HDL Coder™ Evaluation Reference Guide (R2018b)

HDL Coder generates synthesizable VHDL® or Verilog® from your Simulink® or MATLAB® design so that you can implement it in an FPGA or ASIC. Focusing on the Simulink-driven flow, this guide outlines recommended practices for creating a design that can be implemented in hardware efficiently, and for optimizing the generated HDL for your targeted device. The examples that come with this guide can be easily accessed by running in MATLAB the file entitled “openHdlExampleMenu.m” in the “Examples” directory.

1. Getting Started

You can model a wide variety of systems in Simulink, but hardware design has certain requirements that are best addressed as early in the design process as possible.

1.1 HDL-supported blocks

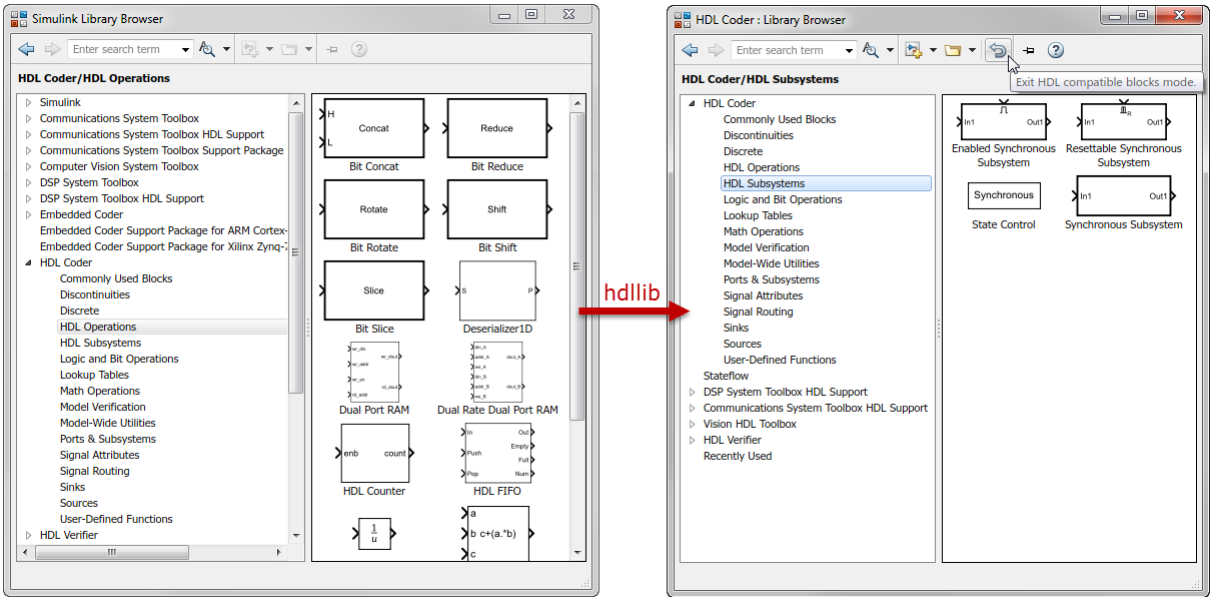
In the **Simulink Library Browser > HDL Coder** library, you can find Simulink blocks that are compatible with HDL code generation. In many cases, the blocks are also pre-configured with HDL-friendly settings, compared to the same blocks in the regular Simulink library.

- The 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.
- Sub-libraries such as **HDL Coder > HDL RAMs** and **HDL Subsystems** provide blocks specific to HDL applications – e.g. RAMs, and subsystems with synchronous enable/reset control inputs.

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:

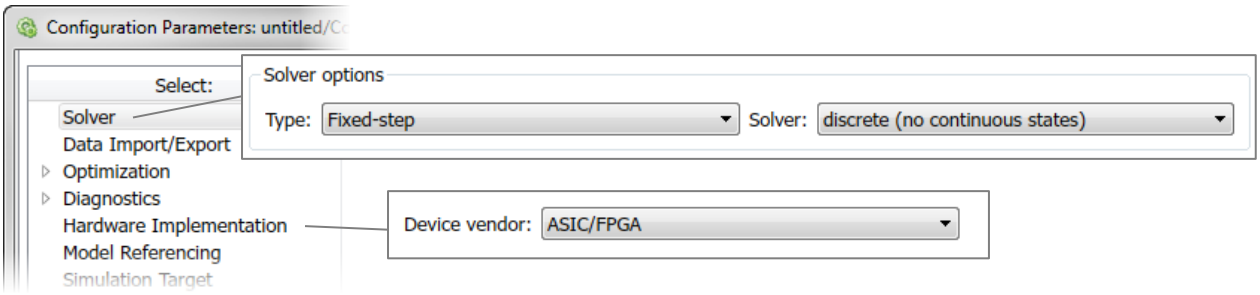
- DSP System Toolbox HDL Support
- Communications Toolbox HDL Support
- Vision HDL Toolbox
- LTE HDL Toolbox

Finally, you can use the command `hdllib` to display only HDL-supported blocks in the library browser.

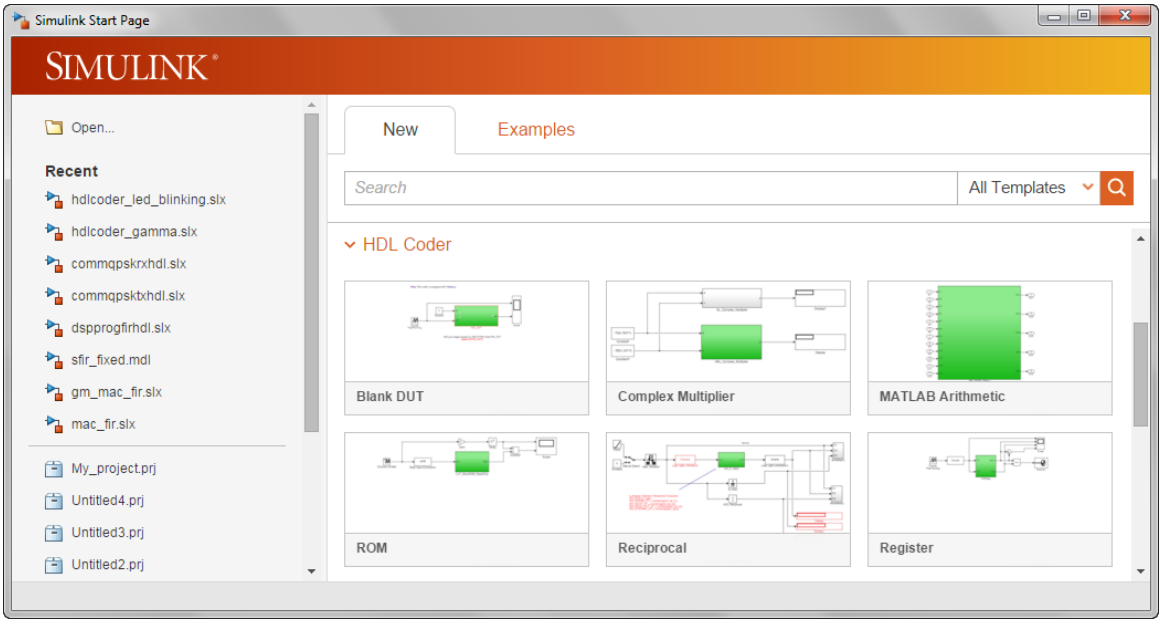


1.2 Model setup

When creating a new model, you can quickly configure its parameter settings with the command `hdlsetup('<your_model_name>')`. Many of these settings are recommended for HDL code generation, such as using a discrete, fixed-step solver, and setting hardware device vendor 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. You can examine all the settings applied using the command `edit hdlsetup.m`.



In addition, you can create a Simulink model using templates pre-configured for HDL code generation. On the **Simulink Start Page**, scroll down to the HDL Coder section to choose from a variety of starting points:



Finally, you can use the HDL Model Checker, described in section 1.5, to check the settings on your existing models.

1.3 DUT and test bench partitioning

Define a subsystem as the Design-Under-Test (DUT), containing all the elements to be implemented on your target ASIC or FPGA. This subsystem is usually at the top-level of the model, although it may also be at a lower level.

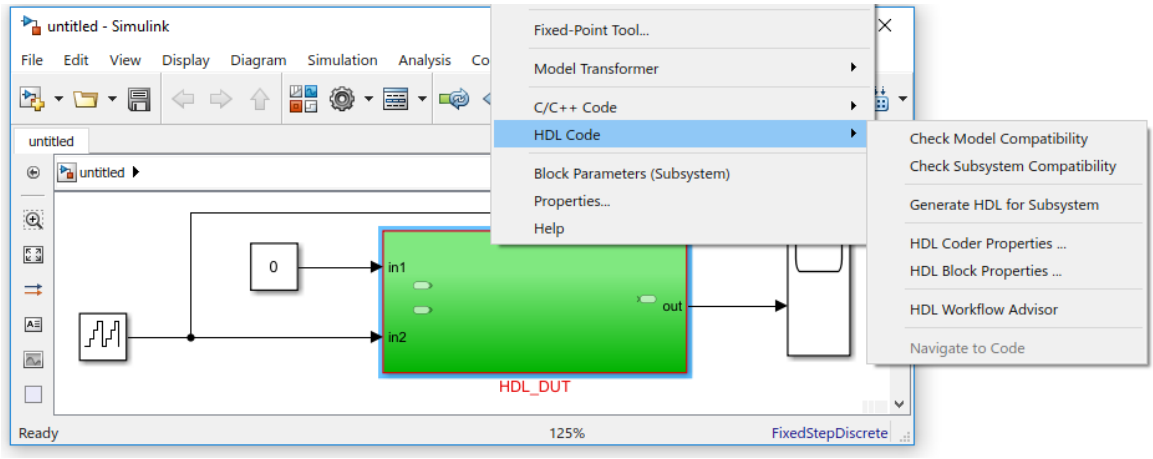
- All blocks outside of the DUT are part of the test bench. You can use any blocks in the test bench, including blocks not supported for HDL code generation.
    - When generating RTL for the test bench, code is not generated for the blocks in the test bench; only the input and output values of the DUT are logged.
    - Test bench generation is disabled if you designate the entire model as DUT.

