSample.slx

2.9 Signal routing

2.9.1 Choosing the right block for extracting a portion of a vector signal

The following blocks are available for extracting a scalar or a portion of a vector signal from a vector signal:

- HDL Coder/Signal Routing/Selector
- HDL Coder /Signal Routing/Index Vector
- HDL Coder /Signal Routing/Multiport Switch
- DSP System Toolbox/Signal Management/Indexing/Multiport Selector
- DSP System Toolbox/Signal Management/Indexing/Variable Selector

Select blocks appropriately considering whether the extraction ranges are fixed or variable. The following table lists supported blocks and recommended blocks according to types of extraction ranges.

Extraction ranges	Available blocks (**bold**=recommended)	Recommended	Parameters
Fixed	**Selector** [1]	Yes	**Index mode** = `Zero-based`
	Multiport Selector [2]		
Variable	**Multiport Switch**	Yes for multiple scalar inputs	**Number of data ports** = *<# of scalar inputs>* **Data port order** = `Zero-based contiguous` for zero-based indexing
	Index Vector	Yes for vector input	**Number of data ports** = `1`
	Selector [3]		
	Variable Selector [3]		

Footnotes

1. Zero-based indexing is supported. A vector signal is output from one port when multiple elements are selected. Although the generated HDL for a Selector and a Multiport Selector are the same with zero-based indexing, the Selector block offers better traceability between the model and the generated code.
2. The generated HDL code uses zero-based indexing even though MATLAB code supports only one-based indexing. In the use case of extracting multiple elements, these can be output from multiple separate output ports.
3. The use of Variable Selector is not recommended, because it needs as input built-in data types such as uint8 for the index port (Idx), causing bit redundancy.

Example: BS008_selector.slx

2.9.2 Block parameter setting for the Multiport Switch Block

As described in section 2.9.1 Choosing the right block for extracting a portion of a vector signal, there are multiple options for switch blocks depending on your needs. The control input signal for these blocks is typically a numeric data type. However the Multiport Switch and Index Vector blocks also support enumerated data types for control input, which helps with debugging. For each approach, set the following **Function Block Parameters**:

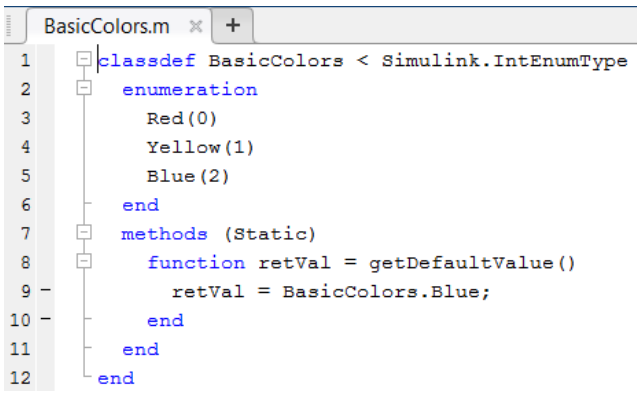
When the control signal input is a numeric type:

Example: BS032_switch.slx

- **Data port order** = `Zero-based contiguous`
- **Data port for default case** = `Last data port`

When the control input signal is an enumerated type, you will need to define a MATLAB enumeration type class:

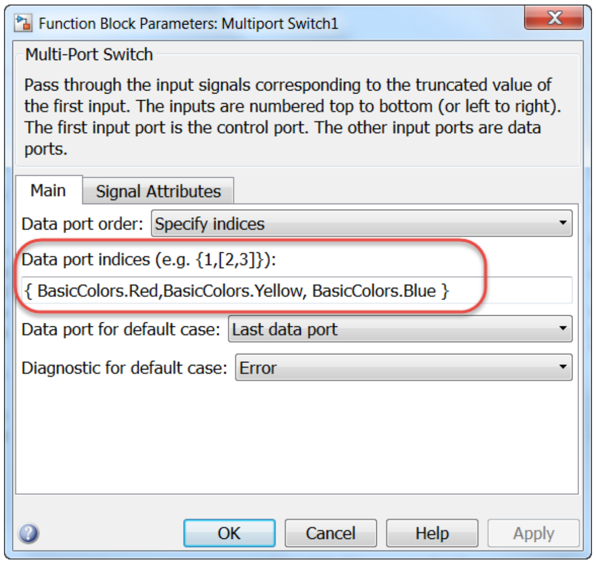
Example: BasicColors.m



```
classdef BasicColors < Simulink.IntEnumType
  enumeration
    Red(0)
    Yellow(1)
    Blue(2)
  end
  methods (Static)
    function retVal = getDefaultValue()
      retVal = BasicColors.Blue;
    end
  end
end
```

Then you can set Data port indices as follows:

Example: BS031_enum.slx


Function Block Parameters: Multiport Switch1

Multi-Port Switch

Pass through the input signals corresponding to the truncated value of the first input. The inputs are numbered top to bottom (or left to right). The first input port is the control port. The other input ports are data ports.

Main | Signal Attributes

Data port order: Specify indices

Data port indices (e.g. {1,[2,3]}):

{ BasicColors.Red,BasicColors.Yellow, BasicColors.Blue }

Data port for default case: Last data port

Diagnostic for default case: Error

OK | Cancel | Help | Apply

2.9.3 Add 1 to index signals when describing a selector circuit in a MATLAB Function block

In addition to the Multiport Switch and Index Vector blocks, selector circuit can be described using a MATLAB Function block. When doing so, add 1 to the value of the index since MATLAB uses 1-based indexing while Simulink uses 0-based indexing for HDL code generation. Additionally, add 1 to an index variable before selecting the index in MATLAB in order to prevent additional logic from being generated.

The following generated Verilog code snippets illustrates the difference:

Example: BS041_MLFSelector.slx

Correct	Incorrect

`function Do = mfb_select(Di,Idx)`
`%#codegen`
`index = int32(Idx)+1; % OK`
`Do = Di(index);`
	`function Do = fcn(Di,Idx)`
	`%#codegen`
	`Do = Di(uint32(Idx) + uint32(1)); % This statement will not be optimized.`

`assign index = Idx;`
`assign Do_rsvd_1 = Di[index];`
	`assign add_temp = Delay1_out1 + 1;`
	`assign Do_rsvd_1 = Delay_out1[$signed({1'b0, add_temp}) - 1];`

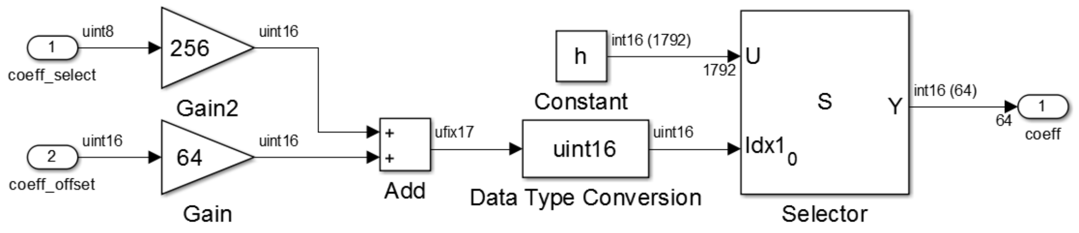
When using this technique, in MATLAB under **Edit Data>Ports and Data Manager** make sure that **Saturate on integer overflow** is not selected.

2.9.4 Use a MATLAB Function block to select indices when extracting portions of a very large constant vector

When extracting a portion of a very large constant vector, using a MATLAB Function block with the indexing technique described in 2.9.3 Add 1 to index signals when describing a selector circuit in a MATLAB Function block will generate more optimal hardware than using a Selector block. The following example illustrates:

Example OP004_largeVector.slx

- Using a Selector block:

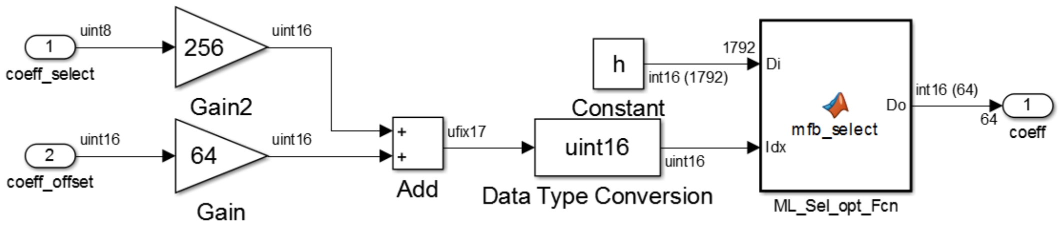


This approach will generate HDL code with large assign statements that represent 1792-to-1 multiplexors, for each of the 64 elements of the output vector, similar to the following:

```
Selector_out1_0 <= Constant_out1_0 WHEN Data_Type_Conversion_out1 = to_unsigned(16#0000#, 16) ELSE
Constant_out1_1 WHEN Data_Type_Conversion_out1 = to_unsigned(16#0001#, 16) ELSE
Constant_out1_2 WHEN Data_Type_Conversion_out1 = to_unsigned(16#0002#, 16) ELSE
Constant_out1_3 WHEN Data_Type_Conversion_out1 = to_unsigned(16#0003#, 16) ELSE
.
.
.
Constant_out1_1790 WHEN Data_Type_Conversion_out1 = to_unsigned(16#06BF#, 16) ELSE
Constant_out1_1791;
.
.
.
Selector_out1_63 <= Constant_out1_63 WHEN Data_Type_Conversion_out1 = to_unsigned(16#0000#, 16) ELSE
Constant_out1_64 WHEN Data_Type_Conversion_out1 = to_unsigned(16#0001#, 16) ELSE
```

The optimization of these multiplexors will depend on your RTL synthesis tool, but such an inefficient input to synthesis can be avoided by using a MATLAB Function block.

- Using a MATLAB Function block:



Example code MATLAB Function block code

`function Do = mfb_select(Di,Idx)`
`%#codegen`
`index = int32(Idx)+1;`
`Do = Di(index);`

This approach will generate HDL code similar to the following:

```
-- <S4>/ML_Sel_opt_Fcn
--
--index is assigned the output from the adder
index <= signed(resize(Data_Type_Conversion_out1, 32));

-- index is used as an offset in a generate statement to select the output
Do_gen: FOR t_0 IN 0 TO 63 GENERATE
  Do(t_0) <= Constant_out1(to_integer(to_signed(t_0, 32) + index));
END GENERATE Do_gen;

outputgen: FOR k IN 0 TO 63 GENERATE
  coeff(k) <= std_logic_vector(Do(k));
END GENERATE;
```

2.9.5 Writing to individual elements of a vector signal using the Assignment block

The Assignment block enables you to write to selected elements of an n-D matrix. In order to target this block to hardware, use the following settings:

- **Number of output dimensions**: 1
- **Index mode**: `Zero-based`
- **Index Option**: `Index vector (port)`

- **Initialize output (Y)**: `Initialize using input port (Y0)`

This block is useful for writing to register banks or RAM, as illustrated in *Example BS055_assignment.slx*.

2.9.6 Proper usage of Goto/From blocks

There are a few guidelines that must be followed when using Goto/From blocks in subsystems from which you plan to generate HDL:

When using a Goto/From block for an HDL generation target subsystem, it is following restriction within the limits, and use it.

1. Don't use a Goto/From combination that crosses the boundary of the HDL generation target subsystem. To interface outside of the HDL target subsystem, use an Inport or Output.
2. When using a Goto/From combination, try to keep its scope local to a given hierarchy, and set its **Tag visibility** to `local`. If you have to use a Goto/From across hierarchies within a subsystem, set its **Tag visibility** to `global`.
3. Even though there is a Goto Tag Visibility block available, this cannot be used in an HDL generation target subsystem. For this reason, don't set a Goto block's **Tag visibility** to `scoped`.

For proper usage examples, see:

Example: BS019_gotoFrom.slx

2.9.7 Ascending bit ordering for 1-D arrays may cause warnings from HDL rule checkers

Because the default ordering of arrays in MATLAB is ascending (LSB→MSB), HDL will also be generated using ascending order (VHDL: to, Verilog: [LSB:MSB]) for one-dimensional arrays of 1-bit values. This will typically generate a warning from HDL code rule checkers. This is illustrated in the following example:

Example: BS054_downto.slx

1. Delay block with a delay value greater than 1 will generate a buffer in HDL with ascending bit ordering:

```
Delay_process : PROCESS (clk, reset)
   BEGIN
     IF reset = '1' THEN
       Delay_reg <= (OTHERS => '0');
     ELSIF clk'EVENT AND clk = '1' THEN
       IF enb = '1' THEN
         Delay_reg(0) <= In1;
         Delay_reg(1 TO 4) <= Delay_reg(0 TO 3);
       END IF;
     END IF;
   END PROCESS Delay_process;
```

2. Mux block with a vector signal of 1-bit data:

```
  SIGNAL Mux_out1   : std_logic_vector(0 TO 3);  -- ufix1 [4]
```

3. Constant:

```
SIGNAL Constant_out1    : std_logic_vector(0 TO 3);  -- boolean [4]
```

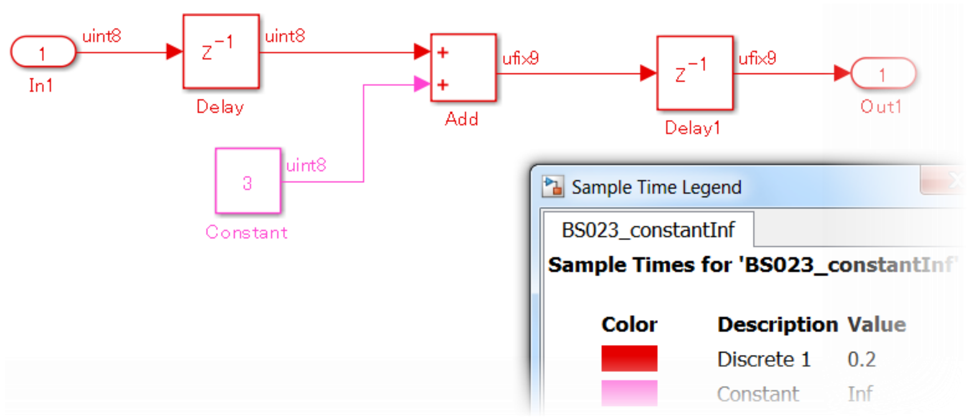
2.10 Source blocks

2.10.1 Do not use a sample time of `inf` for a Constant block

The default sample time for a Constant block is `inf`. However connecting a Constant block with sample time `inf` to an input port of an HDL target subsystem will prevent optimizations such as pipelining, retiming, sharing, and streaming. This is because these optimizations rely on an understanding of the clock rate.

When using the Constant block, set the sample time to -1 to inherit.

To quickly find all Constant blocks with a sample time of inf, turn on Display>Sample Time>Colors. After running simulation, you will see these blocks highlighted as follows:



Sample Times for 'BS023_constantInf'

Color	Description	Value
🟥	Discrete 1	0.2
🟪	Constant	Inf

Example: BS023_constantInf.slx, constSampleTimeSet.m

The constSampleTimeSet.m script will iterate through these blocks and set the sample times properly for HDL code generation and optimization.

2.11 MATLAB Function blocks

2.11.1 Proper usage of dsp.Delay as a register

In MATLAB code, there are two ways to describe registers:

- Defining a persistent variable
- Using the dsp.Delay System object (in DSP System Toolbox)

Using the dsp.Delay System object results in better readability – it is easier to see where the registers will be. However using System objects in MATLAB code may inhibit certain optimizations. For example there can be code generation limitations related to partitioning or recognition of System objects. Additionally, dsp.Delay cannot be used in cases where its previous value needs to be accessed, such as a FIR filter or a counter. For such cases, a persistent variable can be used to model a register. The following MATLAB code illustrates:

Example: BS012_MLFSequential.slx

Persistent variable	dsp.Delay System object

```
function [yp0,yp1] = fcn(u0)
%#codegen
persistent FFp0 FFp1
if isempty(FFp0)    % initialize
    FFp0 = fi(0, 1, 9, -2);
    FFp1 = fi([0 0 0 0] , 1, 9, -2);
end

% Output
yp0 = FFp0;
yp1 = FFp1(4);

% Update FF after output
FFp0 = u0*fi(4, 0, 1, -2);   %
FFp1(:) = [u0*fi(4, 0, 1, -2),
FFp1(1:3)];
% new data and shift register value
```
	```
	function [yd0,yd1] = fcn(u0)
	%#codegen
	persistent FFd0 FFd1
	if isempty(FFd0)    % initialize
	    FFd0 = dsp.Delay(1);     % 1sample
	delay
	    FFd1 = dsp.Delay(4);     % 4sample
	delay
	end

	tmp0 = u0*fi(4, 0, 1, -2);
	% Output
	yd0 = step(FFd0, tmp0);
	yd1 = step(FFd1, tmp0);
	```

As shown in the following, when using vector and matrix data with dsp.Delay, `Units` should be set to `Frames`, and `FrameBasedProcessing` set to `false`.

```
FFin0 = dsp.Delay(1, 'Units', 'Frames', 'FrameBasedProcessing', false);
```

2.11.2 Update persistent variables at the end of a MATLAB function

In order to map to a register, a persistent variable cannot be updated before its value is read or used by the function. The following table illustrates:

Correct	Incorrect
`function FF_out0  = fcn(FF_in)`	`function FF_out0  = fcn(FF_in)`
`%#codegen`	`%#codegen`
`persistent FF0`	`persistent FF0`
`if isempty(FF0)`	`if isempty(FF0)`
`    FF0 = zeros(1, 'like', FF_in);`	`    FF0 = zeros(1, 'like', FF_in);`
`end`	`end`
`% Output FF0`	`% Incorrect Order for FF update`
`FF_out0 = FF0;`	`FF0 = FF_in`
`% Write FF update at the end of the code`	`% Output FF0`
`FF0 = FF_in`	`FF_out0 = FF0;`
	`% FF_out0 is NOT delayed`

Example: BS016_MLFPersistentOrder.slx

2.11.3 Explicitly define data types for constants used in expressions

When performing an operation in a MATLAB Function block that will cast the data type of the result, be sure to define the datatype for constant operands using the fi object. Otherwise the generated HDL code could contain non-synthesizable constructs such as `$rtoi` or `real()`.