  Inside the DUT, you can further partition your design into subsystems based on functionality, sample rate, power architecture, division of labor, etc. A separate HDL file is created for each subsystem by default.

1.4 Accessing HDL settings and operations

You can access many HDL settings and operations using the context menu in Simulink. Right-click on a subsystem or block, and select the following under the **HDL Code** menu:

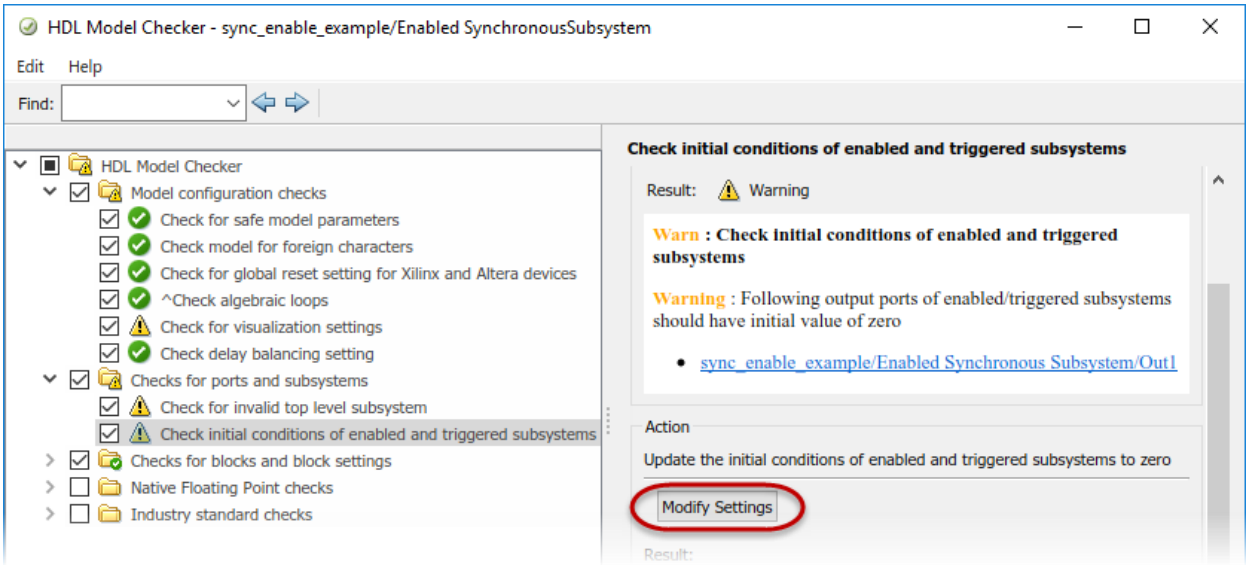
- **Check Model Compatibility**
  Analyze your model for HDL compatibility and efficiency using the HDL Model Checker – see 1.5 for details.
- **HDL Block Properties** and **HDL Coder Properties**
  Configure block and model-level HDL properties that take effect when you generate code.
- **Check Subsystem Compatibility** and **Generate HDL for Subsystem**
  Quickly check HDL compatibility and generate HDL code for a subsystem.
- **HDL Workflow Advisor**
  Launch the HDL Workflow Advisor to perform operations such as FPGA synthesis – see 3.1 for details.



1.5 HDL Model Checker

Beginning in R2017b, you can use the HDL Model Checker to check your model for HDL compatibility, and to ensure that your design follows recommended modeling practices, many of which are described in this document. You can often update your model with the recommended settings directly within the Model Checker UI.

To launch the HDL Model Checker, right-click on your DUT subsystem and select **HDL Code > Check Model Compatibility**. Consult the documentation for the list of checks in the HDL Model Checker.



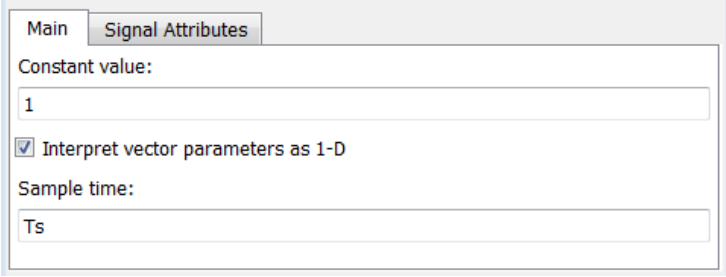
2. Using Simulink for HDL Designs

As you create your design in Simulink, it helps to keep your hardware implementation in mind. For instance, hardware requires one or more clocks, it usually requires fixed-point data and operations for efficiency, and design decisions can greatly impact the resulting hardware architecture, which affects resource usage and performance. Finally, there will be some functionality better suited for design using MATLAB Function or Stateflow blocks.

2.1 Concept: Sample time and clock

"How do I model the clock signal?" – is a question frequently asked by hardware engineers who are new to using Simulink. Here's how it works:

- 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.
- 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.
- Note: Some optimization settings (e.g. sharing factor) and alternative block architecture (e.g. Newton-Raphson square root) introduce additional sample rates not present in the original model. In those cases, the fastest generated sample time is mapped to 1 HDL clock.
- Tip: Define sample time and ratio using MATLAB variables (e.g. `Ts = 10e-9`, `upsamp = 4`). This makes it easy to change all sample time settings quickly.



2.2 Modeling best practices for generating efficient HDL

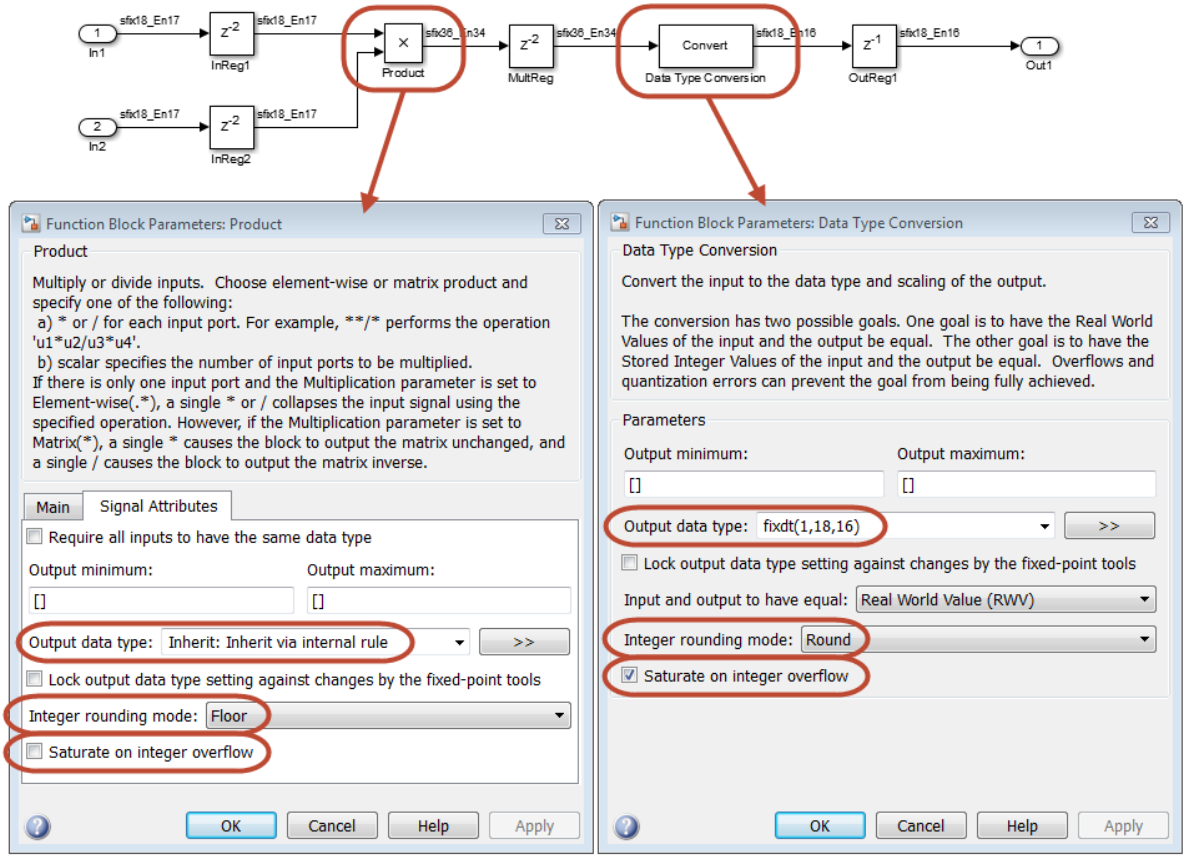
HDL Coder can generate synthesizable code for single and double-precision floating-point operations and data types. This is useful for high dynamic range calculations or to generate early prototype HDL. For more information on this feature, see this video. Best practices are listed in section 2.2.6.

2.2.1 Fixed-point settings

For most data types and operations, your design will consume fewer resources and cycles of latency if you convert to the smallest fixed-point data types that deliver sufficient accuracy for your needs. While this is an engineering effort in and of itself, the following tips will help you manage the effort in HDL Coder.

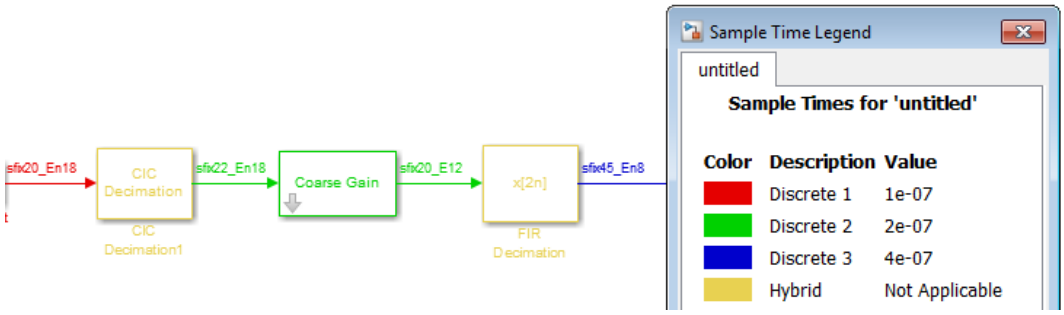
- Enable data type display on signals and ports (menu **Display > Signals & Ports**) to help visualize fixed-point settings and propagations.
- HDL code generation supports binary-point scaling only.
- Fixed-point modeling is supported for up to 128 bits.
- For 1-bit data, use `Boolean` for logical operations such as control signals, and `ufix1` for arithmetic operations. Performing math operations on Boolean data type can lead to unexpected results.
- The data type setting `Inherit: Inherit via internal rule` automatically calculates word and fraction lengths for full-precision fixed-point arithmetic. To ensure the calculations are optimized for HDL, set the model parameter **Hardware Implementation > Device vendor** to `ASIC/FPGA`.