Example: BS024_MLFConstOpp.slx

Incorrect: The constant's datatype is not explicitly defined.

```
out0 = in0 / 4;
out1 = in0 * 4;
out2 = fi(in0 / 4, 1, 11, 1);
out3 = fi(in0 * 4, 1, 11, 1);
```

Correct: Define the constant's data type immediately where it's used in an operation.

```
out0 = in0 / fi(4, 0, 3, 0);
out1 = in0 * fi(4, 0, 3, 0);
out2 = fi(in0 / fi(4, 0, 3, 0), 1, 11, 1);
out3 = fi(in0 * fi(4, 0, 3, 0), 1, 11, 1);
```

Correct: Define a constant as a variable with an explicit data type and use the variable in an operation.

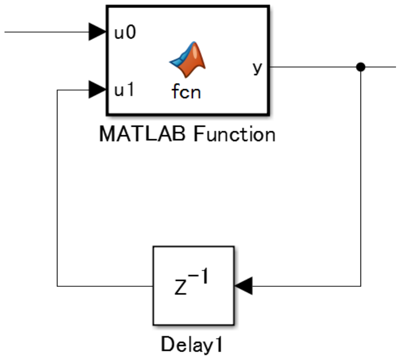
```
ConstDiv = fi(4, 0, 3, 0);
ConstProd = fi(3, 0, 2, 0);
out0 = in0 / ConstDiv;
out1 = in0 * ConstProd;
out2 = fi(in0 / ConstDiv, 1, 11, 1);
out3 = fi(in0 * ConstProd, 1, 11, 1);
```

2.11.4 Use Delay blocks to break feedback loops in MATLAB Function blocks

If your MATLAB function block has a feedback loop, break the loop with a Simulink Delay block instead of using a persistent variable in MATLAB. Using only a persistent variable in a feedback loop will result in an algebraic loop violation during HDL code generation from Simulink.

The following example illustrates how to properly break a feedback loop with a Delay block:

Example: AD017_MLFFeedback.slx



2.11.5 Do not use logical operators in conditional statements when initializing persistent variables

Typically a conditional statement such as “`if isempty(var)`” is used to check whether a persistent variable needs to be initialized. When multiple variables are checked, try to avoid using logical operators since this will generate HDL code with intermediate variables, making it less readable. Consider the following example:

Example: BS0015_MLF_persistent_vars.slx

Correct	Incorrect

persistent FF0 FF1
if isempty(FF0)
    FF0 = fi(0, 0, 8, 0);
end
if isempty(FF1)
    FF1 = fi(0, 0, 8, 0);
end
	persistent FF0 FF1
	if isempty(FF0)&&isempty(FF1)
	    FF0 = fi(0, 0, 8, 0);
	    FF1 = fi(0, 0, 8, 0);
	end

always @(posedge clk or posedge reset)
     begin : MLF_good_1_process
       if (reset == 1'b1) begin
         FF0 <= 8'b00000000;
         FF1 <= 8'b00000000;
       end
       else begin
         if (enb) begin
           FF0 <= FF0_next;
           FF1 <= FF1_next;
         end
       end
     end
 	
   always @(FF_in, FF0, FF1) begin
     FF0_next = FF0;
     FF1_next = FF1;
     FF_out0_1 = FF0;
     FF_out1_1 = FF1;
 	
     if (FF_in > 17'sb00000000000000000)
begin
       FF0_next = FF0 + 1;
     end
     if (FF_in < 17'sb00000000000000000)
begin
       FF1_next = FF1 + 1;
     end
	always @(posedge clk or posedge reset)
	     begin : MLF_bad_1_process
	       if (reset == 1'b1) begin
	         FF0_not_empty <= 1'b0;
	         FF1_not_empty <= 1'b0;
	       end
	       else begin
	         if (enb) begin
	           FF0 <= FF0_next;
	           FF0_not_empty <= FF0_not_empty_next;
	           FF1 <= FF1_next;
	           FF1_not_empty <= FF1_not_empty_next;
	         end
	       end
	     end
	
	   always @(FF_in, FF0, FF0_not_empty, FF1, FF1_not_empty)
	   begin
	     FF0_temp_1 = FF0;
	     FF1_temp_1 = FF1;
	     FF0_not_empty_next = FF0_not_empty;
	     FF1_not_empty_next = FF1_not_empty;
	
	     if ((! FF0_not_empty) && (! FF1_not_empty)) begin
	       FF0_temp_1 = 8'b00000000;
	       FF0_not_empty_next = 1'b1;
	       FF1_temp_1 = 8'b00000000;
	       FF1_not_empty_next = 1'b1;
	     end

```
  end
```

```
    FF_out0_1 = FF0_temp_1;
    FF_out1_1 = FF1_temp_1;
    if (FF_in > 17'sb00000000000000000) begin
      FF0_temp_1 = FF0_temp_1 + 1;
    end
    if (FF_in < 17'sb00000000000000000) begin
      FF1_temp_1 = FF1_temp_1 + 1;
    end
    FF0_next = FF0_temp_1;
    FF1_next = FF1_temp_1;
  end
```

2.11.6 Use X(:)=X+1; when input and output data types are the same in MATLAB code expressions

When the datatype of the output variable of an expression is the same as an input, you can re-use the variable to avoid having to define a temporary variable and its associated data type. Take care to use subscripted assignment to avoid bit growth. For example:

Example: BS011_MLFSameData.slx

```
>> X(:) = X+1;
>> X(:) = X+Y;
>> Y(:) = Y*5;
>> Y(:) = X*Y;
```

In contrast, the following would result in non-synthesizable HDL because the new variable would be cast by default to a `double` data type due to the use of the constant:

```
>> tmp = Y*5;
```

So in this case you would have to explicitly define the data type for the new variable:

```
>> tmp = Y*fi(5, 0, 3, -1);
>> tmp = fi(Y*5, 1, 16, 8);
```

2.11.7 Avoid unintended latch inference by performing arithmetic operations outside of if/else branches

Example: BS040_MLFLatch.slx

1. When the branches of an if/else statement perform different arithmetic operations, HDL code generation creates intermediate variables for the different operations. Since these intermediate variables are only assigned in the branches of the if/else statement where they are used, this will result in latch inference by RTL synthesis.

MATLAB code that **will not** cause latch inference

```
function cOut  = fcn(a_in, c_in)

ntype = numerictype(1,32,0);
c_o_1 = fi(c_in, ntype);
c_o_2 = fi(c_in*2, ntype);
c_o_4 = fi(c_in*4, ntype);

if a_in == 0
    cOut = c_o_2;
elseif a_in == 1
    cOut = c_o_4;
else % Default
    cOut = c_o_1;
end
```

MATLAB code that **will** cause latch inference

```
function cOut  = fcn(a_in, c_in)

ntype = numerictype(1,32,0);
c_o_1 = fi(c_in, ntype);
c_o_2 = fi(c_in*2, ntype);
c_o_4 = fi(c_in*4, ntype);

if a_in == 0
    cOut = fi(c_in*2, ntype);
elseif a_in == 1
    cOut = fi(c_in*4, ntype);
else % Default
    cOut = fi(c_in, ntype);
end
```

Generated VHDL `cast` and `cast_0` written to outside of the if/else statement	Generated VHDL `cast` and `cast_0` written to inside only one branch of the if/else statement

Generated VHDL — `cast` and `cast_0` written to outside of the if/else statement

```
IfElse_OK_1_output : PROCESS (a_in_unsigned,
c_in_signed)
     VARIABLE c_o_2 : signed(5 DOWNTO 0);
     VARIABLE cast : signed(11 DOWNTO 0);
     VARIABLE cast_0 : signed(11 DOWNTO 0);
   BEGIN
     cast := resize(c_in_signed & '0' & '0' & '0' &
'0', 12);
    IF ((cast(11) = '0') AND (cast(10 DOWNTO 8) /=
"000")) OR ((cast(11) = '0') AND (cast(8 DOWNTO 3)
= "011111")) THEN
       c_o_2 := "011111";
    ELSIF (cast(11) = '1') AND (cast(10 DOWNTO 8)
/= "111") THEN
       c_o_2 := "100000";
    ELSE
       c_o_2 := cast(8 DOWNTO 3) + ('0' & cast(2));
     END IF;
     cast_0 := resize(c_in_signed & '0' & '0' & '0'
& '0', 12);
    IF ((cast_0(11) = '0') AND (cast_0(10 DOWNTO 7)
/= "0000")) OR ((cast_0(11) = '0') AND (cast_0(7
DOWNTO 2) = "011111")) THEN
       cOut_tmp <= "011111";
    ELSIF (cast_0(11) = '1') AND (cast_0(10 DOWNTO
7) /= "1111") THEN
       cOut_tmp <= "100000";
    ELSE
       cOut_tmp <= cast_0(7 DOWNTO 2) + ('0' &
cast_0(1));
     END IF;
    IF a_in_unsigned = 0 THEN
       cOut_tmp <= c_o_2;
    ELSIF a_in_unsigned = 1 THEN
    ELSE
       cOut_tmp <= c_in_signed;
     END IF;
   END PROCESS IfElse_OK_1_output;
```

Generated VHDL — `cast` and `cast_0` written to inside only one branch of the if/else statement

```
IfElse_Bad_1_output : PROCESS (a_in_unsigned,
c_in_signed
     VARIABLE cast : signed(11 DOWNTO 0);
     VARIABLE cast_0 : signed(11 DOWNTO 0);
   BEGIN
    IF a_in_unsigned = 0 THEN
       cast := resize(c_in_signed & '0' & '0' &
'0' & '0', 12);
      IF ((cast(11) = '0') AND (cast(10 DOWNTO 8)
/= "000")) OR ((cast(11) = '0') AND (cast(8 DOWNTO
3) = "011111")) THEN
         cOut_tmp <= "011111";
      ELSIF (cast(11) = '1') AND (cast(10 DOWNTO
8) /= "111") THEN
         cOut_tmp <= "100000";
      ELSE
         cOut_tmp <= cast(8 DOWNTO 3) + ('0' &
cast(2));
       END IF;
    ELSIF a_in_unsigned = 1 THEN
       cast_0 := resize(c_in_signed & '0' & '0' &
'0' & '0', 12);
      IF ((cast_0(11) = '0') AND (cast_0(10 DOWNTO
7) /= "0000")) OR ((cast_0(11) = '0') AND
(cast_0(7 DOWNTO 2) = "011111")) THEN
         cOut_tmp <= "011111";
      ELSIF (cast_0(11) = '1') AND (cast_0(10
DOWNTO 7) /= "1111") THEN
         cOut_tmp <= "100000";
      ELSE
         cOut_tmp <= cast_0(7 DOWNTO 2) + ('0' &
cast_0(1));
       END IF;
    ELSE
       cOut_tmp <= c_in_signed;
     END IF;
   END PROCESS IfElse_Bad_1_output;
```

2. When the condition in the if/else branch contains an operation, HDL code generation creates intermediate variables for the different conditional checks. Since these intermediate variables are only assigned in the branches of the if/else statement where they are used, this will result in latch inference by RTL synthesis.

MATLAB code that **will not** cause latch inference

```
function cOut  = fcn(a_in)

 ntype = numerictype(1,6,0);
 nt_ain = numerictype(a_in);
 a_in2 = fi(a_in+2, nt_ain);
 a_in3 = fi(a_in-3, nt_ain);

 if a_in2 > 6
     cOut = fi(1, ntype);
 elseif a_in3 > 5
     cOut = fi(2, ntype);
 else % Default
     cOut = fi(0, ntype);
 end
```

MATLAB code that **will** cause latch inference

```
function cOut  = fcn(a_in)

 ntype = numerictype(1,6,0);

 if a_in+2 > 6
     cOut = fi(1, ntype);
 elseif a_in-3 > 5
     cOut = fi(2, ntype);
 else % Default
     cOut = fi(0, ntype);
 end
```

Generated Verilog — `add_temp_1` and `sub_temp_1` written to outside of the if/else statement

```
always @(a_in) begin
  add_temp_1 = a_in + 2;
  if (add_temp_1[4] != 1'b0) begin
    a_in2_1 = 4'b1111;
  end
  else begin
```

Generated Verilog — `sub_temp_1` and `cast_1` written to inside only one branch of the if/else statement

```
always @(a_in) begin
  if ((a_in + 5'b00010) > 5'b00110) begin
    cOut_1 = 6'sb000001;
  end
  else begin
```

```
    a_in2_1 = add_temp_1[3:0];
  end
  sub_temp_1 = $signed({1'b0, a_in}) - 3;
  if ((sub_temp_1[5] == 1'b0) && (sub_temp_1[4] !=
1'b0)) begin
    a_in3_1 = 4'b1111;
  end
  else if (sub_temp_1[5] == 1'b1) begin
    a_in3_1 = 4'b0000;
  end
  else begin
    a_in3_1 = sub_temp_1[3:0];
  end
  if (a_in2_1 > 4'b0110) begin
    cOut_1 = 6'sb000001;
  end
  else if (a_in3_1 > 4'b0101) begin
    cOut_1 = 6'sb000010;
  end
  else begin
    cOut_1 = 6'sb000000;
  end
end

assign cOut = cOut_1;
```

```
    sub_temp_1 = $signed({1'b0, a_in}) - 3;
    if (sub_temp_1[5] == 1'b1) begin
      cast_1 = 5'b00000;
    end
    else begin
      cast_1 = sub_temp_1[4:0];
    end
    if (cast_1 > 5'b00101) begin
      cOut_1 = 6'sb000010;
    end
    else begin
      cOut_1 = 6'sb000000;
    end
  end
end

assign cOut = cOut_1;
```

3. When the MATLAB Function block performs a `For` loop and matrix operation and HDL Block Properties…>StreamingFactor is set to a value greater than 0 (which turns on the streaming optimization), a latch will be created for the `For` loop variable.

2.11.8 Avoid generating `always @*` Verilog code for Xilinx Virtex-4 and 5

In certain cases, Verilog code generated from MATLAB function blocks will generate Verilog code that utilizes the `always @*` construct. This will result in the following error when targeting Xilinx Virtex-4 and 5 devices:

"ERROR:Xst:1468 - "file.v" line xx : Unexpected event in always block sensitivity list"

This issue is fixed for Virtex-6 and Spartan-6 devices.

The following MATLAB code constructs will generate Verilog with the `always @*` construct:

- `for` loop
- Shift register that uses vector data
- Pipeline insertion set via the VariablesToPipeline parameter

2.11.9 Using MATLAB code for [M, N] matrix operations

Matrix operations in a MATLAB Function block are supported for HDL code generation, however the input port cannot be matrix data. Therefore use a vector data type for the input port, and convert to an MxN matrix using the `reshape` function. Resource sharing can be applied to save area, using the `SharingFactor` in **HDL Block Properties…**

Example: BS001_MLFMatrix.slx

2.11.10 Use a single `for` loop for element-by-element operations to reduce area

When performing element-by-element operations on matrix data, it is more area-efficient to use a single `for` loop than it is to use nested `for` loops that iterate row-by-column. The following table illustrates the difference:

Nested `for` loop

```
persistent m1;
 if isempty(m1)
     m1 = ones(3,3);
 end
 for ix = 1:3
     for iy = 1:3
```

Single `for` loop

```
persistent m1;
 if isempty(m1)
     m1 = ones(3,3);
 end
 for ix = 1:numel(m1
     m1(ix) = m1(ix) * u;
 end
```

`m1(ix,iy)=m1(ix,iy)*u;`
`     end`
` end`
` y = sum( sum( m1 ) );`
	` y = sum( sum( m1 ) );`

Multipliers: 11 Adders: 10 Registers: 19	Multipliers: 9 Adders: 8 Registers: 19

2.12 Stateflow

2.12.1 Choosing Mealy vs Moore for Stateflow state machine type

When using Stateflow charts, it is important to keep in mind how Stateflow semantics are realized in hardware.

Therefore the two types of Stateflow charts supported for HDL code generation are Mealy and Moore. The high-level differences between Mealy and Moore state machines are as follows:

- Mealy: outputs are a function of the current state and the inputs. This allows for more flexible usage, but are more difficult to read.
- Moore: outputs are a function of the current state only. These charts are typically easier to read, but restrict flexibility in defining state transitions. Beginning in R2015b, Moore state machines produce more efficient HDL.

2.12.2 Stateflow Chart block configuration

Bring up the Model Explorer to specify model-wide settings:

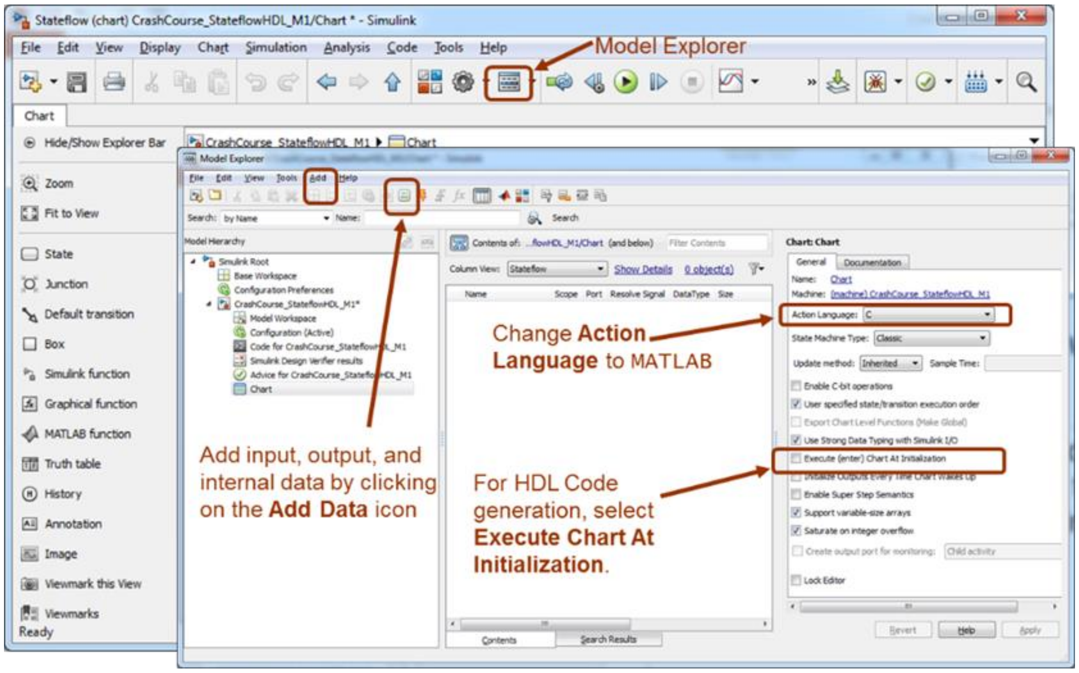


Chart: Chart

General | Documentation

Name: Chart

Machine: (machine) CrashCourse_StateflowHDL_M1

Action Language: C

State Machine Type: Classic

Update method: Inherited Sample Time:

- [ ] Enable C-bit operations
- [x] User specified state/transition execution order
- [ ] Export Chart Level Functions (Make Global)
- [x] Use Strong Data Typing with Simulink I/O
- [ ] Execute (enter) Chart At Initialization
- [ ] Initialize Outputs Every Time Chart Wakes Up
- [ ] Enable Super Step Semantics
- [x] Support variable-size arrays
- [x] Saturate on integer overflow
- [ ] Create output port for monitoring: Child activity
- [ ] Lock Editor

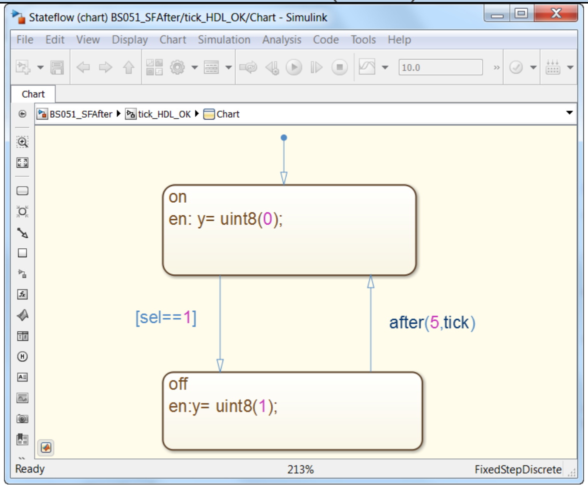
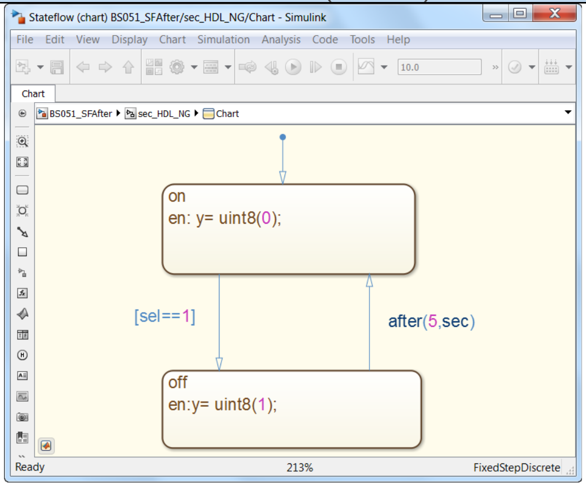
See the product documentation for Chart block implementations, properties, and restrictions for HDL code generation.

2.12.3 Do not use absolute time for temporal logical logic (after, before and every)

Temporal logic is useful in Stateflow charts to implement counter functions, such as a timeout counter. However for HDL code generation, do not use absolute time such as seconds; instead use the `tick` construct for relative time.

Example: BS051_SFAfter.slx

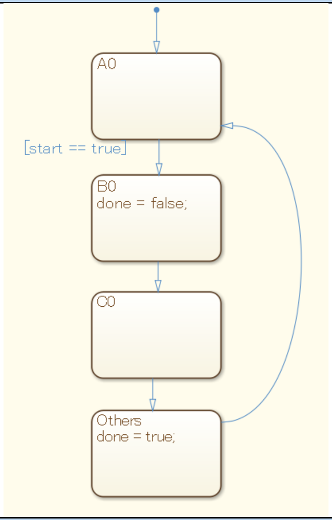
Relative time (correct)	Absolute time (incorrect)

 Stateflow (chart) BS051_SFAfter/tick_HDL_OK/Chart - Simulink
BS051_SFAfter ▸ tick_HDL_OK ▸ Chart
on
en: y= uint8(0);
[sel==1]
after(5,tick)
off
en:y= uint8(1);
Ready 213% FixedStepDiscrete File Edit View Display Chart Simulation Analysis Code Tools Help
10.0
Chart
	 Stateflow (chart) BS051_SFAfter/sec_HDL_NG/Chart - Simulink
	BS051_SFAfter ▸ sec_HDL_NG ▸ Chart
	on
	en: y= uint8(0);
	[sel==1]
	after(5,sec)
	off
	en:y= uint8(1);
	Ready 213% FixedStepDiscrete File Edit View Display Chart Simulation Analysis Code Tools Help
	10.0
	Chart

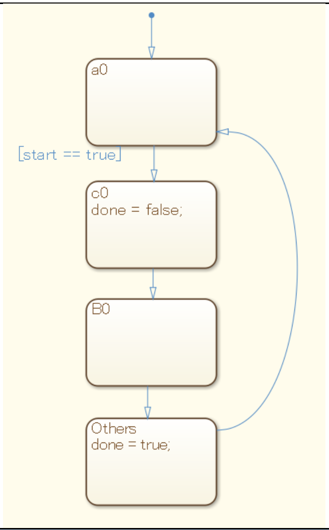
2.12.4 Consider desired state order in generated HDL when naming states

The HDL code generated from a Stateflow chart lists the conditional branches in alphabetical order (A, B, C, -- a, b, c, -- 1, 2, 3, --) according to the name of each state. Therefore it is important to take this into consideration when naming the states in Stateflow. In particular, since the `Others` (VHDL) `default` (Verilog) state is listed last in generated code, it should also be the last alphabetically named state in the chart. The following table illustrates the behavior:

Stateflow chart	Generated VHDL (state order is same as the model)

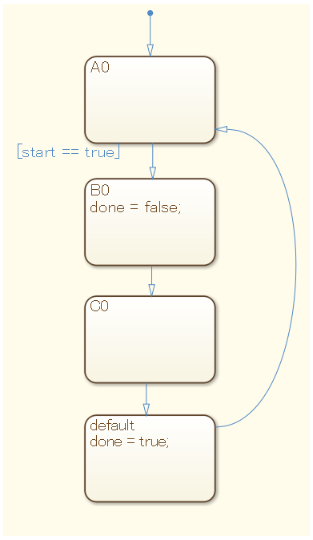
 A0
[start == true]
B0
done = false;
C0
Others
done = true;
	`CASE is_chart IS`
	`  WHEN IN_A0 =>`
	`    --During 'A0': '<S2>:1'`
	`   IF start = '1' THEN`
	`      --Transition: '<S2>:11'`
	`      is_chart_next <= IN_B0;`
	`    END IF;`
	`  WHEN IN_B0 =>`
	`    --During 'B0': '<S2>:4'`
	`    --Transition: '<S2>:68'`
	`    is_chart_next <= IN_C0;`
	`  WHEN IN_C0 =>`
	`    --During 'C0': '<S2>:71'`
	`    --Transition: '<S2>:65'`
	`    is_chart_next <= IN_Others;`
	` WHEN OTHERS =>`
	`    --During 'Others': '<S2>:72'`
	`    --Transition: '<S2>:31'`
	`    is_chart_next <= IN_A0;`
	`END CASE;`

Stateflow chart	Generated VHDL (state order is different from the model)



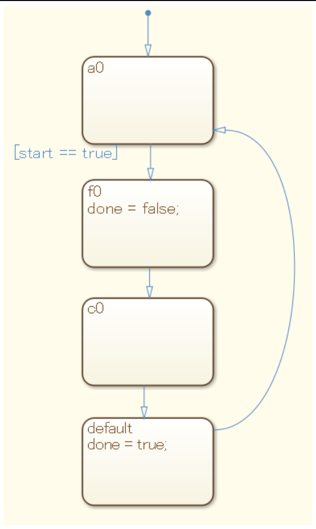
```
CASE is_chart IS
  WHEN IN_B0 =>
    --During 'B0': '<S2>:71'
    --Transition: '<S2>:65'
    is_chart_next <= IN_Others;
  WHEN IN_Others =>
    --During 'Others': '<S2>:72'
    --Transition: '<S2>:31'
    is_chart_next <= IN_a0;
  WHEN IN_a0 =>
    --During 'a0': '<S2>:1'
   IF start = '1' THEN
      --Transition: '<S2>:11'
      is_chart_next <= IN_c0;
    END IF;
 WHEN OTHERS =>
    --During 'c0': '<S2>:4'
    --Transition: '<S2>:68'
    is_chart_next <= IN_B0;
END CASE;
```

Stateflow chart	Generated Verilog (state order is same as the model)



```
case ( is_chart)
  IN_A0 :
    begin
      //During 'A0': '<S2>:1'
      if (start == 1'b1) begin
        //Transition: '<S2>:11'
        is_chart_next = IN_B0;
      end
    end
  IN_B0 :
    begin
      //During 'B0': '<S2>:4'
      //Transition: '<S2>:68'
      is_chart_next = IN_C0;
    end
  IN_C0 :
    begin
      //During 'C0': '<S2>:71'
      //Transition: '<S2>:65'
      is_chart_next = IN_default;
    end
  default :
    begin
      //During 'default': '<S2>:72'
      //Transition: '<S2>:31'
      is_chart_next = IN_A0;
    end
endcase
```

Stateflow chart	Generated Verilog (state order is different from the model)



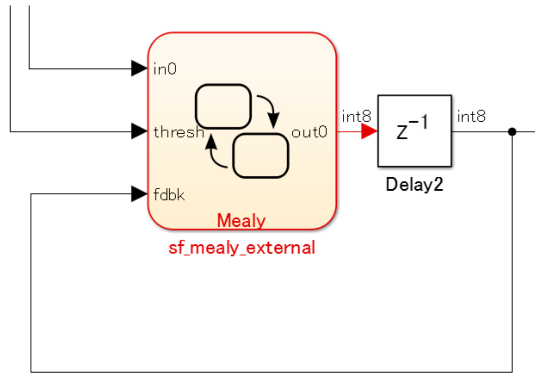
```
case ( is_chart)
  IN_a0 :
    begin
      //During 'a0': '<S2>:1'
      if (start == 1'b1) begin
        //Transition: '<S2>:11'
        is_chart_next = IN_f0;
      end
    end
  IN_c0 :
    begin
      //During 'c0': '<S2>:71'
      //Transition: '<S2>:65'
      is_chart_next = IN_default;
    end
  IN_default :
    begin
      //During 'default': '<S2>:72'
      //Transition: '<S2>:31'
      is_chart_next = IN_a0;
    end
  default :
    begin
      //During 'f0': '<S2>:4'
      //Transition: '<S2>:68'
```

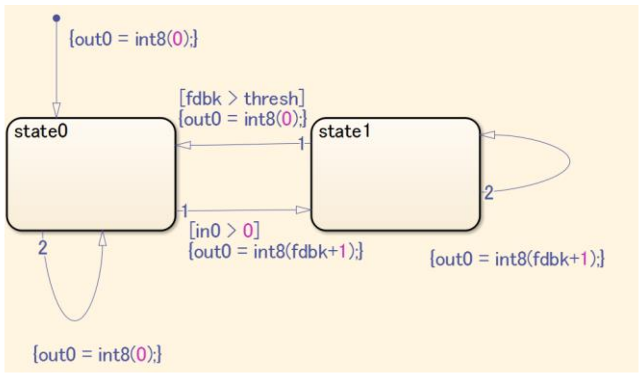
	`        is_chart_next = IN_c0;`
	`      end`
	`    endcase`

2.12.5 Using a chart output as an input via a feedback loop

To avoid algebraic feedback loop errors when feeding back the output of a Stateflow chart to be used in the input, insert a Delay (Unit) block outside the Stateflow chart. Note that beginning with R2015b, this is no longer an issue for Moore state machines.

Example: AD018_SFFeedback.slx



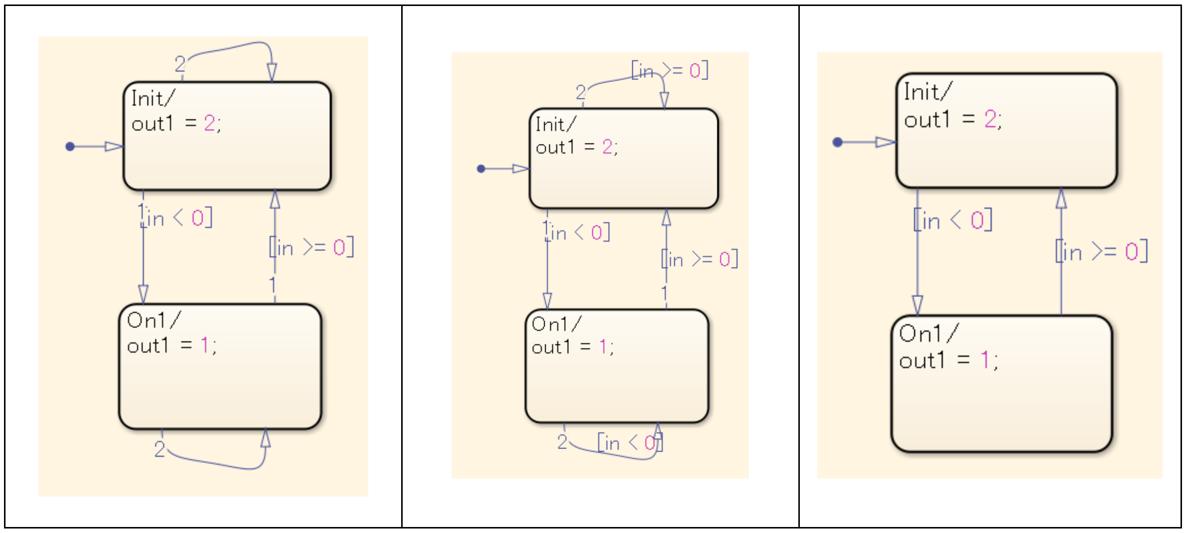


2.12.6 Insert an unconditional transition state to create an else statement in the generated HDL

In general it is good practice to insert an unconditional transition in a Stateflow chart. When generating HDL code, unconditional transitions ensure that an `Others` (VHDL) or `default` (Verilog) branch will be inserted into `case` statements, which prevents unintended latch creation during logic synthesis. The following example demonstrates correct and incorrect usage:

Example: BS033_ifElse.slx

Correct	Incorrect	Incorrect

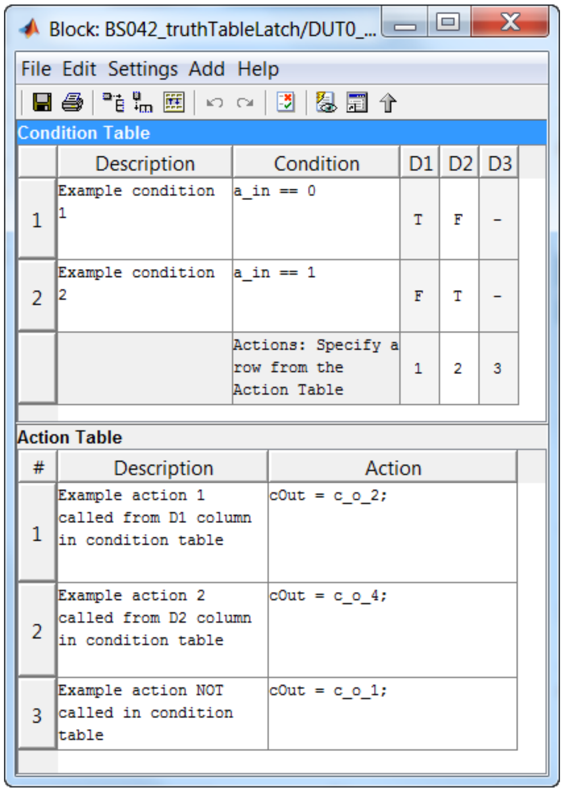


2.12.7 Avoid unintended latch inference by performing arithmetic operations outside of truth tables

Similar to [2.11.7 Avoid unintended latch inference by performing arithmetic operations outside of if/else branches](), move operation expressions outside the truth table in order to avoid inferring latches during logic synthesis. The following example illustrates:

Example: BS042_truthTableLatch.slx

Operations performed outside truth table:

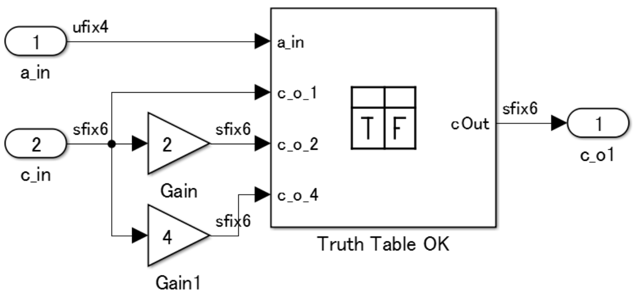

Block: BS042_truthTableLatch/DUT0_...
File Edit Settings Add Help

Condition Table

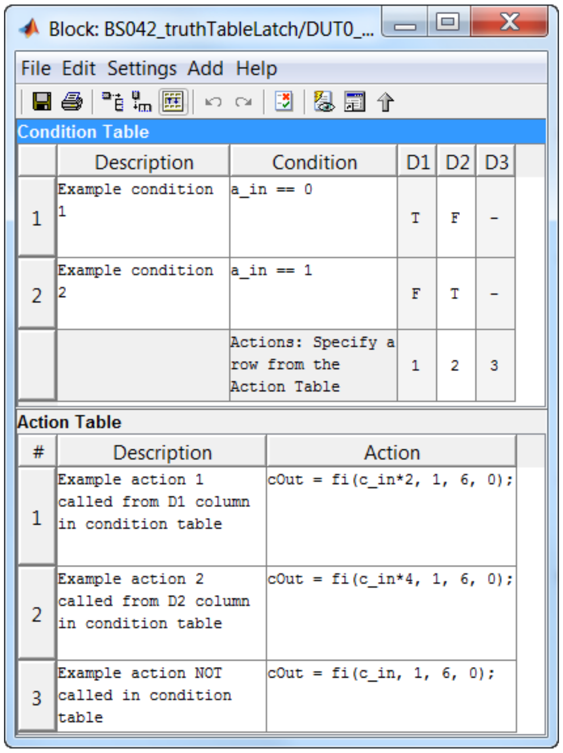
	Description	Condition	D1	D2	D3
1	Example condition 1	a_in == 0	T	F	-
2	Example condition 2	a_in == 1	F	T	-
		Actions: Specify a row from the Action Table	1	2	3