- Rounding and saturation logic consume hardware resources and may become the critical path for making timing closure. If possible, set **Integer rounding mode** to `Floor` and turn off **Saturate on integer overflow** on blocks that provide those options.
- If your algorithm does require rounding and saturation, insert a pipeline register between the arithmetic operation and the rounding/saturation logic for better timing performance. You can do so explicitly like the following example, where the multiplication is full-precision, and the Data Type Conversion block rounds/saturates the output after a pipeline delay. Alternatively, HDL Coder can automate the pipeline insertion for you via Adaptive Pipelining. See section 3.3.4 for details.

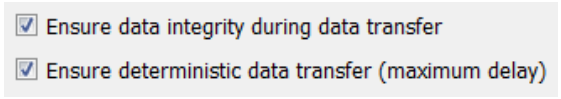


2.2.2 Multi-rate modeling

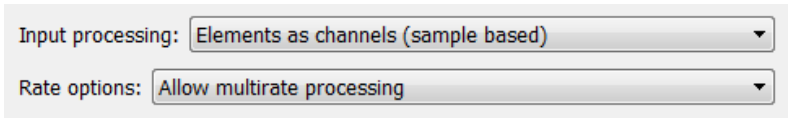
- Use sample time colors and sample time legend (menu **Display > Sample Time**) to help visualize different sample rates in the model. The fastest sample time is always annotated in red.



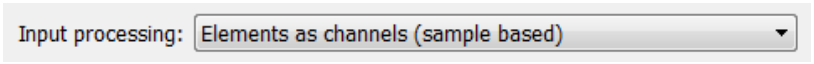
- Use only the following blocks and block settings to change sample rate:
  - Rate Transition block

  - Downsample block (DSP System Toolbox)

  - Upsample, Repeat blocks (DSP System Toolbox)

- Downsampling

  Example: downsample_upsample_example.slx

  - Downsampling using the Rate Transition block or a Downsample block with offset = 0 happens immediately – notice the output and input changes to 2 at the same time in the example below. This combinatorial path results in additional HDL bypass logic that may impact timing performance.

  - To generate more efficient HDL, place a delay after the Downsample/Rate Transition block. This removes the combinatorial path from the model, and allows HDL Coder to optimize away the bypass logic in the HDL. HDL Coder can also automate the delay insertion for you via Adaptive Pipelining – see section 3.3.4 for details.
  - For offset > 0, it is not necessary to add delay after the Downsample block. The generated HDL is already efficient.
- Upsampling

  Example: downsample_upsample_example.slx

  - The Upsample block contains a similar combinatorial path as the downsampling case described above. To generate more efficient HDL, use the Repeat block (which becomes a wire in the HDL) or the Rate Transition block.

2.2.3 Conditional subsystems

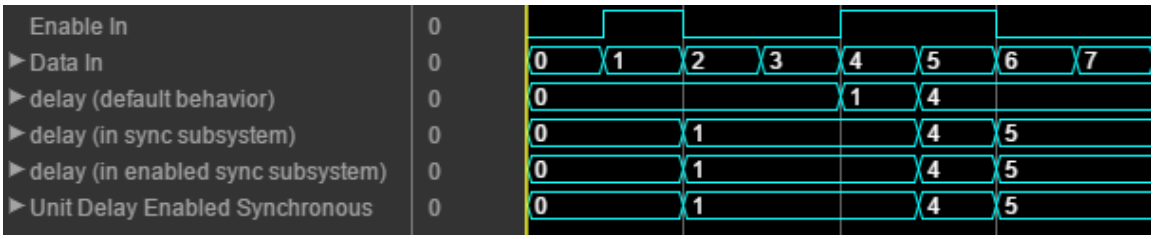
Example: enabled_subsystem_example.slx

- Enabled and triggered subsystems (using rising or falling trigger type) are supported for HDL code generation, but they must reside within a regular subsystem.
- Consider using enabled subsystems in *synchronous* mode to avoid the following manual steps. See the next section on synchronous enable and reset for more details.
  - Output ports in enabled and triggered subsystems must have **Initial output** set to 0 (The initial output is set to [] if you obtain the subsystem from the main Simulink library).
  - Enabled and triggered subsystems contain a similar combinatorial path as the downsampling case described in the previous section. To generate more efficient HDL, place a delay at the output of these subsystems. HDL Coder will automatically optimize away the bypass logic in the HDL.

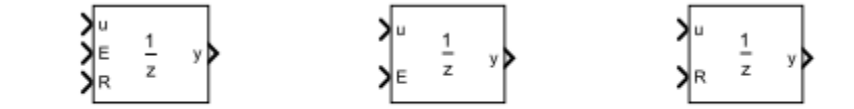
2.2.4 Synchronous enable and reset

Example: sync_enable_example.slx

Local synchronous enable and reset signals are often used in hardware designs for control path logic, power management, etc. Depending on your design needs, you can model them in Simulink in a few different ways.



- Blocks such as Delay and Discrete FIR Filter provide optional enable and reset ports. However, these ports do not act synchronously by default. To model synchronous enable/reset, place a State Control block in the subsystem (at the same level or higher), and configure it as `Synchronous`.
- To model synchronous enable/reset for multiple blocks in a subsystem without explicit enable/reset wiring, use the enabled or resettable subsystem with a synchronous State Control block.
- For a single register, you can use the following blocks in the **HDL Coder > Discrete** library. No State Control block is necessary for these blocks; it already exists under the hood.

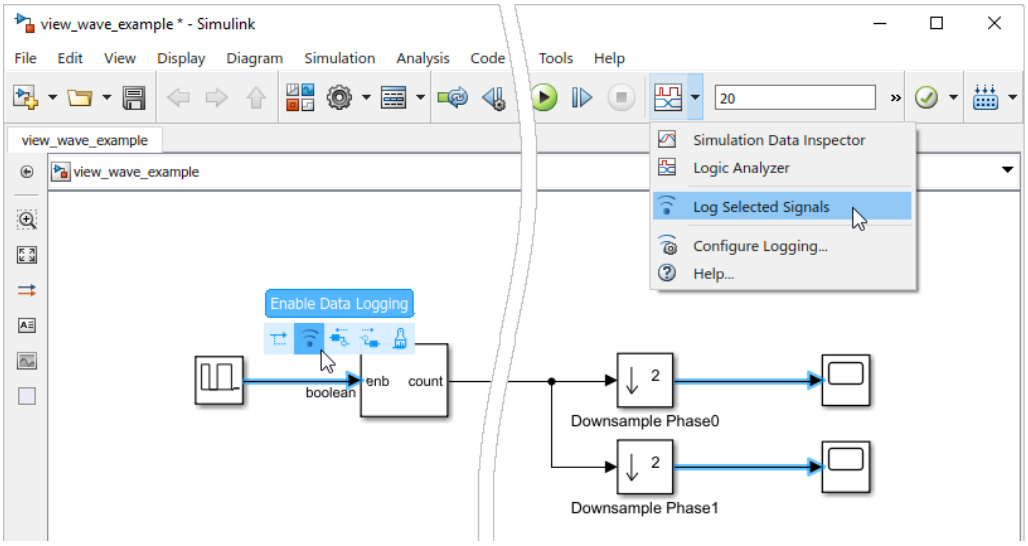


  Tip: If your design contains the Unit Delay Enabled block from older MATLAB releases, consider replacing it with the Unit Delay Enabled Synchronous block. The older version does not work in a synchronous subsystem, and is no longer available in the Simulink library browser.
- The State Control block and pre-configured synchronous subsystems can be found in the **HDL Coder > HDL Subsystems** library. For more information, consult the documentation Synchronous Subsystem Behavior with the State Control Block.

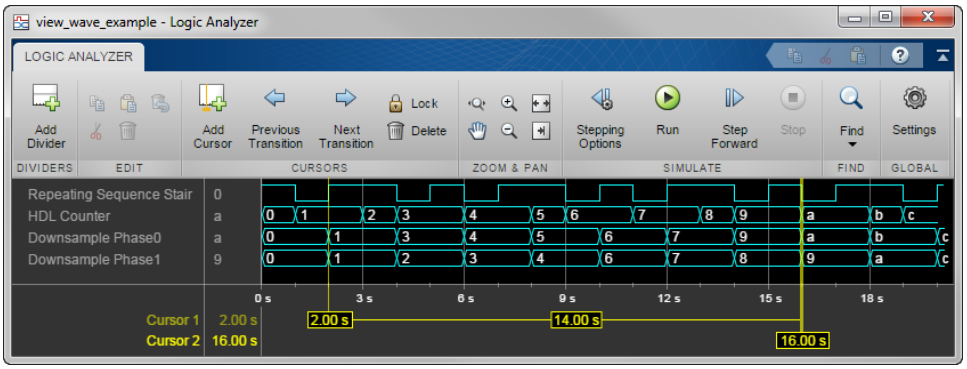
2.2.5 Logic analyzer scope

Example: view_wave_example.slx

The Logic Analyzer scope from DSP System Toolbox allows you to easily view Simulink signals as digital waveforms. To use the Logic Analyzer scope, select and **log** one or more signals using the model toolbar or pop-up cue:



After simulating the model, click the Logic Analyzer button  to view the waveforms.



2.2.6 Native floating-point code generation

- Use blocks from the **HDL Coder > HDL Floating Point Operations** library

  The library contains Simulink blocks that are configured for HDL code generation in native floating-point mode.

- Mix fixed- and floating-point types

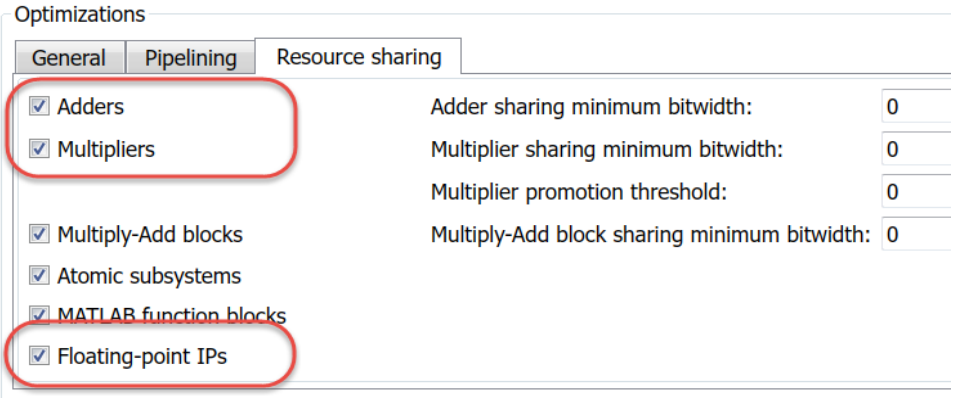
  Floating-point designs provide better precision and higher dynamic range than fixed-point, but can potentially occupy more area on the target hardware. To reduce area usage, consider using floating-point for only part of the model. For example, it is always good practice to keep the control logic in fixed-point but the data path requiring high dynamic range in floating-point. Use Data Type Conversion blocks between the floating- and fixed-point sections.

- Customize latency

  Native floating-point operators insert pipeline registers in the generated code to optimize hardware performance. You can customize the latency strategy at the model or block level to tradeoff between latency and throughput. For more information, see the documentations Latency Considerations with Native Floating Point and Latency of Floating Point Operators.

- Use optimizations such as resource sharing

  Consider enabling optimizations to improve area and timing performance of your floating-point design. You can share floating-point adders, multipliers and other resources with the following settings, when used in conjunction with subsystem resource sharing.



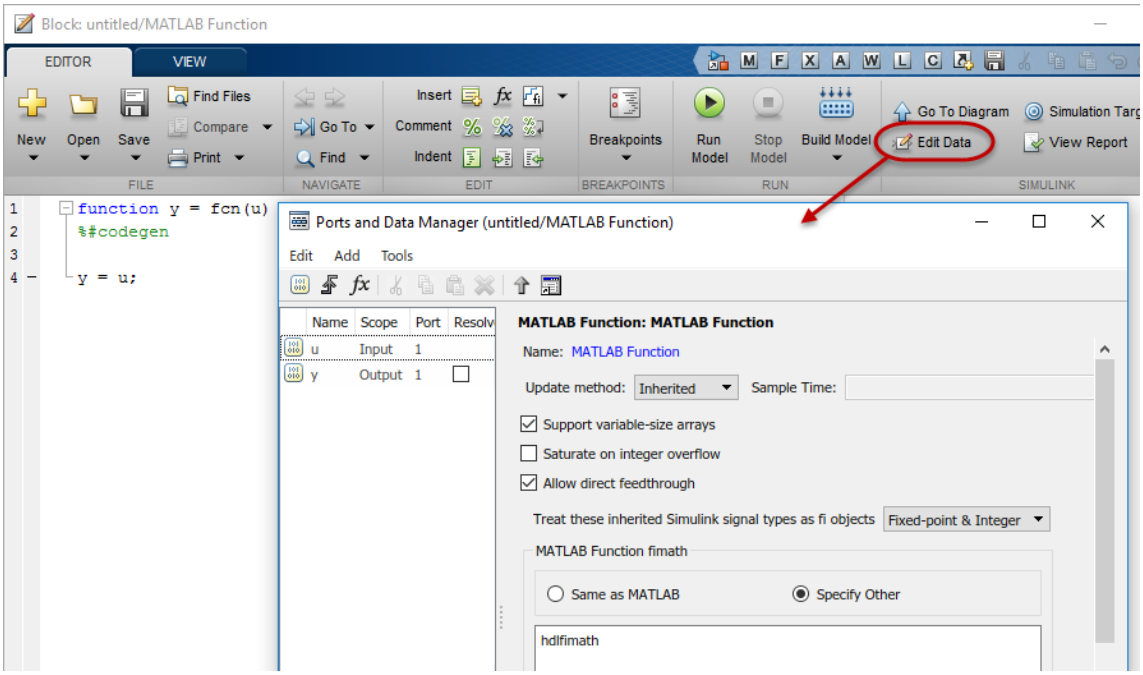
- For additional best practices, see the documentation Getting Started with HDL Coder Native Floating-Point Support.

2.3 Using a MATLAB Function block

When you need to create control logic that is not available in standard Simulink blocks, for instance an if-else mux or a simple finite state machine, you can use a MATLAB Function block embedded in the Simulink model. This is an easy way to create custom logic using MATLAB but integrated with your Simulink algorithm. To get started with this flow, place a MATLAB Function block from the library **HDL Coder > User-Defined Functions** into your design and double-click it to edit in MATLAB.

2.3.1 Block setup

- Use the `%#codegen` compilation directive below the function signature. While this has no impact on HDL code generation, it may result in better diagnostic messages during model compilation.
- To define block properties such as data types and fimath, and specify function input parameter, click the **Edit Data** button to open the **Ports and Data Manager**:



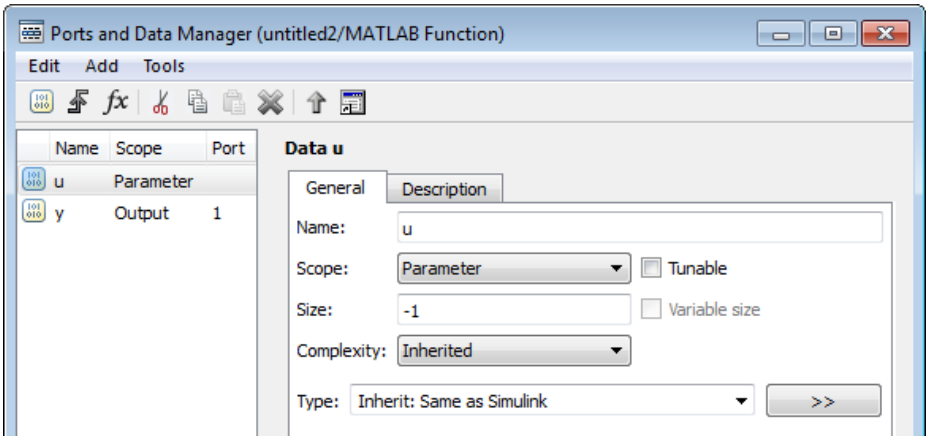
  - As discussed in the fixed-point section, rounding and saturation logic consume hardware resources and may impact timing performance. Unless they are required:
    - Turn off **Saturate on integer overflow**
    - Set **MATLAB Function fimath** to `hdlfimath`, which has the following properties:

```
>> hdlfimath
ans =

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

  - You may need to set **Treat these inherited Simulink signal types as fi objects** to `Fixed-point & Integer` in some cases, such as when performing bit operations on built-in data types like `uint16`.
  - Uncheck **Allow direct feedthrough** if a MATLAB Function block contains only registered logic. This allows you to use the block in a feedback loop and prevents algebraic loops. See the next section on how to model registers using MATLAB code.

- To configure a function input argument as a parameter (e.g. a MATLAB workspace variable) instead of an input to the block, highlight the input on the left, and change **Scope** to `Parameter`. Un-check the **Tunable** box unless you intend to turn the parameter into a top-level input port in the HDL code.



- Finally, consider partitioning large amounts of MATLAB code into multiple blocks and/or stand-alone MATLAB functions (.m) for ease of reuse and pipelining.

2.3.2 Modeling registers with MATLAB code

Registers can be modeled two ways in MATLAB:

1. Persistent variables. Follow these rules to properly infer registers from MATLAB:
   - Specify initial value and data type using the `isempty` function
   - Read from the variable before assigning a new value to it. Otherwise, the HDL output will either be obtained at the input of the register, or a register may not be generated at all. If your code only contains sequential logic, you can uncheck **Allow direct feedthrough** to enforce this rule.

```
persistent u_d;
if isempty(u_d)
    % defines initial value driven by unit delay at time step 0
    u_d = cast(0, 'like', u);
end                                    <- initialization

% return delayed input from last sample time hit
y = u_d;                               <- read current value

% store the current input
u_d = u;                               <- assign new value
```

2. `dsp.Delay`. This function from DSP System Toolbox is a straightforward way to create a register. However it cannot be used to model delay in a feedback loop.
3. `coder.hdl.pipeline`. This pragma allows you to specify the addition of pipeline registers at the output of an expression. The registers are added in the generated HDL, without changing the model simulation behavior. For instance to add two pipeline states at the output of an arithmetic operation:

```
y = coder.hdl.pipeline(A*D + B*C, 2);
```

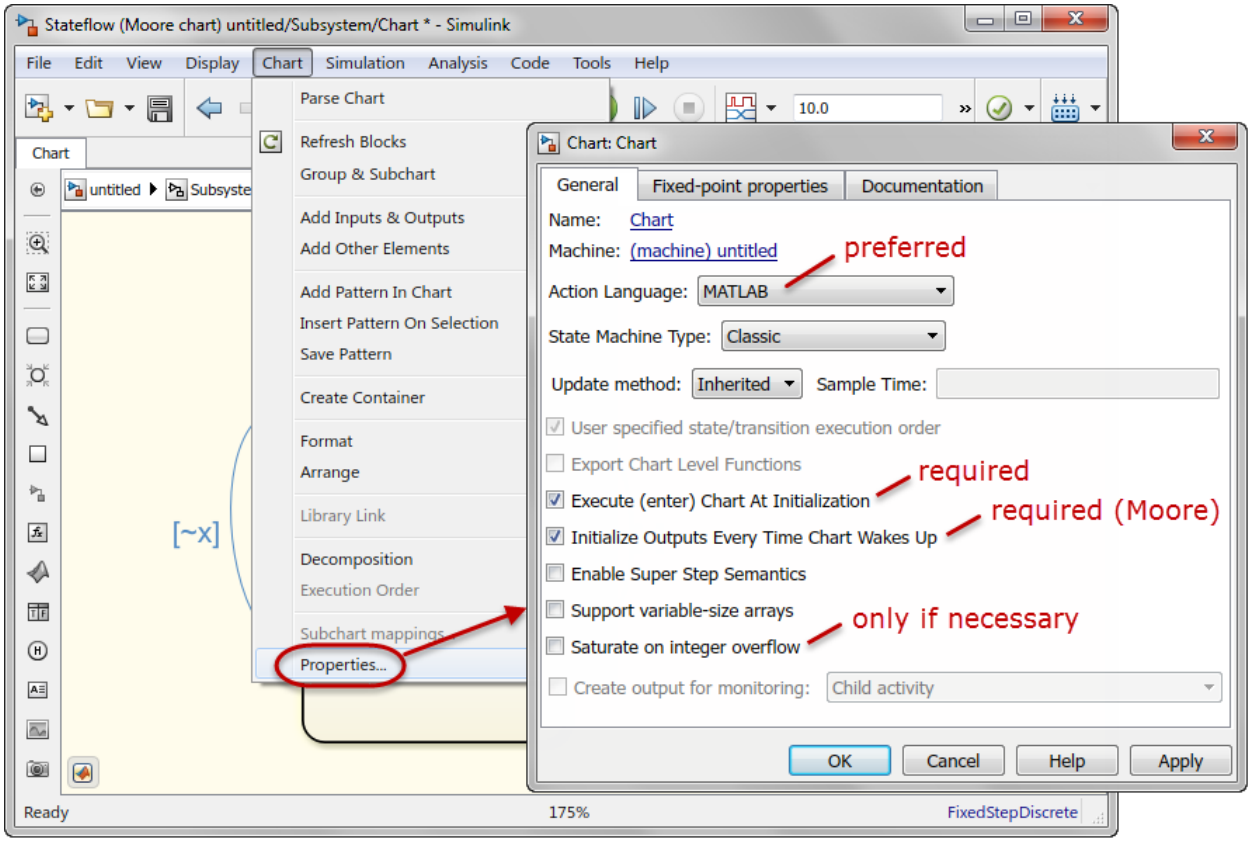
2.4 Using a Stateflow chart

Stateflow allows you to visually describe complex finite state machines (FSM). As designs become more complex, this is useful for managing all of the traffic between algorithms in a way that will be understandable throughout the product's lifecycle. Some example applications include:

- Interface logic between blocks (e.g. SPI)
- Standard connectivity protocols between different blocks at the link layer and above (e.g. PCI, proprietary protocols)
- Projects that need requirements traceability such as DO-254

2.4.1 Chart setup

Follow the guidelines below to set up your Stateflow chart for HDL code generation. To open the chart properties dialog box, select **Chart > Properties** on the Stateflow editor menu.

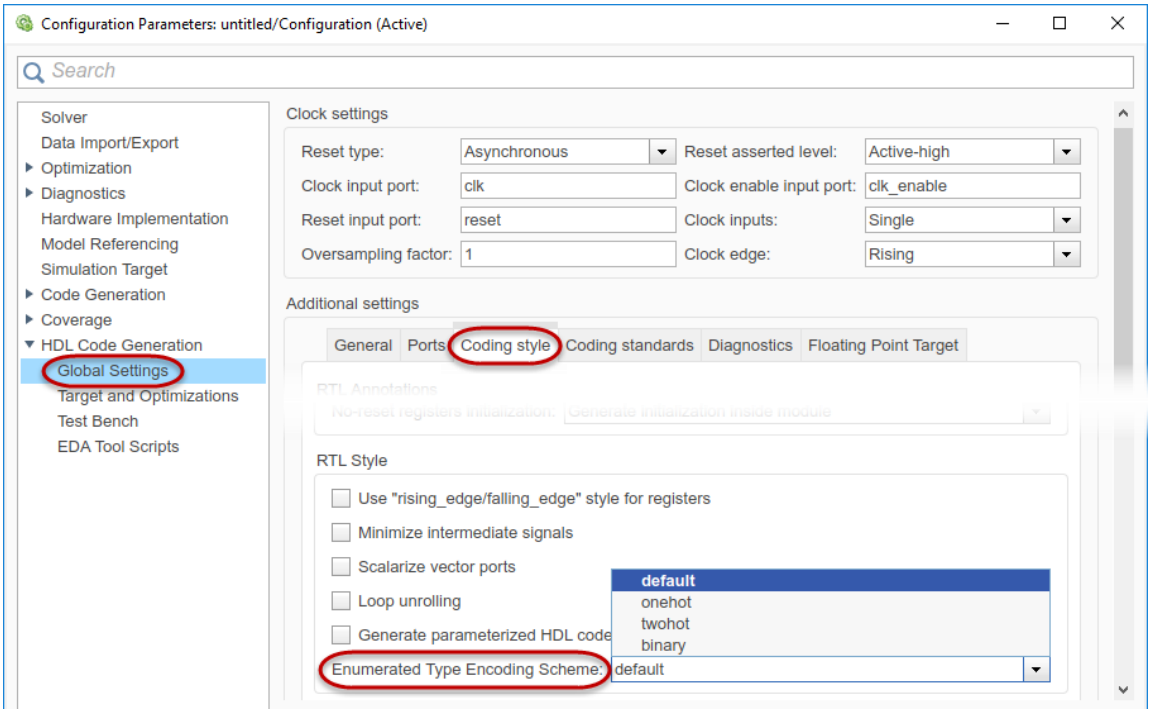


- Set **Action Language** to `MATLAB` (default). C action language is not recommended although it is allowed.
- Enable the option **Execute (enter) Chart At Initialization**.
- For Moore state machines, enable the option **Initialize Outputs Every Time Chart Wakes Up**.
- When used inside synchronous subsystems (described in section 2.2.4), **State Machine Type** must be set to `Moore`.
- As discussed in the fixed-point section, rounding and saturation logic consume hardware resources and may impact timing performance. Unless they are required:
    - Turn off **Saturate on integer overflow**
    - Set **MATLAB Chart fimath** (under the **Fixed-point properties** tab) to `hdlfimath`, which has the following properties:

```
>> hdlfimath
ans =

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

- You may need to set **Treat these inherited Simulink signal types as fi objects** (under the **Fixed-point properties** tab) to `Fixed-point & Integer` in some cases, such as when performing bit operations on built-in data types like `uint16`.
- To specify state machine encoding scheme such as one-hot, use the global parameter **Enumerated Type Encoding Scheme**. It can be found under the **Coding style** tab, on the **Global Settings** pane of the HDL Coder UI, or in HDL Workflow Advisor **3.1.3 Set Advanced Options**.

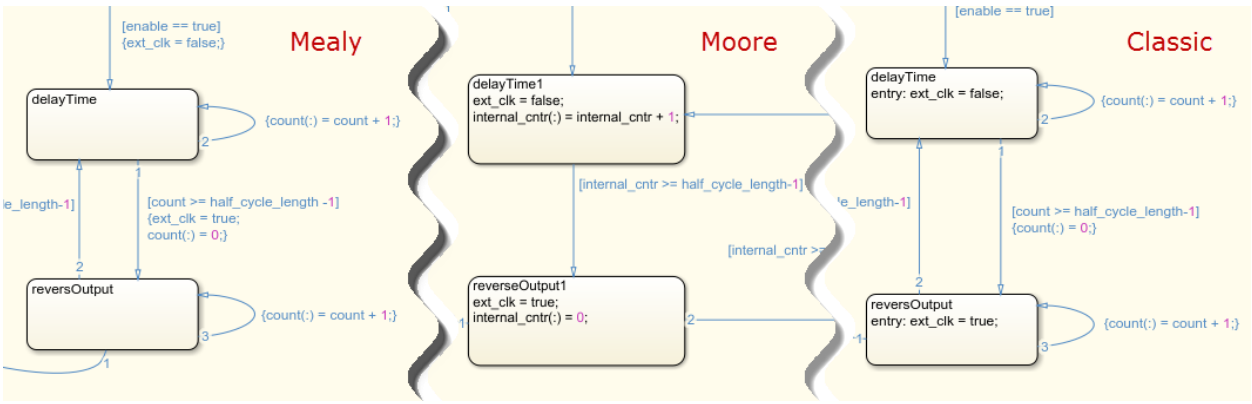


2.4.2 Mealy, Moore, and Classic charts

Example: stateflow_example.slx

You can configure the **State Machine Type** of a chart as `Mealy`, `Moore` or `Classic`. Stateflow enforces chart semantics according to the type of state machine selected.

- A *Mealy* state machine generates outputs as a function of the current state and inputs. Outputs are defined along state transitions only. It is often more flexible to use and consumes fewer hardware resources.
- A *Moore* state machine generates outputs based on the current state only. Outputs are defined in states only. It is typically easier to read, but consume more hardware resources to perform a given set of operations.
- A *Classic* state machine is a combination of Mealy and Moore machines. You can use both Mealy and Moore semantics in a classic chart.



2.4.3 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:

- Minimize unnecessary transition conditions and redundant logic.
- Implement resource-intensive logic such as large comparators and arithmetic operations outside of the Stateflow chart, and interact with those logic using simple control signals. This allows resources to be easily shared, improving both area and timing performance.
- Use a default transition, which will generate HDL with an “else” clause for an if/else statement or a “default” clause for a case statement. This ensures that unintended latches will not be created for undefined state transitions.
- For Mealy and Classic charts, make sure inputs are properly registered to prevent long combinatorial data paths.