Action Table

#	Description	Action
1	Example action 1 called from D1 column in condition table	cOut = c_o_2;
2	Example action 2 called from D2 column in condition table	cOut = c_o_4;
3	Example action NOT called in condition table	cOut = c_o_1;



Operations performed in branches of truth table, which will result in latch creation during logic synthesis:

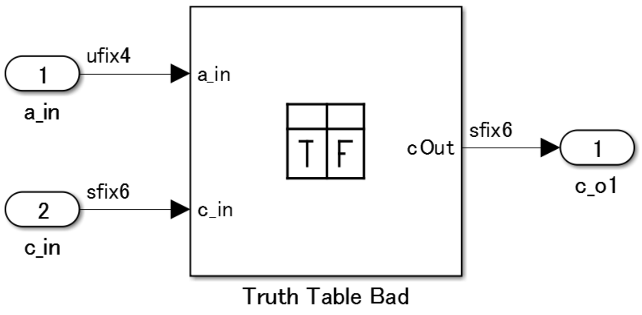

Block: BS042_truthTableLatch/DUT0_...
File Edit Settings Add Help

Condition Table

	Description	Condition	D1	D2	D3
1	Example condition 1	a_in == 0	T	F	-
2	Example condition 2	a_in == 1	F	T	-
		Actions: Specify a row from the Action Table	1	2	3

Action Table

#	Description	Action
1	Example action 1 called from D1 column in condition table	cOut = fi(c_in*2, 1, 6, 0);
2	Example action 2 called from D2 column in condition table	cOut = fi(c_in*4, 1, 6, 0);
3	Example action NOT called in condition table	cOut = fi(c_in, 1, 6, 0);



2.12.8 Hardware considerations when designing an FSM

The most important thing to remember is that code in the Stateflow chart will consume hardware resources. Here are some tips to consider when making tradeoffs to improve timing and reduce resource usage:

- Minimizing amount of redundant code, such as unnecessary transition conditions, will keep overall resource usage down
- Using a default transition will generate HDL with an “else” clause for an if/else statement or a “default” clause for a case statement, which ensures that unintended latches will not be created for undefined state transitions
- To improve timing, sometimes it helps to keep comparators for large numbers outside of the Stateflow chart, instead using their Boolean outputs as input to the Stateflow
- Sometimes having redundancy can help with meeting timing and reducing resource usage too.
- Pipelined operations are best done outside of Stateflow

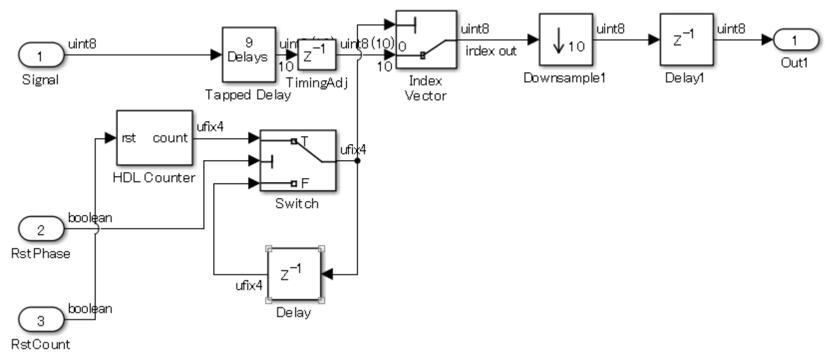
2.13 DSP System Toolbox

2.13.1 Use the DSP System Toolbox Delay block if the number of samples to delay might be 0

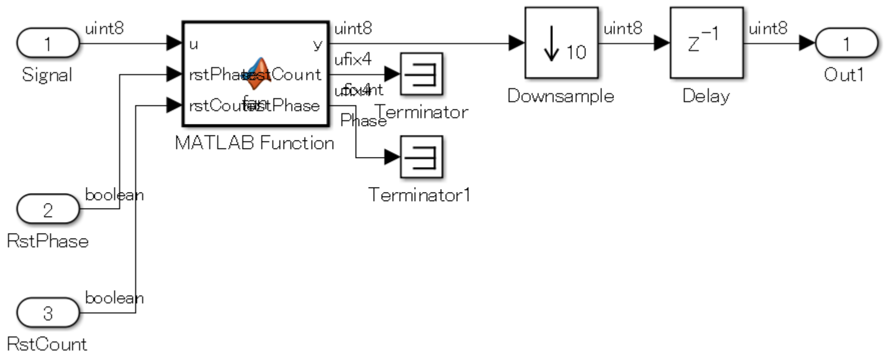
When adjusting the amount of samples to delay during the trial-and-error phase of setting your pipelining preferences, you may have the need to set the number of samples to 0. This will result in an error if you use the **Simulink/Discrete/Delay (Unit**) block. However you can set the **DSP System Toolbox/Signal Operations/Delay (Unit)** block’s number of samples to 0, so use this block in this situation.

2.13.2 Changing the phase offset of a Downsample block

The Downsample block which is used in multi-rate design does not have the ability to change its phase offset dynamically. This is controlled by the **Sample offset** block parameter, which is a static value. However you can change the downsample factor dynamically by changing the amount of delay of the input using a counter or shift register. *Example BS037_downSamplePhase.slx* demonstrates how to do this.



Alternatively, a MATLAB Function block can be used as follows:



```
function [y, testCount, testPhase] = fcn(u, rstPhase, rstCount, dsFactor)
 %#codegen

 ntReg = numerictype(u);
 ntCt = numerictype(0, ceil(log2(dsFactor)), 0);
 persistent shiftReg count dsPhase
 if isempty(shiftReg
     shiftReg = fi(zeros(1,dsFactor), ntReg);
     count = fi(0, ntCt);
     dsPhase = fi(0, ntCt);
 end

 if rstPhase
     dsPhase = count;
 end
 y = shiftReg(dsPhase+1);
 % y = shiftReg(1);
 testCount = count;
 testPhase = dsPhase;

 %% update routine
 if (count >= dsFactor-1)||rstCount
     count = fi(0, ntCt);
 else
     count = fi(count + 1, ntCt);
 end

 shiftReg = [shiftReg([2:end]) u];
```

2.13.3 Use the NCO HDL Optimized block for sine and cosine computation and signal generation

- When planning to generate HDL from a design, use the NCO HDL Optimized block for sine and cosine computations and as a numerically controlled oscillator (NCO).
- Select the parameter **Enable look up table compression method** when using the NCO HDL Optimized block to compress the table size. When this is not selected, it uses table data for a quarter-cycle for sine/cosine.
- In order to map to a ROM Lookup Table when targeting Altera or Xilinx FPGAs, set the **HDL Block Properties…>**LUTRegisterResetType to `none`.

2.13.4 Block settings for FIR filter blocks

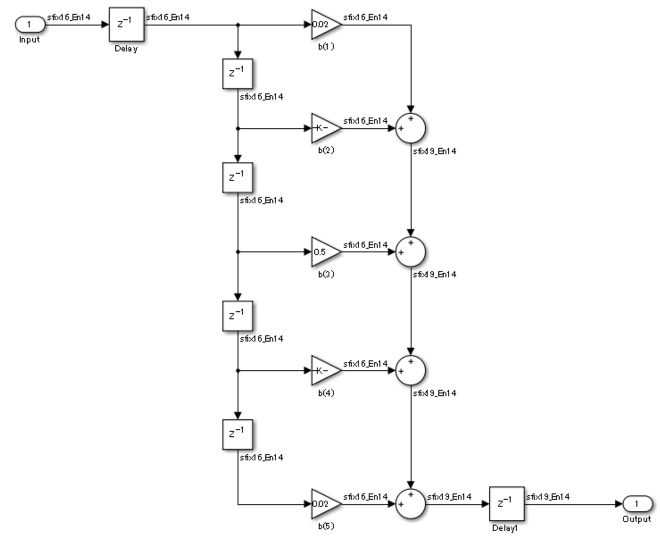
The following table summarizes the settings differences for designing a filter using building blocks versus using the Discrete FIR Filter block from the **Simulink/Discrete** or **DSP System Toolbox/Filtering/Filter Implementations** library.

	Built using building blocks	Discrete FIR Filter
Filter structure	Full customization is possible	Direct type, direct form transposed, direct type symmetry, direct type antisymmetric
Initial conditions	Can set using a Delay (Unit) block	Must be set to 0
Data type support	Anything is possible	Uses a common parameter, so individual arithmetic unit setting is not possible Unsigned fixed-point data input is not possible.
Changing the structure	Difficult – requires manual effort	Easy – select from a drop-down list
Support for complex numbers	Data/coefficient support	Data/coefficient support
HDL Block Properties	**InputPipeline**, **OutputPipeline**, **SharingFactor**, and **StreamingFactor** for the subsystem	**InputPipeline**, **OutputPipeline**, **SharingFactor**, and **StreamingFactor** for the subsystem (must specify at the block level pre-R2015b). Select the block's **Architecture** using a pulldown menu.
Programmable filter	Use `realizemdl(d,'MapCoeffsToPorts','on');` exports the filter coefficients to the MATLAB workspace where they can be tuned.	In **Function Block Parameters,** set **Coefficient source** to `input port` (direct form transposed structure not supported)
Multichannel filter	For the Delay (Unit) block set the parameter **Input Processing** to `Elements as Channels (sample based)`	Set the block parameter **Input Processing** to `Elements as Channels (sample based)`

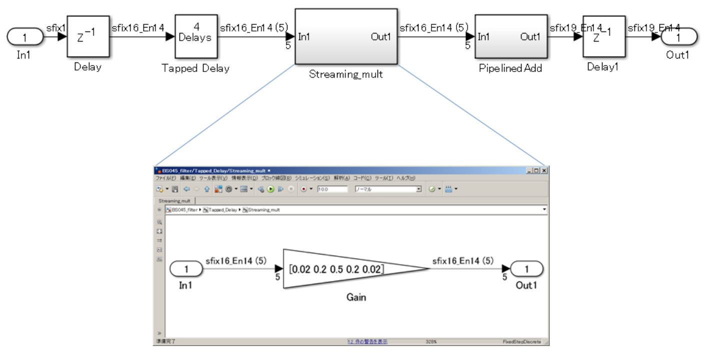
Resource sharing	Set **SharingFactor**, and **StreamingFactor** for the subsystem	Set **Architecture** as `Cascade Serial`, `Partly Serial`, or `Fully Serial`, and also set **Serial Partition** & **ResuseAccum**. See the Discrete FIR Filter HDL Coder page for more info on deciding when to use subsystem vs. block-level optimizations.

Example: BS045_FIRFilter.slx

This example demonstrates filter modeling using the Discrete FIR Filter as well as using building blocks. The PrimitiveFilter model generated by FDATool uses discrete Delay (Unit) blocks:



While the Tapped_Delay model improves readability and tune-ability by using a single Tapped Delay block:



Considerations for speed vs. area:

- When speed (minimal critical path) is the priority:
    - In **Function Block Parameters**, set **Filter structure:** to `Direct Form Transposed`
    - Insert pipeline registers at the input and output of a Product

    - For a custom filter designed with building blocks, insert Delay (Unit) blocks at the input and output of a Gain block
    - For a Discrete FIR Filter block, set the **HDL Block Properties… MultiplierInputPipeline** and **MultiplierOutputPipeline**
- When area (resource usage) reduction is the priority:
    - In **Function Block Parameters**, set **Filter structure:** to `Direct Form Transposed Symmetric` when the number of taps is even and to `Direct Form Transposed Asymmetric` when the number of taps is odd
    - If the implementation meets your timing goals, don't insert any extra pipeline registers
    - Perform resource sharing
        - For a custom filter designed with building blocks, set **SharingFactor** and **StreamingFactor**
        - For a Discrete FIR Filter block, in **HDL Block Properties…** set **Architecture** to `Fully Serial`, `Partly Serial`, or `Cascade Serial`

The **Architecture** for a Discrete FIR Filter block can be set in **HDL Block Properties…** but settings under **Implementation Parameters** can change how it is built. This will be summarized in the report generated after HDL code generation. The following table summarizes the differences between the various **Architecture** options:

	Fully Parallel	Fully Serial	Partly Serial	Cascade Serial	Distributed Arithmetic
Clock frequency	High	Lowest	Low (depending on preferences)	Low (depending on preferences)	High
Over clock	Not needed	Number of taps	Size of largest partition set by SerialPartition	Size of largest partition set by SerialPartition	Not needed
Circuit area	Large	Small	Small (depending on preferences)	Small (depending on preferences)	Large
DSP block count	Many	Few	Depends on SerialPartition setting	Depends on SerialPartition setting	Not utilized
Filter structure	All	Direct form transposed is unsupported	Direct form transposed is unsupported	Direct form transposed is unsupported	Direct form transposed is unsupported

2.13.5 IIR Filter blocks

There are two ways to implement an IIR filter:

Building one manually by combining Add, Product, and Delay (Unit) blocks, or generated by FDATool and Filterbuilder with **Realize Model** selected

Using the Biquad Filter block from the **DSP System Toolbox/Filtering/Filter Implementations** library

These two approaches are compared in the following table:

	Built using building blocks	Biquad Filter
Filter structure	Full customization is possible	Direct form I, Direct form I transposed, Direct form II, Direct form II transposed
Initial conditions	Can set using a Delay (Unit) block	Must be set to 0

Data type support	Anything is possible	Uses a common parameter, so individual arithmetic unit setting is not possible Unsigned fixed-point data input is not possible
Changing the structure	Difficult – requires manual effort	Easy – select from a drop-down list
Support for complex numbers	Data/coefficient support	Data/coefficient support
HDL Block Properties	**InputPipeline**, **OutputPipeline**, **SharingFactor**, and **StreamingFactor** for the subsystem	**InputPipeline**, **OutputPipeline**, **SharingFactor**, and **StreamingFactor** for the block Select the block's **Architecture** using a pulldown menu
Programmable filter	Change a Gain block into a Product block	Connect the source of a coefficient to an input terminal
Multichannel filter	For the Delay (Unit) block set the parameter **Input Processing** to `Elements as Channels (sample based)`	Set the block parameter **Input Processing** to `Elements as Channels (sample based)`
Resource sharing	Set **SharingFactor**, and **StreamingFactor** for the subsystem	Set **Architecture** to `Partly Serial` or `Fully Serial` Set NumMultipliers or FoldingFactor

Example: *BS047_IIRfilter.slx, designIIR.m*

In order to perform resource sharing of the product of a Biquad Filter block, set **HDL Block Properties>Architecture** to either Partly Serial or Fully Serial. When Partly Serial is selected, the number of resources used can be set. For serial architectures, you can either set the number of multipliers to use with the NumMultipliers property, or specify the amount of sharing by using the FoldingFactor property. The following table compares the implementation tradeoffs of the **Architecture** settings:

	Fully Parallel	Fully Serial	Partly Serial
Clock frequency	High	Minimal	Low, depending on settings
Over clock	Not needed	Number of taps	Size of largest partition set by SerialPartition
Circuit area	Large	Small	Small, depending on settings
DSP block count	Many	Few	Few, depending on settings
Filter structure	All	Direct form I, II	Direct form I, II

2.14 Others

2.14.1 Use case restrictions when importing user-defined HDL code with an HDL Cosimulation block

The HDL Cosimulation block is designed to import handwritten HDL code for cosimulation with the Simulink model. You can also utilize this block to import a handwritten HDL block as part of an HDL generation subsystem. In this use case it will generate the port interfaces necessary to connect the generated HDL to it.

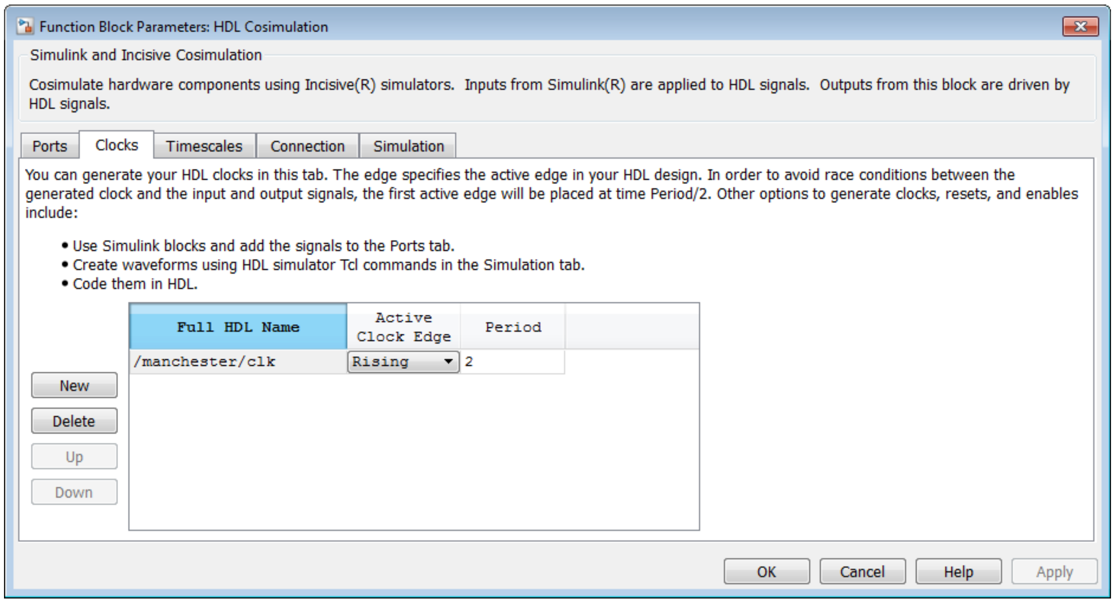
In order to do this, set the following preferences for the HDL Cosimulation block:

- Explicitly specify **Sample Time** for the output ports
- Only select **Enable direct feedthrough** for a purely combinational circuit
- Avoid using double data types in the HDL Cosimulation block

Finally, for this use case, follow the guidelines in 1.2.7 Disable code generation to insert handwritten code for a block into the generated code for the DUT.

2.14.2 Define clock and block name to match user-defined HDL settings when using an HDL Cosimulation block

When generating HDL for a subsystem that includes an HDL Cosimulation block representing imported HDL code, define the block name the same as the VHDL entity or Verilog module name, and define the clock to match the clock signal name (with full hierarchical path) in the imported HDL. Failure to define a clock signal for this block will result in a code generation error. Note that the period should be an even-numbered integer so Simulink can create a 50% duty cycle. For more information, consult the product documentation Creating Optional Clocks with the Clocks Pane of the HDL Cosimulation Block.


Function Block Parameters: HDL Cosimulation

Simulink and Incisive Cosimulation

Cosimulate hardware components using Incisive(R) simulators. Inputs from Simulink(R) are applied to HDL signals. Outputs from this block are driven by HDL signals.

You can generate your HDL clocks in this tab. The edge specifies the active edge in your HDL design. In order to avoid race conditions between the generated clock and the input and output signals, the first active edge will be placed at time Period/2. Other options to generate clocks, resets, and enables include:

- Use Simulink blocks and add the signals to the Ports tab.
- Create waveforms using HDL simulator Tcl commands in the Simulation tab.
- Code them in HDL.

Full HDL Name	Active Clock Edge	Period
/manchester/clk	Rising	2

Finally, in the **Connection** pane, set **Connection Mode** to `Full Simulation`. Code generation supports all three settings for this parameter, however using the `No Connection` setting will not allow Simulink to inherit the data types from the cosimulation block, so the block's output will default to a `double` type.

3. Data type settings

3.1 Basic data type settings

3.1.1 Use fixed binary point scaling up to 128-bit for fixed-point operations

The dynamic range of fixed-point numbers is much less than floating-point numbers with equivalent word sizes. To avoid overflow conditions and minimize quantization errors, fixed-point numbers must be scaled according to the guidelines described in the Fixed Point Designer product documentation.

For operations that result in more than 128 bits, the intermediate value of the operation can be scaled.

3.1.2 Trading off rounding error vs processing expense

Rounding mode	Function	Equivalent MATLAB Property value
Ceiling	Rounds both positive and negative numbers toward positive infinity. Note that this approach will lead to a cumulative bias toward positive numbers.	ceil
Convergent	Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. This can generate an overflow but no bias.	convergent
Floor	Rounds both positive and negative numbers toward negative infinity. Note that this approach will lead to a cumulative bias toward negative numbers.	floor
Nearest	Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity.	nearest
Round	Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity.	round
Simplest	Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.	None
Zero	Rounds both positive and negative numbers toward zero.	fix

The Fixed Point Designer documentation compares the processing cost of each method of rounding.

The following table illustrates how the various rounding modes generate VHDL and Verilog:

Rounding mode	Generated code for 16-bit input and 14-bit output (upper row VHDL, lower row Verilog)
Ceiling (Ceiling)	`ceil_out1 <= In1_signed(15 DOWNTO 2) + ('0' & (In1_signed(1) OR In1_signed(0)));`
	`assign ceil_out1 = In1[15:2] + $signed({1'b0, (|In1[1:0])});`
Convergent (Convergent)	`convergent_out1 <= In1_signed(15 DOWNTO 2) + ('0' & (In1_signed(1) AND (In1_signed(2) OR In1_signed(0))));`
	`assign convergent_out1 = In1[15:2] + $signed({1'b0, In1[1] & (In1[2] | In1[0])});`
Floor	`floor_out1 <= In1_signed(15 DOWNTO 2);`
	`assign floor_out1 = In1[15:2];`
Nearest	`nearest_out1 <= In1_signed(15 DOWNTO 2) + ('0' & In1_signed(1));`
	`assign nearest_out1 = In1[15:2] + $signed({1'b0, In1[1]});`
Round	`round_out1 <= In1_signed(15 DOWNTO 2) + ('0' & (In1_signed(1) AND (( NOT In1_signed(15)) OR In1_signed(0))));`
	`assign round_out1 = In1[15:2] + $signed({1'b0, In1[1] & (( ~ In1[15]) | In1[0])});`
Simplest	Chooses from one of the others
Zero	`zero_out1 <= In1_signed(15 DOWNTO 2) + ('0' & (In1_signed(15) AND (In1_signed(1) OR In1_signed(0))));`

	`assign zero_out1 = In1[15:2] + $signed({1'b0, In1[15] & (|In1[1:0])});`

Here are examples of HDL code generated when **Saturate on integer overflow** is selected:

VHDL	Verilog
`out1 <= "01111111111111" WHEN (In1(15) = '0') AND (In1(14 DOWNTO 13) /= "00") ELSE "10000000000000" WHEN (In1(15) = '1') AND (In1(14 DOWNTO 13) /= "11") ELSE In1(13 DOWNTO 0);`	`assign out1 = ((In1[15] == 1'b0) && (In1[14:13] != 2'b00) ? 14'sb01111111111111 : ((In1[15] == 1'b1) && (In1[14:13] != 2'b11) ? 14'sb10000000000000 : $signed(In1[13:0])));`

3.1.3 Restrictions for data type override

The Fixed-Point Tool can override the output data types of each block in the system. The only blocks that are never affected by data type override are blocks with boolean or enumerated output data types, or blocks that are untouched by it by design (for example, lookup table blocks).

Blocks not affected by data type override	Blocks not supported for automatic derivation of min/max values
HDL Counter	System Object
HDL FIFO	Serializer1D
Dual Port RAM	Deserializer1D
Dual Rate Dual Port RAM	Integer-Input RS Encoder HDL Optimized
Simple Dual Port RAM	Integer-Output RS Decoder HDL Optimized
Single Port RAM	CRC Generator HDL Optimized
Bit Concat	CRC Detector HDL Optimized
Bit Reduce	DC Blocker
Bit Rotate	Complex to Magnitude-Angle HDL Optimized
Bit Shift	NCO HDL Optimized
Bit Slice	FFT HDL Optimized
	IFFT HDL Optimized
	Vision HDL Toolbox Chroma Resampler
	Vision HDL Toolbox Edge Detector
	Vision HDL Toolbox Image Filter
	FIR Rate Conversion HDL Optimized

3.2 Simulink data type setting

3.2.1 Use Boolean for logical data and use ufix1 for numerical data

Even though both Boolean and the fixed-point ufix1 are both 1-bit data types in MATLAB and Simulink, they are treated differently

- Use `boolean` for control logic signals, e.g. enable and local reset. If a Boolean signal needs to be used in a calculation with a fixed-point data type, it can be converted to a `fixdt(0, 1)`
- Use `fixdt(0, 1)` for calculations. Because it has a `numerictype` property, it can be set to `Inherit: Inherit via internal rule` in operations where the output may need to grow to a larger bit width than the input.

3.2.2 Define the data type of a Gain block explicitly

Explicitly specify a `Simulink.NumericType` object (e.g. `fixdt (1, 16, 8)`) for the **Parameter data type** of a Gain block. Leaving it as Inherit: Inherit via internal rule could result in a data type being assigned that results in an HDL code generation error.

3.2.3 Restrictions for using enumerated values

Note that many optimizations do not work seamlessly in the presence of enumerated types, so it is recommended to use them sparingly. When generating HDL using an enumerated data type, there are a few modeling restrictions:

- An enumerated data type cannot be used for the input or output port of the top-level DUT
- As shown below, enumerated values must be monotonically increasing

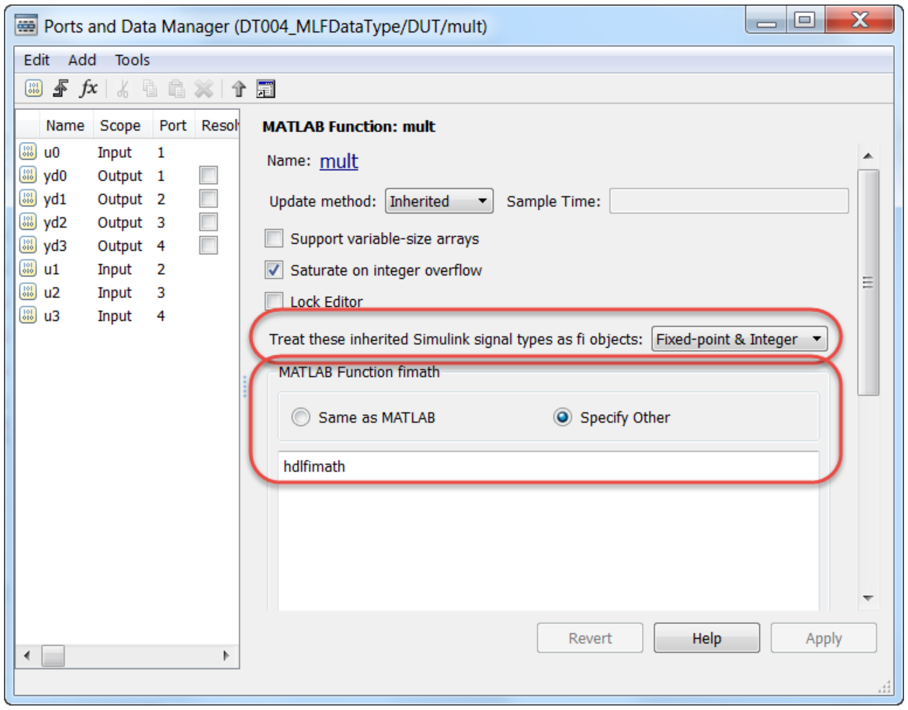
```
classdef BasicColors < Simulink.IntEnumType
   enumeration
     Red(0)
     Yellow(1)
     Blue(2)
   end
   methods (Static)
     function retVal = getDefaultValue()
       retVal = BasicColors.Blue;
     end
   end
 end
```

- An enumerated value cannot be used for arithmetic operation (*, /, -, +).
- An enumerated value cannot be used for a comparison operation (> -- < -- >=, <=, ==, and -=). However it can be used by for a <> operation or a conditional branch (if, switch).

3.3 Data type setting for MATLAB code

3.3.1 Using a fi object in a MATLAB Function block

In the MATLAB Function block go to **Edit Data>Ports and Data Manager** and set **Treat these inherited Simulink signal types as fi objects** to `Fixed-point & Integer`. For **MATLAB Function fimath**, choose `Specify Other` and specify `hdlfimath`.



Name	Scope	Port	Resol
u0	Input	1	
yd0	Output	1	☐
yd1	Output	2	☐
yd2	Output	3	☐
yd3	Output	4	☐
u1	Input	2	
u2	Input	3	
u3	Input	4	

MATLAB Function: mult

Name: mult

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 & Integer

MATLAB Function fimath

Same as MATLAB [✓ Specify Other]

hdlfimath

Using an hdlfimath consumes fewer circuit resources since it uses the following parameters:

```
RoundingMethod: Floor
OverflowAction: Wrap
ProductMode: FullPrecision
SumMode: FullPrecision
```

Choosing `Same as MATLAB` above will generate a `fimath` with the following parameters, which consumes more circuit area:

```
RoundingMethod: Nearest
OverflowAction: Saturate
ProductMode: FullPrecision
SumMode: FullPrecision
```

For operations that require different rounding than what is set by hdlfimath, define the functionality explicitly in the MATLAB Function block code.

```
>> B = fi(4.9, 1, 8)
B =
    4.8750

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 4
>> A = fi(2.3, 1, 10)
A =
    2.2969

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 10
        FractionLength: 7
>> C = fi(A+B, 'RoundingMethod', 'Nearest', 'OverflowAction', 'Saturate')
C =
    7.1719

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 12
        FractionLength: 7

        RoundingMethod: Nearest
        OverflowAction: Saturate
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

Example: DT004_MLFDataType.slx, DT007_MLFBuiltInData.slx

3.3.2 Use like or cast to inherit data types in MATLAB code

Don’t use the `numerictype` construct to inherit a data type from a `fi` object in MATLAB code. Instead use the `cast()` or `zeros()` function because they support inheritance of built-in data types `u/int8, u/int16, u/int32, u/int64, double, single, boolean`.

Example: DT005_MLFLike.slx

```
>> A=fi(2.55, 1, 17, 4)
```

```
A =

                    2.5625

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 17
        FractionLength: 4
>> B = cast(1.22, 'like', A)

B =

                      1.25

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 17
        FractionLength: 4

>> C = zeros(1,4, 'like', A)

C =

     0 0 0 0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 17
        FractionLength: 4
```

3.3.3 Use True/False instead of Boolean data in MATLAB code

Since the Boolean data type cannot be used in MATLAB code, when performing operations that require binary 0 or 1 representation, use `True`/`False`. For example:

```
if in > 3
     out = true;
else
     out = false;
end
```

3.4 Data type setting for Stateflow charts

3.4.1 Use a fi object when the Stateflow action language is MATLAB

When setting properties for a Stateflow chart, if the execution language is MATLAB, select **Treat inherited Simulink signal types as fi objects** in order to prevent errors during simulation execution.

4. Optimization of speed and area

4.1 Resource sharing

Resource sharing uses time division multiplexing to perform multiple operations on a shared hardware resource in order to reduce circuit area. In order to share resources, the input data must be serialized and the output data must then be deserialized.

4.1.1 Resource sharing requirements

There are two important parameters to control resource sharing: `SharingFactor` and `StreamingFactor`. These parameters have different restrictions, modeling methods and resulting circuit configurations, which all depend on the blocks or models to be shared. The following table summarizes the differences.

	StreamingFactor	SharingFactor
Supported blocks	Blocks That Support Streaming	Shareable resources
Requirements and limitations	Checks and Requirements for Streaming Subsystems	Requirements and Limitations for Resource Sharing
Circuit configuration	If Delay blocks exist in front of/behind certain block, the resulting delay blocks are added between a Serializer and a Deserializer. This configuration achieves a higher clock speed with Product and Gain blocks.	• For Product and the Gain blocks, Delay blocks in front of/behind the blocks are placed outside the Serializer and Deserializer. This results in lower clock speeds compared to StreamingFactor. • For an Atomic subsystem, SharingFactor achieves comparable clock speed to that of StreamingFactor since the resulting delay blocks including those inserted by Input/OutputPipeline are added between the Serializer and a Deserializer.
Circuit configuration with delay block in front of/behind target block (e.g., Product)	Serializer=>Delay=>Product =>Delay=>Deserializer Number of delay samples = StreamingFactor The Deserializer introduces 1 cycle of latency at the data (slow) rate	Delay=>Serializer=>Product=>Deserializer=>Delay The Deserializer introduces 1 cycle of latency at the data (slow) rate
Performance when sharing 8 Product blocks on an Altera CycloneV 5CGTFD9E5F35C	With RAM mapping: Logic: 525 Register: 1044 Block RAM: 512 DSP block: 1 Fmax: 179.76 MHz With register mapping: Logic: 638 Reg: 1221 Block RAM: 0 DSP block: 1 Fmax: 181.49MHz	Logic: 986 Register: 1254 Block RAM: 0 DSP block: 1 Fmax: 127.94 MHz
Related guidelines	4.1.2 Use StreamingFactor for resource sharing of 1D vector signal processing	4.1.3 Resource sharing of Gain blocks 4.1.4 Resource sharing of Product blocks 4.1.5 Resource sharing of subsystems

The blocks that support SharingFactor are as follows. Note that all parameters including data type and rounding mode must be same between the blocks to be shared.

Target block	Product block	Gain block	Other blocks

Modeling	Put Product blocks to be shared and Delay blocks for pipelining in a Subsystem. It is better to exclude other blocks to avoid unexpected resource sharing failures.	Gain blocks whose gain is a power of 2 will not be shared so that they can map to shift operations, which are more resource-efficient.	Put blocks to be shared into an Atomic Subsystem.
Block settings	Use the same parameters including rounding mode and saturated mode settings (note: this will be relaxed in R2016a)	Use the same gain parameter and input/output data types (note: this will be relaxed in R2016a)	Use identical Atomic subsystems with the same input/output data types
Related guidelines	4.1.4 Resource sharing of Product blocks	4.1.3 Resource sharing of Gain blocks	4.1.5 Resource sharing of subsystems

In addition to the description above the following requirements must be satisfied to share resources:

- In the subsystem's **Configuration Parameters>Diagnostics>Sample Time**, if **Multitask rate transition** and **Single task rate transition** are not set to `Error` it prevents resource sharing. This is taken care of automatically by `hdlsetup`
- The block to be shared must be a shareable resource.
- More than one (Unit) Delay block must be connected to its output port if the target subsystem is part of a feedback loop. Note: beginning with R2015b, this is not necessary if clock-rate pipelining is active.
- Neither an Enabled subsystem nor a Triggered subsystem can be included in an Atomic subsystem.
- An Atomic subsystem must use (Unit) Delay blocks as state elements, and cannot use other state elements such as Discrete Filter.
- When applying single-rate sharing, which is done by going into **HDL Code Generation>Global Settings>Optimization** and setting **Max oversampling** to `1`, ensure that the target subsystem to be shared is not part of a feedback loop. Note: beginning with R2015b, this is not necessary if clock-rate pipelining is active.
- Do not use a MATLAB Function that has persistent variables, loop streaming, or output pipelining
- Do not insert Scope blocks with empty input ports, which generate a sample time of `inf`.

Pay attention to the following:

- If possible, avoid mixing in the same subsystem arithmetic logic other than a Product block to be shared along with the Delay (Unit) blocks for pipelining
- Note the number of multiples of the generated over-clock for multiple subsystems. For instance, when `SharingFactor` is set to 5 and 7 for two subsystems, the resulting over-clock factor is 35 (least common multiple). In such a case, it would be better to set their `SharingFactor` for both to be the same – 7 in this case. Note: beginning with R2015b, if both subsystems are operating at the slow rate and clock-rate pipelining is active, this is not necessary.
- The Serializer and Deserializer blocks are available in HDL Operations library starting with R2014b. Alternatively, you can manually create models for this functionality.

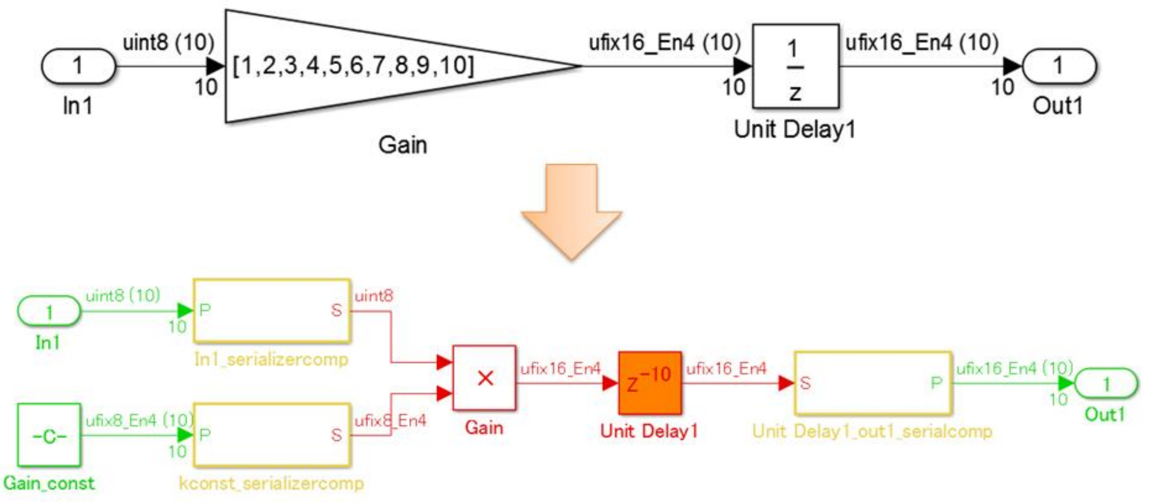
4.1.2 Use StreamingFactor for resource sharing of 1D vector signal processing

When you want reduce circuit area for a subsystem with a vector input/output that computes each N elements of the vector the same way, you can set the **HDL Block Properties** `StreamingFactor` to N. This will time-share the computation resources to reduce area. Note that the clock frequency for that logic will need to be over-clocked by N times.

To apply `StreamingFactor` to a subsystem, the following points need to be considered:

- The vector data needs to be serialized, which adds logic to the circuit as shown in the following figure. If the amount of resources saved by streaming ends up being small as compared to the overhead of serialization, circuit area could actually increase

Example: OP008_vectorStream.slx



- The region of logic that is streamed will need to run at a clock frequency of N times the parallel input in order to not increase the subsystem's latency. Thus it is important to consider the inherent delays of the FPGA/ASIC hardware and balance the maximum achievable frequency with the amount of resource savings. Note: beginning with R2015b, this is not necessary if clock-rate pipelining is active.
- When there are two or more vector signals that you want to stream, ensure that they both have the same number of elements to stream. Otherwise the clock frequency will have to be multiplies by the least common multiple of the two, possibly resulting in a target frequency that is not achievable.
- When the required frequency for streaming is not achievable, but additional delays of N cycles are acceptable, resource sharing may be an option. To apply this, in set the configuration parameter **HDL Code Generation>Global Settings>Optimization>Max oversampling** to 1, and in the same window set **Max computation latency** to an integer value greater than N. This will hold the frequency constant while increasing the latency to whatever is necessary to enable streaming. Note: beginning with R2015b, this is not necessary if clock-rate pipelining is active.

4.1.3 Resource sharing of Gain blocks

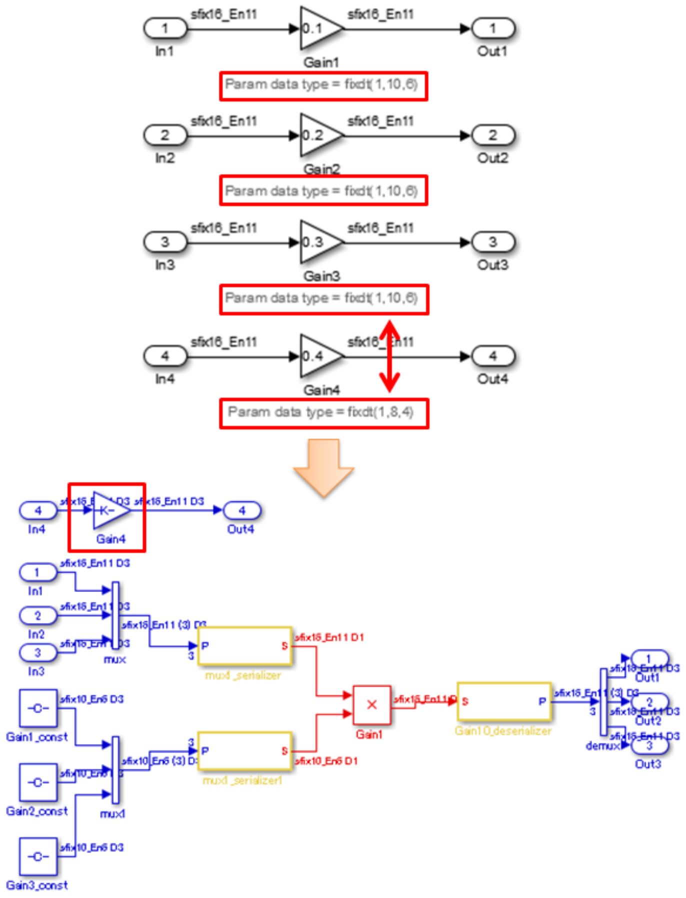
The following guidelines are best practices for sharing multiple Gain blocks:

1. Determine how the Gain block will be implemented. If the **Gain:** parameter is 0 or 1, then no logic is necessary. If the gain is 2, then a cast operation will be used to shift the logic, rather than a multiplication resource.
2. Since serialization and deserialization logic needs to be added in order to share resources, do not apply resource sharing to subsystems with a small amount of Gain blocks.
3. Determine whether resource sharing can be performed using the existing clock rate or whether oversampling is required. Using the existing clock rate requires additional latency – to do this, set the configuration parameter **HDL Code Generation>Global Settings>Optimization>Max oversampling** to 1, and in the same window set **Max computation latency** to an integer value greater than the `SharingFactor`. Note: beginning with R2015b, this is not necessary if clock-rate pipelining is active.
4. When there are two or more subsystems where resource sharing is applied requiring over-clocking, follow the guidelines in 4.1.1 Resource sharing requirements to determine appropriate settings for `SharingFactor` and `StreamingFactor`.

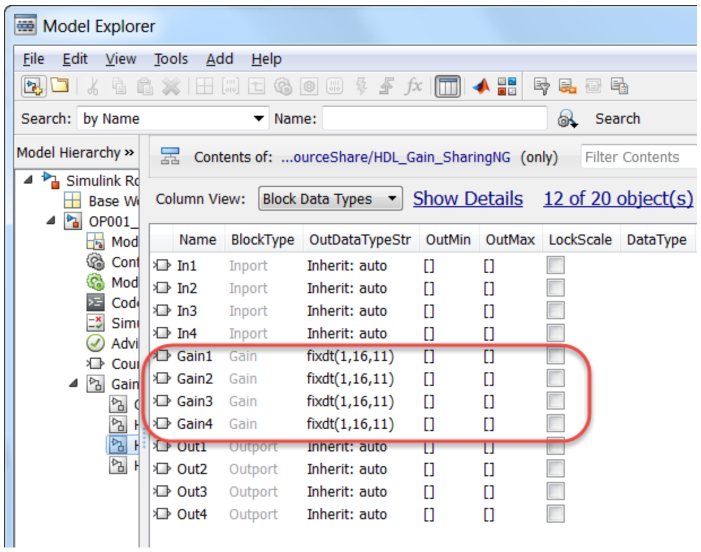
5. Apply the `StreamingFactor` optimization to Gain blocks with vector input/output.
6. Apply the `SharingFactor` optimization to Gain blocks with scalar input/output. Make sure that the Gain blocks' **Signal Attributes>Output data type** and **Parameter Attributes>Parameter data type** are explicitly set to be the same `fixdt` type. It is good practice to display the block's data type in the Simulink diagram as shown in [1.1.10 Display parameters that will affect HDL code generation](#).

Example: OP001_gainResourceShare.slx

In the example, you can see that the Gain4 block has a different data type from the other three Gain blocks, so in the generated model it is not shared.

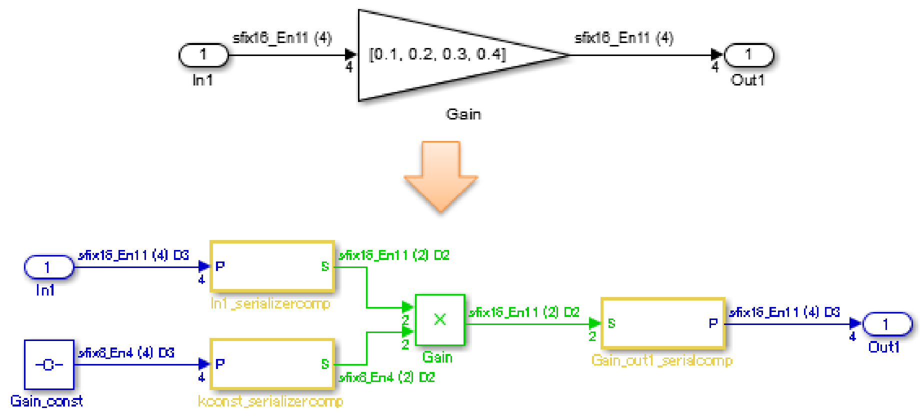


All of the data type parameters for a subsystem can be seen together in **View>Model Explorer>Model Explorer** as shown:

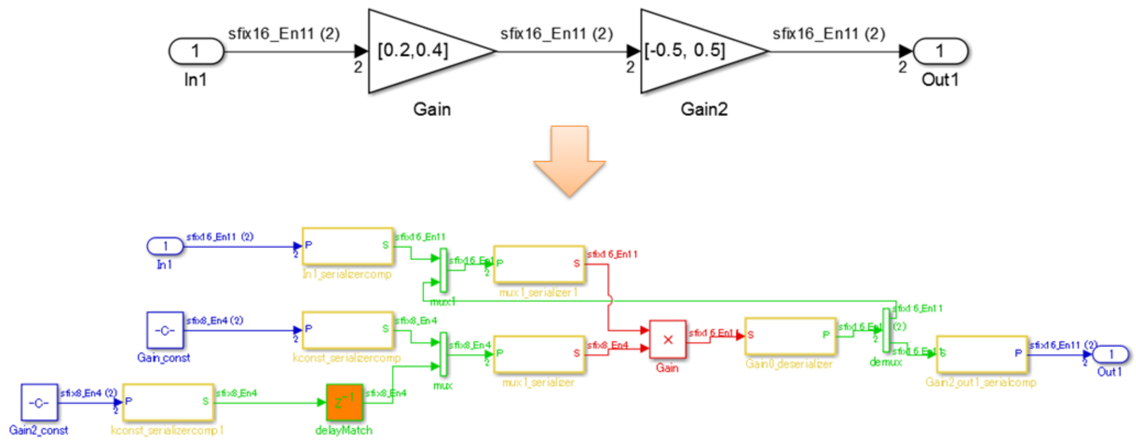


Name	BlockType	OutDataTypeStr	OutMin	OutMax	LockScale	DataType
In1	Inport	Inherit: auto	[]	[]	☐	
In2	Inport	Inherit: auto	[]	[]	☐	
In3	Inport	Inherit: auto	[]	[]	☐	
In4	Inport	Inherit: auto	[]	[]	☐	
Gain1	Gain	fixdt(1,16,11)	[]	[]	☐	
Gain2	Gain	fixdt(1,16,11)	[]	[]	☐	
Gain3	Gain	fixdt(1,16,11)	[]	[]	☐	
Gain4	Gain	fixdt(1,16,11)	[]	[]	☐	
Out1	Outport	Inherit: auto	[]	[]	☐	
Out2	Outport	Inherit: auto	[]	[]	☐	
Out3	Outport	Inherit: auto	[]	[]	☐	
Out4	Outport	Inherit: auto	[]	[]	☐	

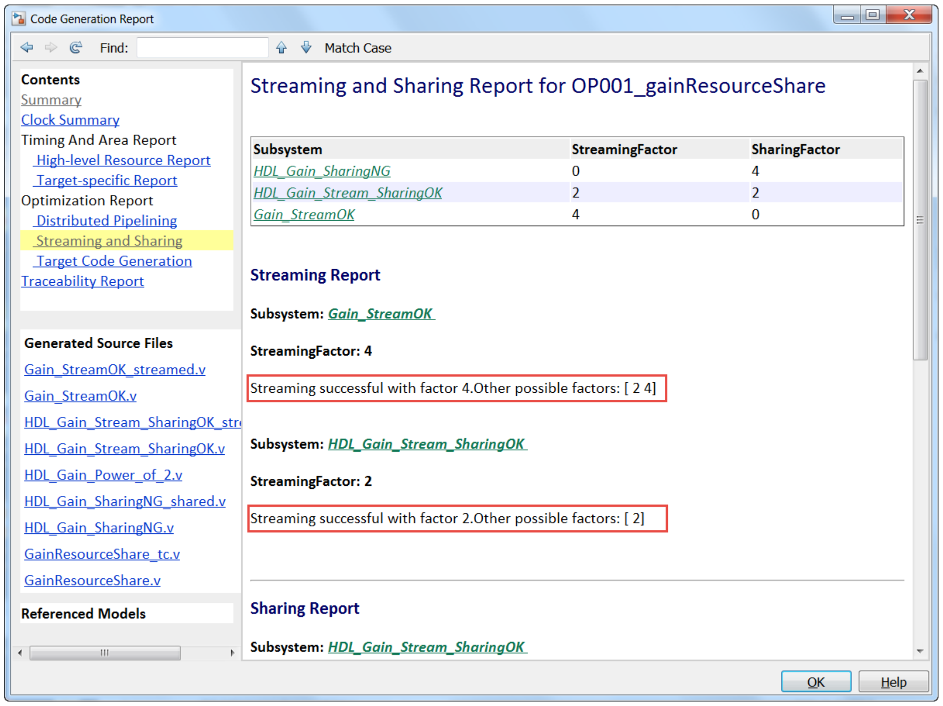
Here (the HDL_StreamOK subsystem in the example), the vector-based Gain block has `StreamingFactor` set to 4 so it is transformed to one Product block in the generated model:



In the HDL_Gain_Stream_SharingOK subsystem in the example, `SharingFactor` and `StreamingFactor` are both set to 2:



You can check the success or failure of streaming and resource sharing optimizations with the Code Generation Report. You can see the resulting circuit structure in the generated model.



Contents
Summary
Clock Summary
Timing And Area Report
High-level Resource Report
Target-specific Report
Optimization Report
Distributed Pipelining
Streaming and Sharing
Target Code Generation
Traceability Report

Generated Source Files
Gain_StreamOK_streamed.v
Gain_StreamOK.v
HDL_Gain_Stream_SharingOK_str
HDL_Gain_Stream_SharingOK.v
HDL_Gain_Power_of_2.v
HDL_Gain_SharingNG_shared.v
HDL_Gain_SharingNG.v
GainResourceShare_tc.v
GainResourceShare.v

Referenced Models

Streaming and Sharing Report for OP001_gainResourceShare

Subsystem	StreamingFactor	SharingFactor
HDL_Gain_SharingNG	0	4
HDL_Gain_Stream_SharingOK	2	2
Gain_StreamOK	4	0

Streaming Report

Subsystem: Gain_StreamOK

StreamingFactor: 4

Streaming successful with factor 4.Other possible factors: [2 4]

Subsystem: HDL_Gain_Stream_SharingOK

StreamingFactor: 2

Streaming successful with factor 2.Other possible factors: [2]

Sharing Report

Subsystem: HDL_Gain_Stream_SharingOK

4.1.4 Resource sharing of Product blocks

The following guidelines are best practices for sharing multiple Product blocks:

1. Set the Multiplier partitioning threshold to 18 for Xilinx targets or 25 for Altera targets. This will create more resource sharing opportunities for wide-bitwidth multiplier, reducing the use of DSPs on the FPGA.
2. When one of the inputs to a Product block is a constant, switch to a Gain block.
3. Follow the guidelines in 4.1.5 Resource sharing settings of subsystems to group two or more kinds of blocks (for example Product, Delay, and Add) to share implementation resources. The effectiveness of this approach will depend on the configuration of the DSP slices on your target device, but typically a series of Delay, Add and Product can be compiled to one DSP slice. Beginning with R2015b, you can use the Multiply-Add block to enable this.
4. Determine whether resource sharing can be performed using the existing clock rate or whether oversampling is required. Using the existing clock rate requires additional latency – to do this, set the configuration parameter **HDL Code Generation>Global Settings>Optimization>Max oversampling** to 1, and in the same window set **Max computation latency** to an integer value greater than the `SharingFactor`. Note: beginning with R2015b, this is not necessary if clock-rate pipelining is active.
5. When there are two or more subsystems where resource sharing is applied requiring over-clocking, follow the guidelines in 4.1.1 Resource sharing requirements to determine appropriate settings for `SharingFactor` and `StreamingFactor`.
6. Apply the `StreamingFactor` optimization to Product blocks with vector input/output.
7. Apply the `SharingFactor` optimization to Product blocks with scalar input/output. Make sure that the Product blocks' **Signal Attributes>Output data type**, **Integer rounding mode** and **Saturate on overflow** parameters are set consistently so that they can be shared. It is good practice to follow the guidelines in 1.1.10 Display parameters that will affect HDL code generation.

Example: OP002_productResourceShare.slx

4.1.5 Resource sharing of subsystems

To share resources for identical subsystems, such as when grouping Product, Add and Delay blocks to map to one DSP slice, you need to set the subsystems to be shared as one of the following types:

- Atomic subsystem
- MATLAB Function block (without persistent variables)

The following guidelines are best practices for sharing multiple subsystems:

1. Determine whether resource sharing can be performed using the existing clock rate or whether oversampling is required. Using the existing clock rate requires additional latency – to do this, set the configuration parameter **HDL Code Generation>Global Settings>Optimization>Max oversampling** to 1, and in the same window set **Max computation latency** to an integer value greater than the `SharingFactor`. Note: beginning with R2015b, this is not necessary if clock-rate pipelining is active.
2. The `SharingFactor` for subsystems cannot be set to be less than the number of Subsystems that are grouped together. For example, if you have 10 instances of an Atomic subsystem and you set the `SharingFactor` to 5, HDL Coder cannot implement resource sharing to 2 instances of the subsystem. There are two alternative ways to accomplish this:
    a. Divide the subsystem further, and share all the instances of each of the smaller subsystems
    b. Change the block names of the Atomic subsystems to be unique per your desired grouping.

    Note: this limitation is removed beginning with R2015b
3. Note the number of multiples of the generated over-clock for multiple subsystems. For instance, when SharingFactors are set to 5 and 7 for two subsystems, the resulting over-clock factor is 35 (least common multiple). In such a case, it would be better to set their SharingFactors to both be the same – 7 in this case.

Example: OP003_subsysShare.slx

The following table compares the results without and with resource sharing applied to the subsystems in the example:

Compute element	SharingFactor = 0 (No resource-sharing)	SharingFactor = 6
Multipliers	6	1
Adders/Subtractors	7	2
Registers	28	87
RAMs	0	0
Multiplexers	2	17

4.2 Pipeline insertion

Inserting extra registers as pipeline stages enables you to more easily meet your target frequency, at the cost of extra cycles of latency and extra resource requirements.

HDL Coder offers the following automated techniques with their associated parameter names:

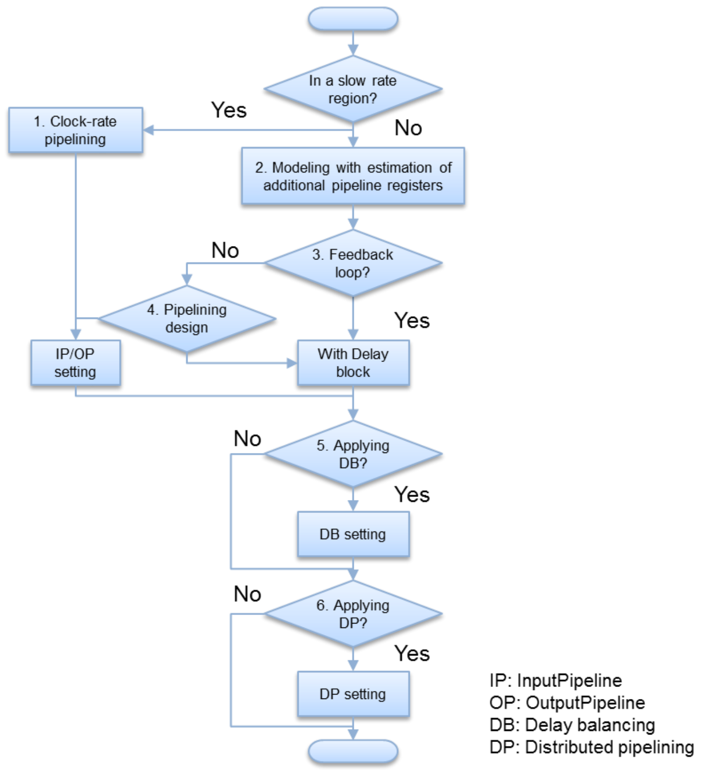
Function	HDL block property name
Pipelining design	`InputPipeline`: Set for each block/subsystem `OutputPipeline`: Set for each block/subsystem `ConstrainedOutputPipeline`: Set for each block/subsystem
Distributed pipelining	`DistributedPipelining`: Set on a subsystem