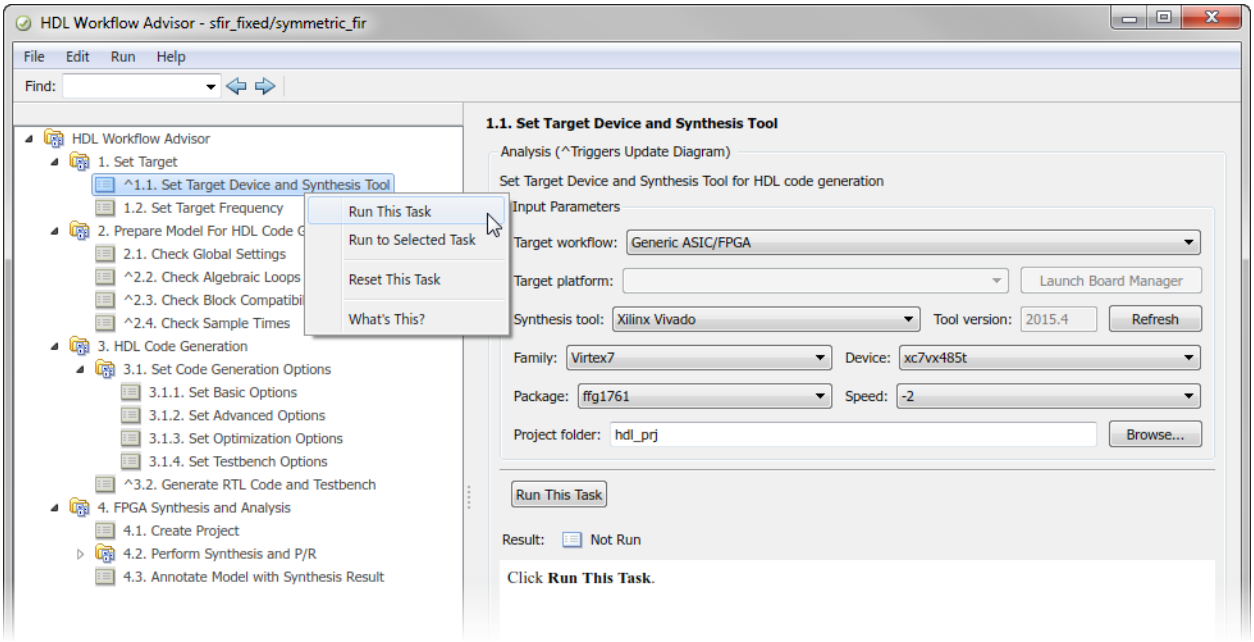
3 Code Generation and Verification Tools

Once you feel your design is ready to implement in HDL, the next step is to generate the HDL code, analyze it to determine whether it meets your goals, make adjustments as necessary, and verify its functional equivalence to your system-level model.

3.1 HDL Workflow Advisor

HDL Coder provides a wizard called the HDL Workflow Advisor to guide you through design compatibility checks and HDL code generation. The guided workflow also integrates with third-party synthesis tools, allowing you to perform downstream operations such as:

- Synthesize your design without launching the synthesis tool manually
- Verify your design on hardware using FPGA-in-the-Loop
- Deploy your design on hardware for real-time operations such as software-defined radio (SDR) prototyping



Follow these steps to start using HDL Workflow Advisor:

- First, use `hdlsetuptoolpath` to add paths for Xilinx® ISE, Vivado®, Intel® Quartus® Prime or Microsemi® Libero® SoC to MATLAB. Run the command once per MATLAB session, or add it to your startup.m to have it run automatically when MATLAB starts. For example:

  ```
  hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath',...
  'C:\Xilinx\Vivado\2017.4\bin');

  hdlsetuptoolpath('ToolName', 'Altera Quartus II', 'ToolPath',...
  'C:\intelFPGA\17.1\quartus\bin64');

  hdlsetuptoolpath('ToolName', 'Microsemi Libero SoC', 'ToolPath',...
  'C:\Microsemi\Libero_SoC_v11.8\Designer\bin\libero.exe')
  ```

- Launch HDL Workflow Advisor by right-clicking on the DUT subsystem, and then select **HDL Code > HDL Workflow Advisor**.
- In **1.1 Set Target Device and Synthesis Tool**, set your desired workflow, hardware target and synthesis tool. For design iterations through synthesis, use the default `Generic ASIC/FPGA` in **Target workflow**.

Note: when a synthesis tool is selected, latency may be introduced to the generated design due to Adaptive Pipelining. See section 3.3.4 for details.

- Subsequent steps are updated according to the workflow selected. Run the steps sequentially by clicking **Run This Task**, or run several steps at once by right-clicking on a particular step and click **Run All** or **Run to Selected Task**.

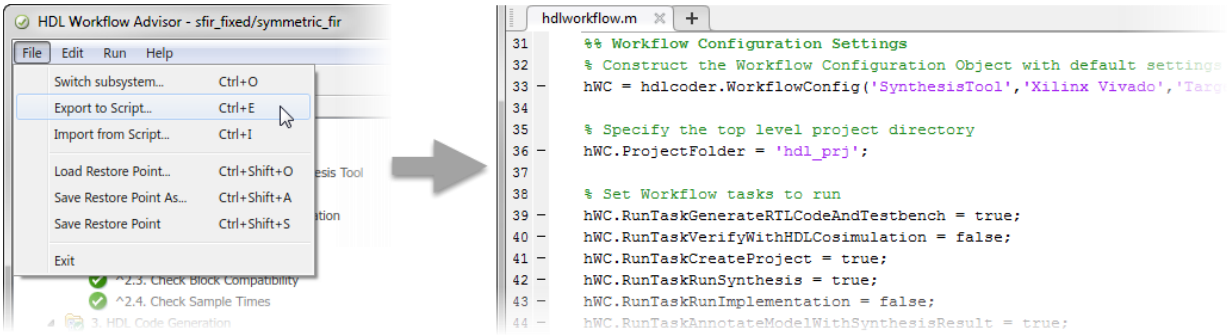
3.2 Scripting

Simulink and HDL Coder provide a powerful programmatic interface to support a script-based workflow. You can query and configure settings, generate code, and even construct Simulink diagrams using programmatic commands. For example:

- To get/set Simulink block and model properties, use `get_param` and `set_param`. For HDL properties, use the corresponding `hdlget_param` and `hdlset_param` commands.
- Export all non-default HDL property settings to a script using `hdlsaveparams`.
- Use `makehdl` and `makehdltb` to generate HDL for the DUT and testbench respectively.
- Tip: HDL properties can be applied to the model using `hdlset_param`, or passed to `makehdl` as arguments to affect current code generation outputs without modifying the model.

For more information on using a command, type `help <command name>` or `doc <command name>`.

Actions and settings in HDL Workflow Advisor can be similarly scripted. To get started, select **File > Export to Script** to generate programmatic commands for the current workflow.



3.3 A primer to optimizations

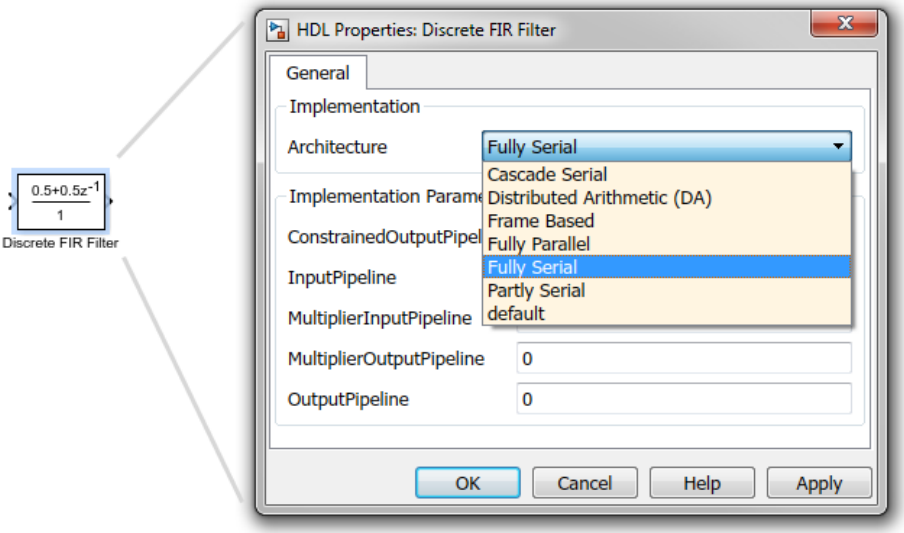
Automated speed and area optimization options provided by HDL Coder allow you to abstract low-level implementation details from your algorithm model. This section describes the behavior and benefit of commonly used optimization features, some of which are enabled by default.

3.3.1 General tips

- HDL optimization settings do not change simulation behavior in Simulink, but they may introduce differences in latency and/or data rate between your Simulink model and the generated HDL. Alternative block architectures may also introduce numerical changes. See section 3.4 on tools that are available to help you examine those differences and review optimization results.
- Blocks such as the FFT HDL Optimized block in DSP System Toolbox are specifically created for HDL code generation. The generated HDL from these “HDL optimized” blocks are bit- and cycle-accurate to their simulation behavior in Simulink.
- Change the **Sample time** setting for Constant blocks to `-1`. The default value `inf` may inhibit optimizations.

3.3.2 Block-level optimizations

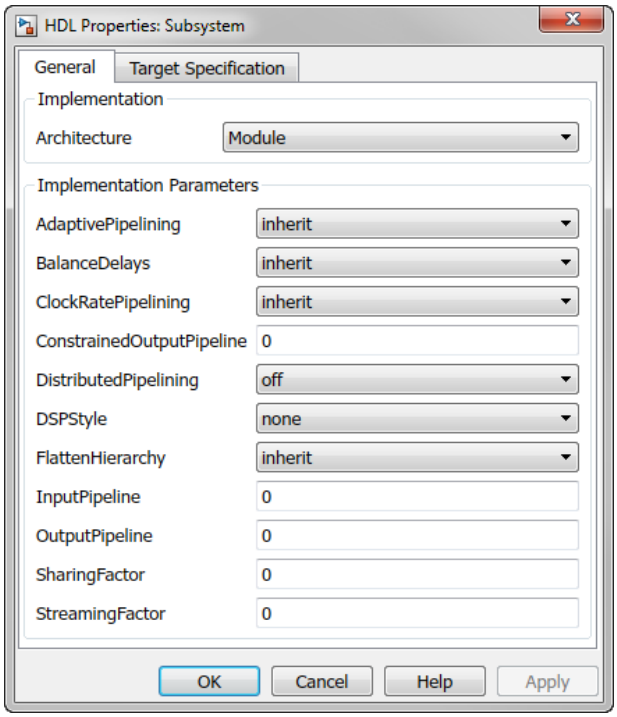
The default HDL implementation of a Simulink block is typically bit- and cycle-accurate to its simulation behavior. You can select alternative implementation options that best meet your desired hardware performance criteria by right-clicking on the block, and choosing **HDL Code > HDL Block Properties**.



- Many blocks offer multiple architectures for different speed/area requirements. For example, the Discrete FIR block uses a Fully Parallel architecture by default, but you can select a Serial architecture to create an area-efficient filter, or a Frame Based architecture to process multiple input samples in a single clock cycle.
- Some parameters control the hardware implementation of a block. For example, you can implement a constant gain as shifts and adds by setting **ConstMultiplierOptimization** to `csd` or `fcsd`, or map delays to RAMs using the **UseRAM** parameter.
- Pipelining parameters such as **OutputPipeline** allow you to specify pipelining in the HDL without affecting the original design in Simulink. Matching delays are automatically inserted to parallel data paths via Delay Balancing, as discussed in section 3.3.4 below.

3.3.3 Subsystem-level optimizations

A subsystem creates a hierarchy in the HDL and many opportunities for optimizations. Subsystem-level optimizations can be accessed in a similar fashion as block-level ones, using the HDL block properties dialog on the subsystem.



- Area optimizations

  Hardware resources can be reused among equivalent operations when oversampling is allowed. For example, a subsystem containing 100 18x18-bit multiplications at a 1MHz sample rate can be implemented using 5 18x18-bit multipliers running at 20MHz. You can specify this oversampling ratio using the **SharingFactor** or **StreamingFactor** parameter (for independent operators and vectors, respectively).

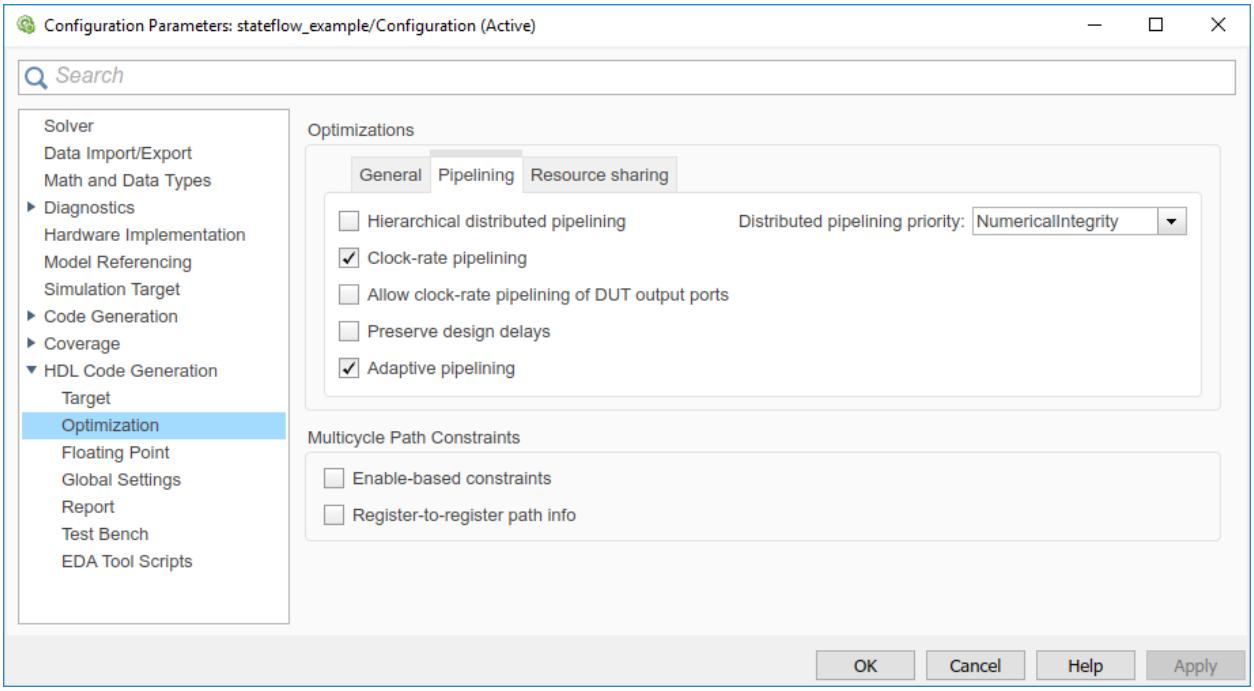
- Speed optimizations

  DistributedPipelining reduces the critical path of a subsystem by retiming registers specified in **InputPipeline**, **OutputPipeline**, as well as existing delay blocks in the subsystem. **FlattenHierarchy** removes the hierarchical boundaries within a subsystem, allowing HDL Coder to achieve better retiming results.

- Finally, you can override global or upper-level optimization settings for a subsystem by changing the value from `inherit` to `on` or `off`.

3.3.4 Model-level optimizations

Model-level optimizations are available in the **Optimization** pane of the HDL Code Generation section of Configuration Parameters, or in HDL Workflow Advisor step **3.1.4 Set Optimization Options**.



- Automatic delay balancing
  - When an optimization setting introduces a delay to a data path during code generation, the Delay Balancing feature automatically inserts matching delays to parallel data paths, so that the generated HDL produces the same numeric results as your original Simulink model, but with the parallel paths aligned at a later time step (clock cycle).
  - **Balance delays** is enabled by default; HDL Coder generates an error if Delay Balancing is unsuccessful. The condition was reported as a warning prior to R2016b.
- Adaptive pipelining
  - Some Simulink blocks require manual pipelining to create an efficient FPGA implementation. For example, a Delay block must be inserted manually between a Product block and the output rounding logic (section 2.2.1) or a Delay block inserted after a Downsample block avoids extra combinatorial logic in the HDL (section 2.2.2). Beginning in R2016b, HDL Coder can automate this

pipelining for you via the Adaptive Pipelining feature, when you are targeting a Xilinx or Intel FPGA device.

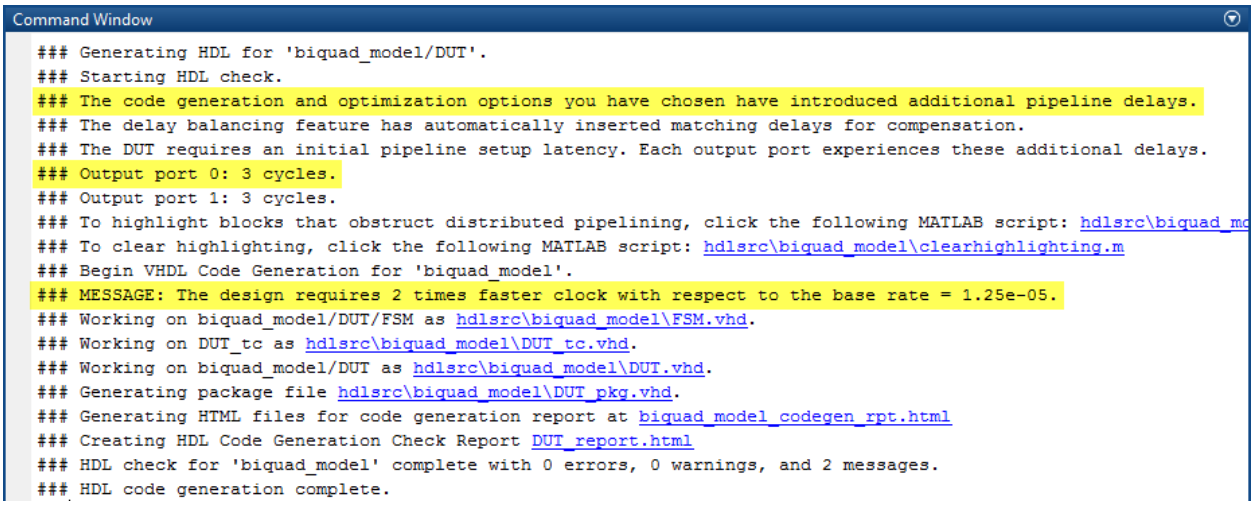
- Adaptive Pipelining automatically inserts pipeline registers in the generated HDL for lookup tables, multipliers and rate change blocks. For lookup tables, a register without reset is inserted to facilitate the mapping of RAM blocks. For multipliers, the number and placement of registers are automatically determined based on the synthesis tool, target frequency and word length settings. If rounding or saturation is selected, the register(s) are inserted before the rounding/saturation logic to ensure mapping of DSP block internal pipeline registers.
- **Adaptive pipelining** is enabled by default, but automatic pipeline insertion only takes place when **Synthesis Tool** is set. Multiplier output registers are only inserted when both **Synthesis Tool** is selected and **Target Frequency** is specified (>0). Adaptive Pipelining can be enabled or disabled at the subsystem or model level.
- You can generate an optimization report to review the result of Adaptive Pipelining. You can also visualize and simulate the additional registers in your design using the generated model. See the next section on how to create reports and generated models.

3.4 Reviewing code generation results

Along with RTL code, HDL Coder provides a number of artifacts to help you understand the code generation results. This section describes the available artifacts and how to enable them.