Delay Balancing	`BalanceDelays`: Set on a subsystem. For best results, leave this `'on'` (which is the default setting) unless you do not want delays inserted in a particular subsystem.
Clock-rate pipelining	`ClockRatePipelining`: Set globally, default is `'on'`

4.2.1 Design considerations for pipelining and delay balancing

The HDL Coder workflow provides the flexibility to insert and distribute pipeline registers automatically or manually. The following table compares and contrasts the approaches:

Technique	Pros	Cons
Manually-inserted and distributed Delay blocks	• Latency between the original model and the generated model does not change. • You have full control	• You need to manually balance delays for parallel paths
Delay inserted by the `InputPipeline` or `OutputPipeline` parameters	• Registers can be inserted inside multi-input Adder and Product blocks • Automatic delay balancing can be performed for parallel paths automatically • Manually-inserted Delay block distribution can also be used in conjunction with `InputPipeline` or `OutputPipeline` by turning off `PreserveDesignDelays`	• The design may be difficult to analyze because the original model does not contain the inserted Delays • The latency of the generated model would be different from that of the original model • `InputPipeline` or `OutputPipeline` cannot be used in a subsystem with a feedback loop

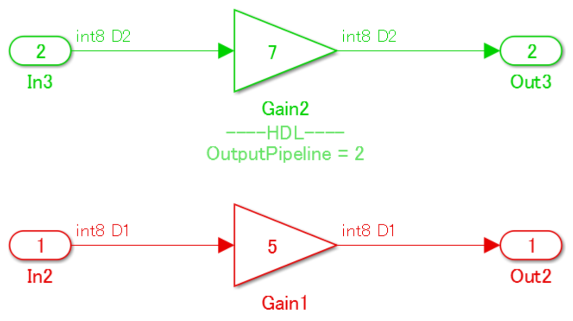
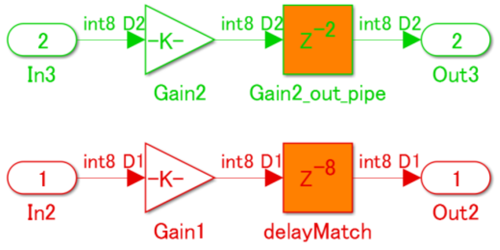
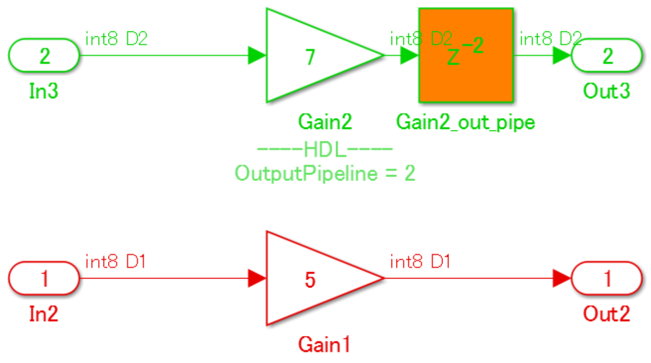
The following flow chart outlines a recommended approach to pipeline insertion and distribution:



1. If the region of your design where the implementation will insert pipeline registers is running at a slower rate than the clock rate, determine whether clock-rate pipelining will be automatically applied as outlined in 4.2.2 Clock-rate pipelining. If so, pipeline registers can be inserted with no or minimal effect on overall design latency.

2. Designing with estimated additional pipeline registers: This applies especially to models where it is good practice to manually insert Delay blocks into the original model to balance those that will be inserted during HDL code generation, as outlined in [2.6.9 Model the delay of blocks that will be auto-pipelined (Divide, Sqrt, Trigonometric Function, Cascade Add/Product, Viterbi Decoder)]
3. Feedback loop: in subsystems with feedback loops and where clock-rate pipelining cannot be applied, you will have to manually insert any desired pipeline stages.
4. Choose whether to insert pipeline registers automatically or manually using Delay blocks.
5. Delay balancing is used to keep parallel paths at the same latency when pipeline registers are inserted into one of them. If delay balancing is applied to the subsystem, the generated model will have matching delays on both paths.

   This even works in multi-rate designs. Consider the following example, where the logic in path D1 (red) has a sample time of 1 while the logic in path D2 (green) has a sample time of 4. `OutputPipeline` is set to insert 2 Delays on D2. Therefore D1 needs 4*2=8 Delays inserted, which is performed automatically with delay balancing:

Original model (Sample times: D1=1, D2 = 4)	 int8 D2 2 In3 7 Gain2 ----HDL---- OutputPipeline = 2 int8 D2 2 Out3 int8 D1 1 In2 5 Gain1 int8 D1 1 Out2
Generated model with delay balancing	 int8 D2 2 In3 -K- Gain2 int8 D2 z^-2 Gain2_out_pipe int8 D2 2 Out3 int8 D1 1 In2 -K- Gain1 int8 D1 z^-8 delayMatch int8 D1 1 Out2
Generated model without delay balancing	 int8 D2 2 In3 7 Gain2 int8 D2 z^-2 Gain2_out_pipe int8 D2 2 Out3 ----HDL---- OutputPipeline = 2 int8 D1 1 In2 5 Gain1 int8 D1 1 Out2

Delay balancing works hierarchically, so applying it to a subsystem will also apply it to all of the subsystems below in the hierarchy. In cases where you wish to turn it off for a particular subsystem, see Disable Delay Balancing for a Subsystem in the HDL Coder User Guide.

For the most up-to-date list of blocks not supported for delay balancing, refer to Delay Balancing Limitations in the HDL Coder User Guide.

4.2.2 Clock-rate pipelining

As described in sections 2.6.9, 2.6.11, and 4.2.1, HDL Coder will insert registers during implementation for a variety of reasons. In many cases, these registers will be inserted into regions at the design that run at a slower rate than the base clock rate. Introduced in R2014b, clock-rate pipelining will insert registers that run at the fast clock rate so as to avoid or minimize extra latency.

Clock-rate pipelining, which is on by default, works in conjunction with the following optimizations:

- Input and output pipelining
- Multi-cycle block implementations, such as complex math operations like sqrt and reciprocal.
- Floating-point library mapping
- Delay balancing
- Resource sharing and streaming (beginning with R2015b)

For best results, enable hierarchy flattening and take note of the blocks which inhibit clock-rate pipelining.

4.2.3 Recommended distributed pipelining settings

The following distributed pipelining settings are available in **HDL Coder Properties > HDL Code Generation > Global Settings > Optimization**:

- Hierarchical distributed pipelining: specify whether to apply distributed pipelining across hierarchical boundaries
- Clock-rate pipelining: specify whether to insert registers at the clock-rate instead of the data rate for multi-cycle paths
- Allow clock-rate pipelining of DUT output ports: specify whether to insert registers at the clock-rate instead of the data rate at the DUT output ports
- Preserve design delays: specify whether to prevent already-inserted delay blocks from being moved
- Distributed pipelining priority: specify whether the priority should be `Numerical Integrity` or `Performance`

To apply distributed pipelining to a subsystem, in HDL Block Properties set **DistributedPipelining** to on. The subsystem must meet the following requirements:

- Remove feedback loops from the target subsystem unless they are in a slow-rate region
- Avoid using unsupported blocks. The following workarounds can be applied for the specified unsupported blocks :
    - Tapped Delay, Dot Product: Build a subsystem to place them in and disable **Hierarchical distributed pipelining**. Note: Tapped Delay is supported for distributed pipelining beginning with R2015b.
    - Enabled subsystem: Change **Distributed pipelining priority** to `Performance` if possible
    - Unit Delay Resettable: Change **Distributed pipelining priority** to `Performance` if possible
- There should be no blocks with **Sample Time** of `inf`. Set blocks' sample times explicitly or to inherit (`-1`)
- Remove any empty input ports on Scope blocks since **Sample Time** is assumed to be `inf`.

The following blocks support pipelining insertion within their generated blocks. For example, in a three-or-more vector-input adder (Sum of Elements), registers can be inserted between the multiple tree-based blocks. See how to set this in 4.2.3 Apply distributed pipelining to adders, products, min/max, and dot products with vector inputs.

Block name	Requirements
MATLAB Function	Set **DistributedPipelining** to `on` in **HDL Block Properties**
Sum (of Elements)	For a greater-than-3 vector input: Set **Architecture** to `Tree` in **HDL Block Properties**
Product (of Elements)	
MinMax	

Dot Product	For a greater-than-3 vector input: Set **Architecture** to `Linear` in **HDL Block Properties**

Other considerations when using distributed pipelining:

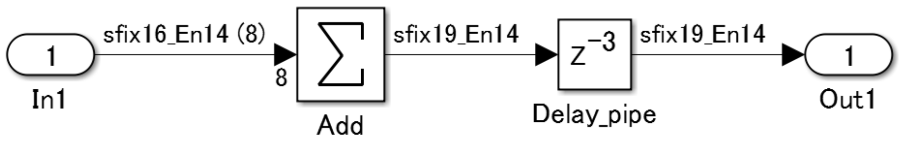
- When using manually-inserted registers, turn on Preserve Design Delays
- Be aware of the individual block settings for **DistributedPipelining** when using hierarchical distributed pipelining. If a subsystem does not have it turned on, it can limit the flexibility to move registers as needed.
- If the upper level of hierarchy does not have **DistributedPipelining** turned on, it will limit the flexibility to move registers from one subsystem that has it turned on to another. If **DistributedPipelining** cannot be turned on at the upper level, consider using **FlattenHierarchy** to enable broader pipelining.
- If Distributed pipelining priority is set to `Performance`, registers could be inserted into blocks with initial values such as Constant blocks, which would affect simulation results until values propagate through the system.
- Delay block setting values may be initialized automatically, so Delay blocks that have **Initial condition** explicitly set may be changed
    - The HDL Block Properties settings for Delay blocks inserted into the generated model by distributed pipelining will set **ResetType** to `default` and **UseRam** to `off`.
    - In the generated and validation models, the Delay blocks inserted by distributed pipelining have the naming convention rd_*index#*. Therefore you can check whether they are replaced or not.
    - If you want to prevent specific Delay blocks from being moved, turn on **Preserve Design Delays**.
- The results of `ConstrainedOutputPipeline`, which inserts registers as specified at the block outputs, should be confirmed.
    - The total delay samples including those inserted by `InputPipeline/OutputPipeline` in an original model should not be less than the value set by `ConstrainedOutputPipeline`, since `ConstrainedOutputPipeline` specifies the required number of delays samples at the output.
    - The results of ConstrainedOutputPipeline can be checked in the Distributed Pipelining results in the code generation report. When the number of registers inserted in the generated model equals this setting, the report will show that the block's status as “Passed”.

4.2.4 Apply distributed pipelining to adders, products, min/max, and dot products with vector inputs

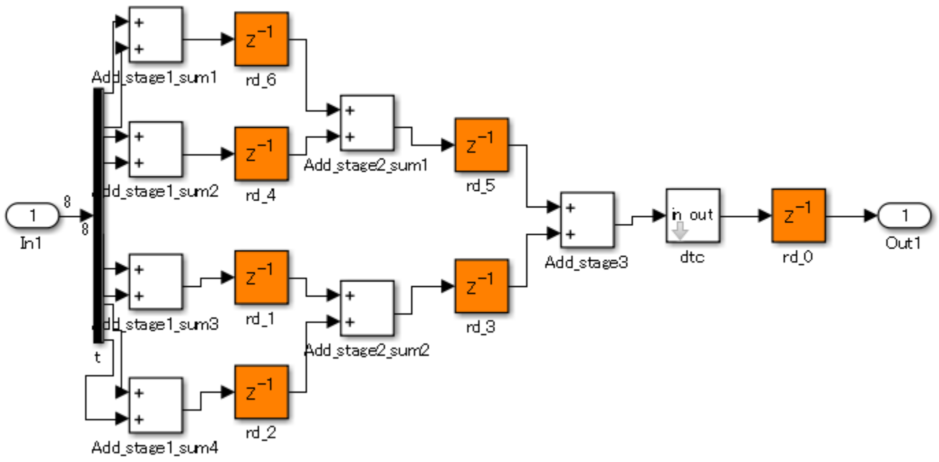
To distribute pipelining within the following blocks with inputs of vector size >3:

- Adders (Add, Sum, Subtract, Sum of Elements)
- Products (Product, Product of Elements)
- Min/Max (Simulink/Math Operations/MinMax, DSP System Toolbox/Statistics/Minimum, Maximum)
- Dot Product (real input)

1. Set **Architecture** in HDL Block Properties as follows:
    - For adders, product, and min/max, set to `Tree`. Pipeline registers will not be inserted if this is set to `Linear` or `Cascade`.
    - For Dot Product, set to `Linear`. Pipeline registers will not be inserted if this is set to `Tree`.
2. Set the number of pipeline stages, either via `InputPipeline/OutputPipeline` or by manually inserting Delay blocks.
    - *Example OP005_addProdPipe.slx* shows 3 delays that were manually inserted into the design that will be distributed within the generated model for the Add block



3. Turn on the **DistributedPipelining** in HDL Block Properties.
4. Check the results by examining the generated model, which is created in the HDL source output directory and linked from the Distributed Pipelining section of the code generation report. For the example design, the generated model looks like this:



5. Appendix

5.1 Considerations in HDL code writing for ASIC/FPGA design

Many ASIC and FPGA design projects follow a set of coding style rules for the following reasons:

- Readability and re-usability
- Avoidance of non-deterministic behavior that can lead to bugs
- Area savings and speed improvements in logic synthesis
- Easier debugging
- Prevention of potential manufacturability workflow issues

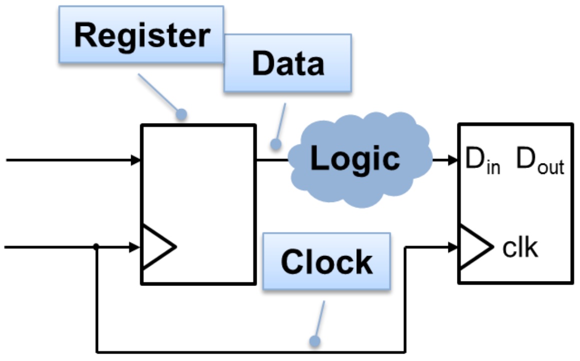
Many companies follow the HDL coding style rules provided by the Japan Semiconductor Technology Academic Research Center (STARC). This guide annotates STARC rules to its guidelines where applicable.

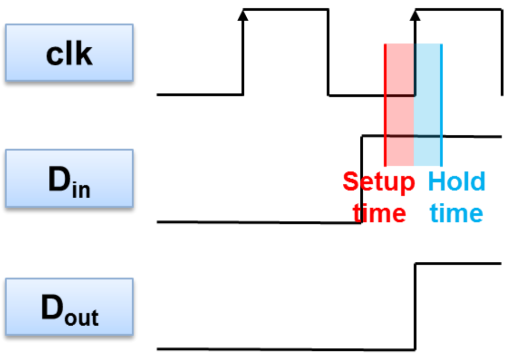
Since HDL Coder automatically generates HDL from a Simulink design, the guidelines for this workflow are focused mainly on design style and structure, along with naming conventions.

5.2 Synchronous circuit design overview and recommendations

A synchronous circuit in digital design uses a clock signal to synchronize the movement of data through the logic paths. Without a clock signal it would be tremendously difficult to coordinate the arrival timing at logic blocks for every signal in a design as it moves through logic and wires that contain delay. This is why manual asynchronous design is used only in special cases. Logic synthesis tools require synchronous design, which is why it is the recommended design style for digital designs.

The following concepts apply to synchronous design:

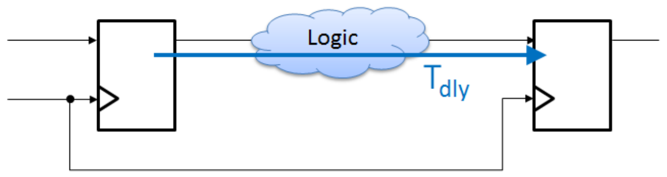


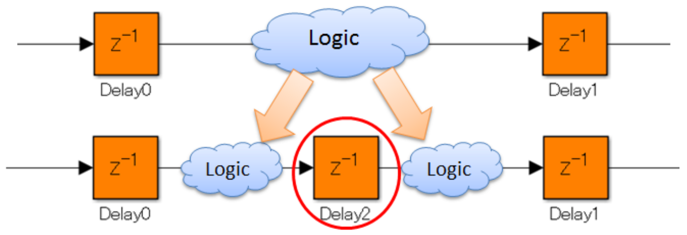


- Registers and clock signals. Registers store data until the active edge of the clock triggers them to pass the data through to the output. This is how logic is synchronized and coordinated throughout the circuit. Note that clock (and register reset) signals do not appear in Simulink – these are created during HDL code generation.
- Timing analysis. All logic, registers and wires contain delay. Timing analysis checks whether data signals are able to travel from register-to-register through combinational logic within one clock cycle (unless specified as a multi-cycle path exception). Furthermore the data signal must abide by the registers' setup and hold requirements.
- Setup and hold time. In order to prevent non-deterministic behavior of the data signal changing at the same time as an active clock edge, timing analysis checks whether data signals arrive with enough early margin (setup time), and maintain their value for enough margin after the active edge (hold time).
- Critical path. This is the path in the design that has the most amount of delay. Sometimes this longest path is still shorter than the clock period so there is no violation (it has "positive slack"), and sometimes this

longest path arrives late, causing setup violations (it has “negative slack”). The critical path determines the maximum frequency at which the circuit can operate – Fmax = 1 / T_{dly}

For a design that does not meet your target frequency (the critical path is too long), if it is too much for the hardware design team to fix during logic synthesis, you may consider breaking the path by adding a Delay:

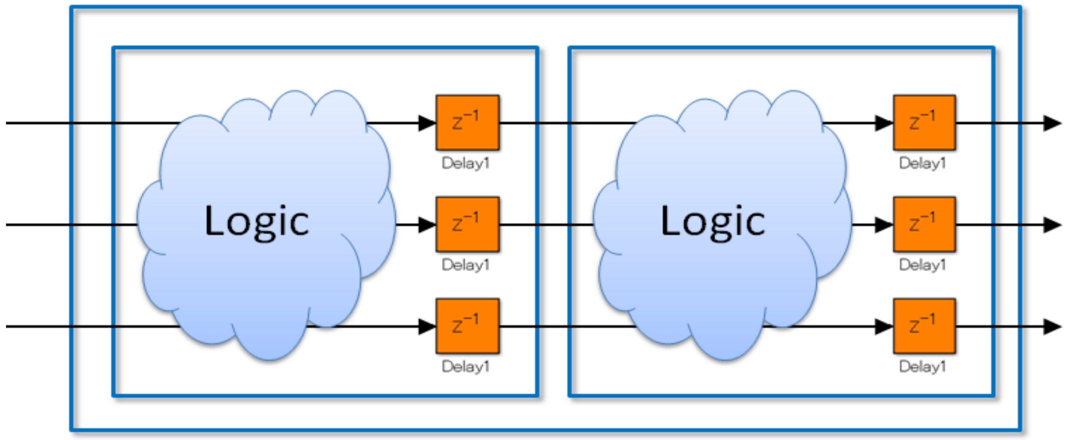




An easy way to accomplish this with minimal manual input is to use Automatic Iterative Optimization. If you want more control over pipelining, see section 4.2 Pipeline insertion.

5.3 Recommended use of registers at outputs of hierarchical structures

In large designs, often logic synthesis and timing analysis is performed at a certain level of hierarchy, and the results are integrated into the top-level design. Since synthesis and timing analysis is not aware of how much delay might exist after the output ports since that is outside the hierarchy, it is good practice to insert registers just before the outputs of these hierarchical boundaries. For example:



Some design teams register both the inputs and the outputs, since timing analysis runs register-to-register. However if they know that every subsystem’s outputs are registered and that there will be no logic between the subsystems, then they can just register the outputs.

5.4 Follow naming conventions

In an HDL description, a naming convention specifies how modules, instances, signals, and ports are named in order to increase readability, re-usability, debugging efficiency, etc. Example naming conventions often include:

- Use of only alphanumeric characters and the “_” character

- Avoid using Verilog/VHDL/SystemVerilog reserved words
- Avoid names starting with “VDD”, VSS”, “GND”, etc.
- Don’t rely on capitalization to distinguish between names (“Data” vs “data”, etc.)
- Use meaningful names as much as possible, for instance a RAM address signal might be “ram_addr” rather than “ra”

HDL Coder will generate HDL with names that correspond to your Simulink input/output ports, signals, blocks, and subsystems, so it is important to understand if your hardware team follows certain naming conventions

5.5 HDL-supported blocks

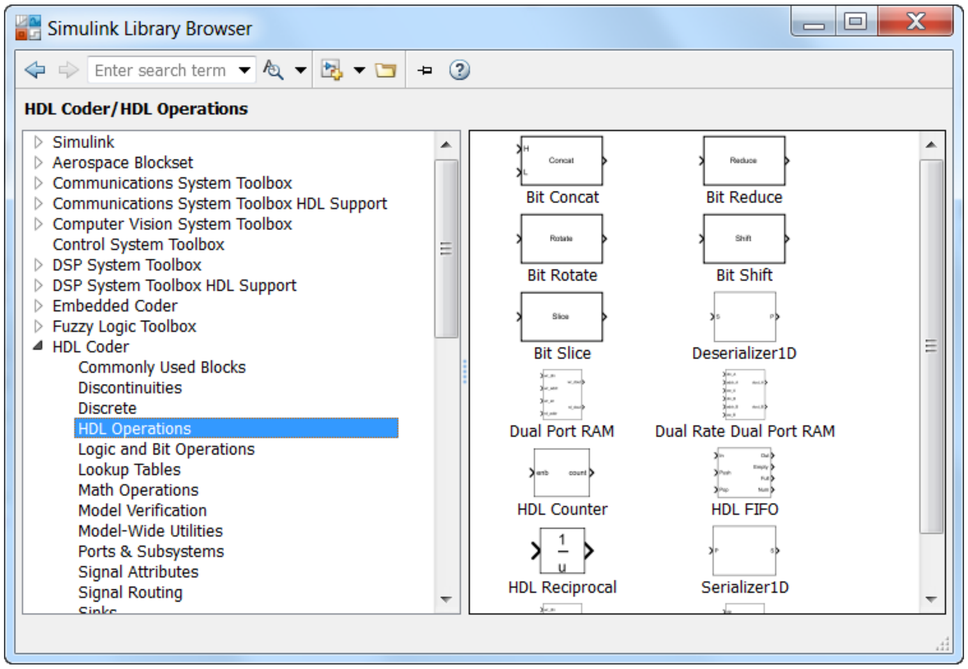
In the **Simulink Library Browser > HDL Coder** library, you can find Simulink blocks that are compatible with HDL code generation. In many cases, the blocks are also pre-configured with HDL-friendly settings, compared to the same blocks in the regular Simulink library.

- The sub-library **HDL Coder > HDL Operations** contains blocks specific to HDL applications such as RAMs, bit operations and a counter with common controls.
- The HDL Coder library blocks come with Simulink, so you can share your models and collaborate with colleagues who may not have access to HDL Coder.

In R2015a:

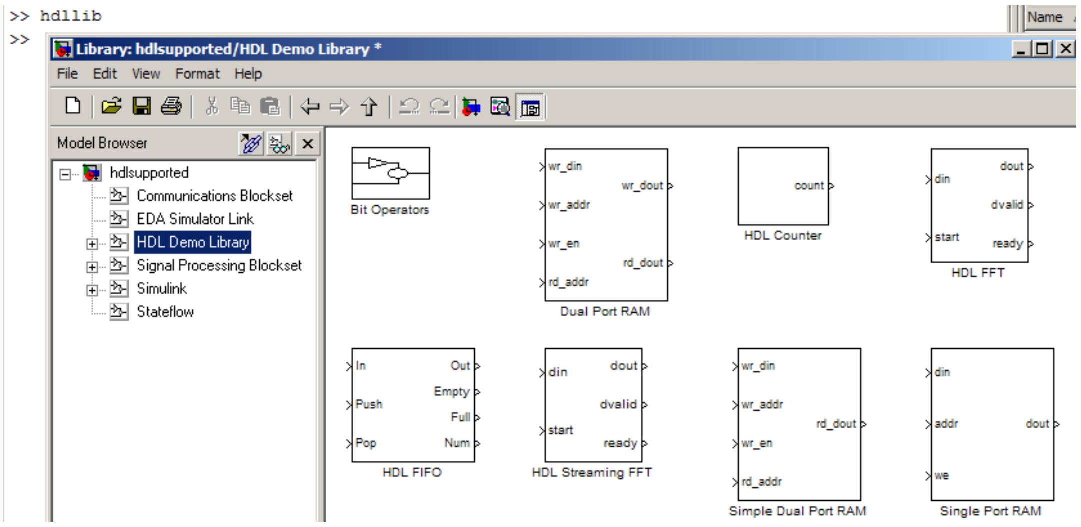
Additional blocks for signal processing, communications and computer vision are available in the library browser if you have the optional toolboxes installed. These can be found in:

- DSP System Toolbox HDL Support
- Communications System Toolbox HDL Support
- Vision HDL Toolbox



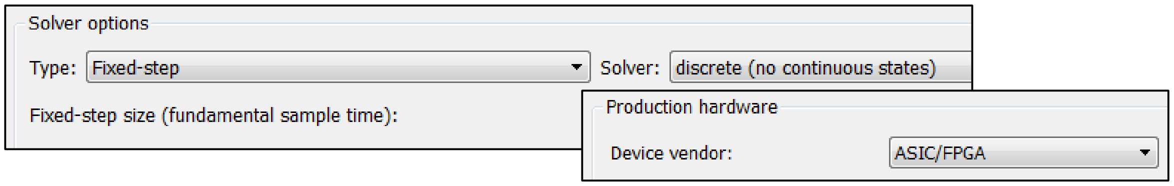
In R2014b or before:

- Use the `hdllib` utility to create a library of all blocks that are currently supported for HDL code generation, including blocks provided with DSP or Communications System Toolboxes.


```
>> hdllib
>>
```

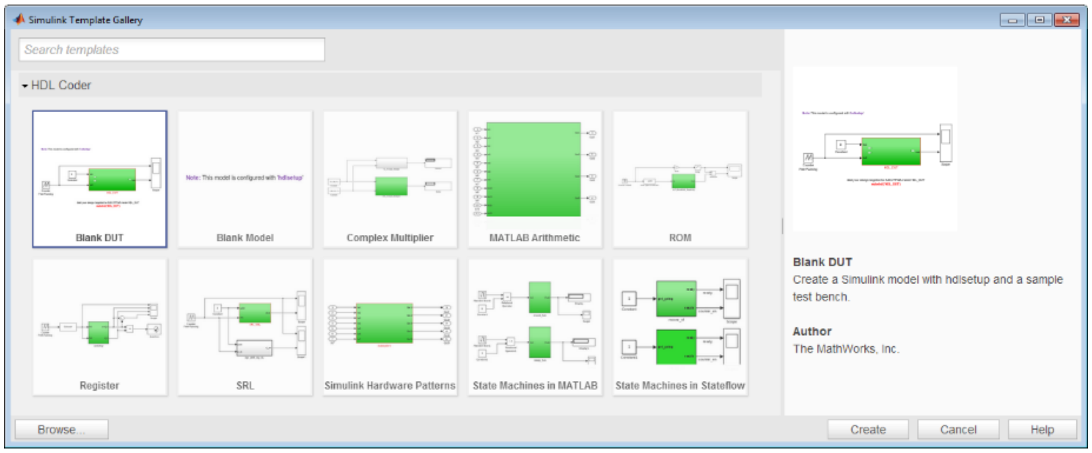
5.6 Compatibility check for HDL code generation

When creating a new model, you can quickly configure its parameter settings with the command `hdlsetup('<your_model_name>')`. Some of these settings are necessary for HDL code generation, such as using a discrete, fixed-step solver, and setting hardware target device to ASIC/FPGA, which changes the fixed-point inheritance rules for the model. Other settings help you create and debug your designs, such as turning on sample time color and signal data type display.


Solver options
Type: Fixed-step Solver: discrete (no continuous states)
Fixed-step size (fundamental sample time):
Production hardware
Device vendor: ASIC/FPGA

You can examine all the settings applied using the command edit `hdlsetup.m`.

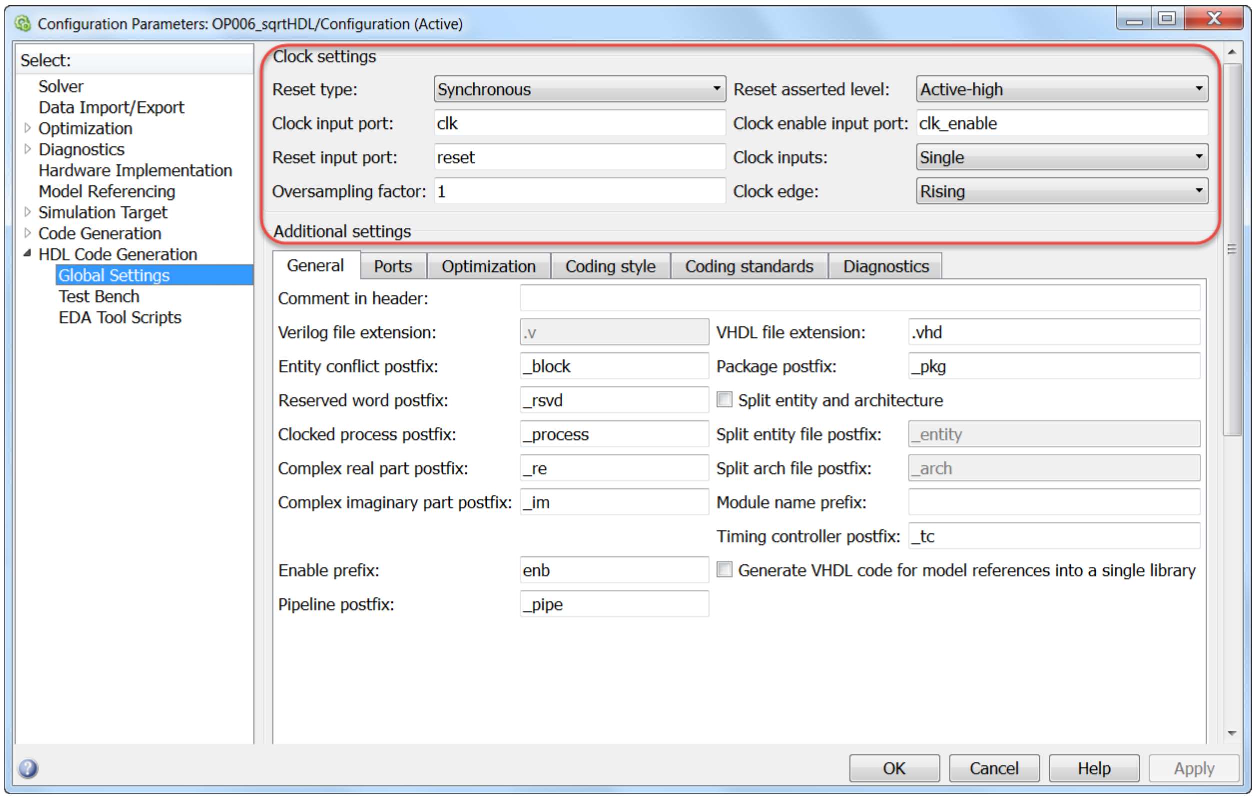
In R2015a, you can get started by using templates pre-configured for HDL code generation. Go to **New > From Template** and scroll down to the HDL Coder to choose from a variety of starting points:



Blank DUT
Create a Simulink model with hdlsetup and a sample test bench.
Author
The MathWorks, Inc.

5.7 Setting global clock and reset signals in for HDL code generation

Registers are generated from Simulink Delay blocks and MATLAB persistent variables. The clock and reset signals for registers do not appear in Simulink or MATLAB, they are generated by HDL Coder. The global names for these signals can be set before code generation in **HDL Code Generation > Global Settings**:



Configuration Parameters: OP006_sqrtHDL/Configuration (Active)

Clock settings

Reset type:	Synchronous	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 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	Module name prefix:	
		Timing controller postfix:	_tc
Enable prefix:	enb	☐ Generate VHDL code for model references into a single library	
Pipeline postfix:	_pipe		

5.8 Add comments for generating readable HDL code

You can set comments on Simulink blocks that will be generated as comments in the HDL code for better readability, debugging, and re-usability. The following set of comments illustrate a good documentation practice:

- [Explanation] of [block property] of a block
- [Explanation] of [block property] of a subsystem
- Block Annotation (double-click the Block property token you wish to set)
- DocBlock
- Model information

For information on how to do this, see 1.1.6 Document block name, block features, authors, etc., in subsystem block properties.