3.4.1 Code generation messages

During code generation, status messages are printed to the MATLAB Command Window or the status window in HDL Workflow Advisor:

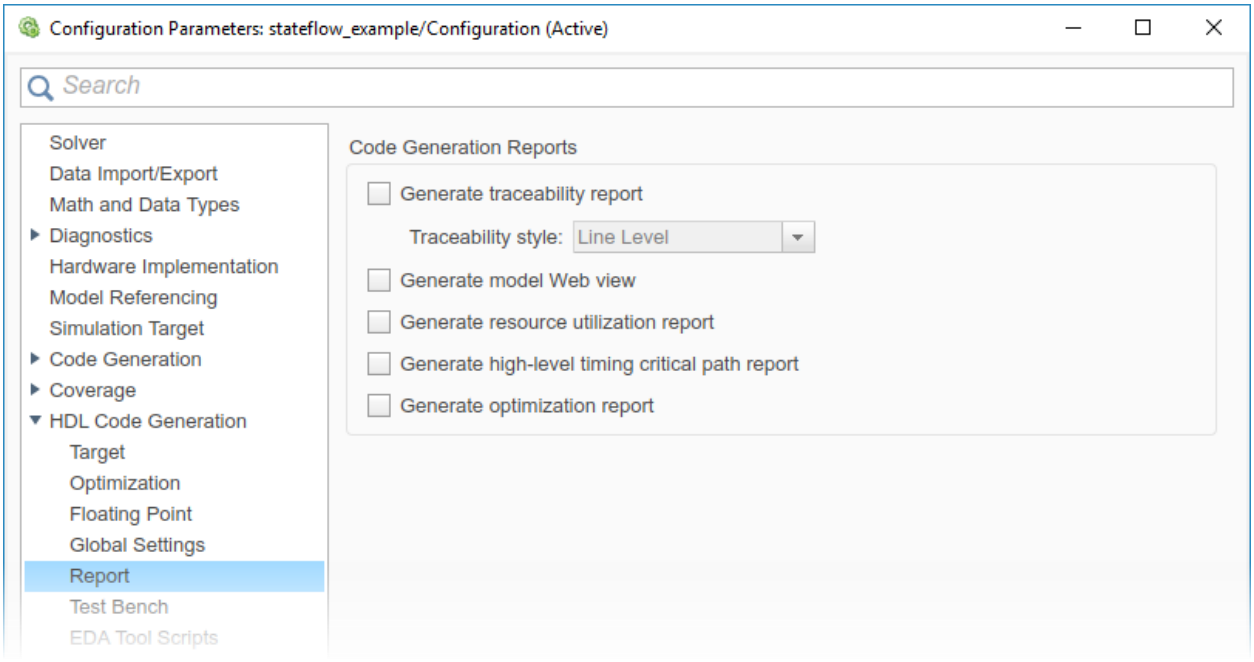


- As described in the previous section, some optimization settings may introduce latency, data rate and numeric changes in the HDL. Any difference in latency and data rates between the generated HDL and your original Simulink model are reported in the status messages, as shown in the example above.
- HDL Coder provides diagnostic scripts to highlight in your model what prevents optimizations from being applied. Links to these scripts are printed when the requested optimizations (e.g. Delay Balancing) are unsuccessful.
- A summary of the code generation result is printed at the end, along with a link to the detailed html check report. Make sure to click on the report and review any code generation warnings or messages. The report is automatically brought up if code generation resulted in any error.

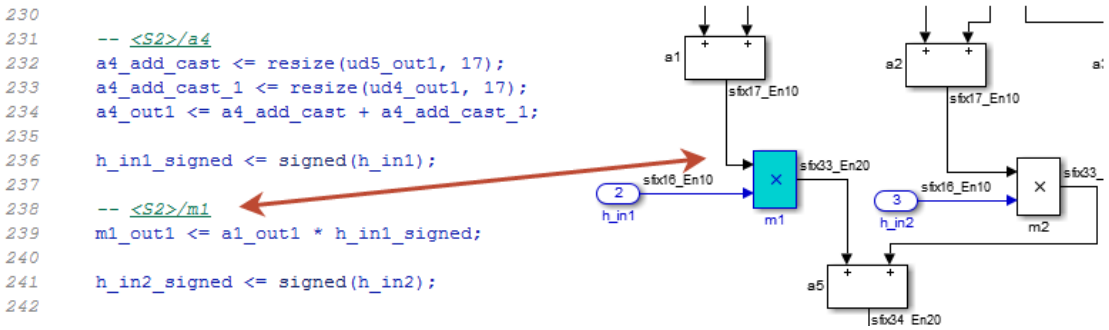
```
### Creating HDL Code Generation Check Report DUT_report.html
### HDL check for 'biquad_model' complete with 0 errors, 0 warnings, and 2 messages.
```

3.4.2 Optional reports

You can create a number of reports to examine details of the generated design. A summary of all non-default block and model-level HDL property settings is also included with the generation of any of these reports. Select the optional reports in the model Configuration Parameters under **HDL Code Generation > Report**, or in HDL Workflow Advisor **3.1.2 Set Report Options**.



- *Traceability report* links a Simulink block to its generated RTL code, and vice versa:



- *Resource utilization report* provides a high-level usage summary for resources such as multipliers, registers and RAMs. While the report is not specific to your target-device (e.g. 10 18x30 bit multiplier instead of the number of DSP48 on a Xilinx device), it provides a quick resource estimation without performing a full synthesis run, which is especially helpful for exploring the resource impact of different design choices.

Generic Resource Report for biquad_model

Summary

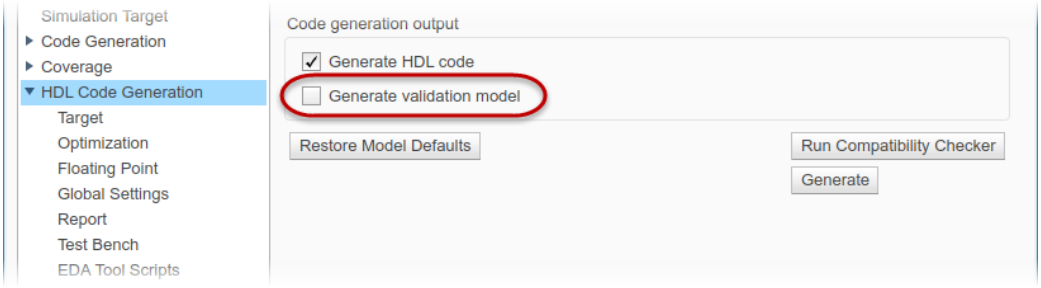
Multipliers	1
Adders/Subtractors	2
Registers	20
Total 1-Bit Registers	269
RAMs	0
Multiplexers	11
I/O Bits	62
Static Shift operators	0
Dynamic Shift operators	0

- *Optimization report* summarizes the results of various optimization settings such as resource sharing, distributed pipelining, delay balancing and adaptive pipelining.
- *High-level timing critical path report* provides estimated critical path based on characterized device information, without running synthesis.

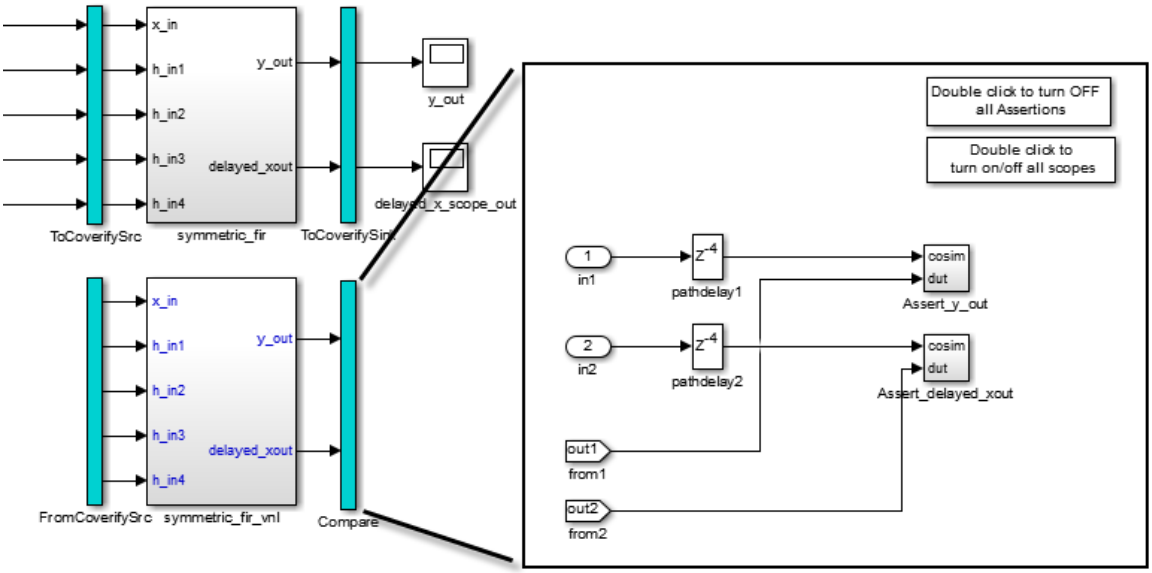
3.4.3 Generated and validation models

The generated model and validation model are tools that can help you analyze changes in the generated design due to optimization settings, and ensure the generated HDL is functionally equivalent to your original Simulink model.

- Generated model
  - A *generated model* is always produced and loaded in memory when you generate code from a Simulink model. It is a behavioral model that reflects the latency and numeric changes that resulted from the HDL architecture or optimization settings you have chosen, and it is bit- and cycle-accurate to the generated HDL.
  - When you generate an HDL testbench, the generated model is simulated to log the input and output values of the DUT. Those values are then used to compare to the HDL output produced in an HDL simulation.
  - The generated model uses your model name prefixed with `gm_`. For example, the example model `sfir_fixed` produces the generated model `gm_sfir_fixed`.
- Validation model
  - Since the generated model is equivalent to the generated HDL, you can compare it with your original Simulink model and analyze any latency or numeric differences between them. This comparison is automated for you in the *validation model*, which can be enabled in the HDL Coder section of the Configuration Parameters, or in HDL Workflow Advisor **3.2 Generate RTL Code and Testbench**.



  - A validation model combines your original Simulink model and the generated model in one place, and compares the outputs between the two models. Latency differences are compensated before the comparison, while numeric differences will trigger an assertion. The comparison logic is implemented in the Compare subsystem as shown in the example below.



- A link to the validation model will be printed as part of the code generation status messages. It can also be located in the optimization report if you choose to generate one.

```
### Generating HDL for 'sfir_fixed/symmetric_fir'.
### Starting HDL check.
### Generating new validation model: gm_sfir_fixed_vnl.
### Validation model generation complete.
```

3.5 Verifying generated code

Most likely you will iterate between analyzing results and adjusting your optimization settings in order to meet your goals. Once you are satisfied with the quality of results targeting your intended device, you can move to functional verification. There are three main methods:

- Generate an RTL testbench

  As mentioned before, the generated model is simulated to log the input and output values of the DUT during testbench generation. When you run the testbench in an HDL simulator, the HDL outputs are compared to the logged outputs from Simulink, and a PASS or FAILED message will be printed indicating whether the HDL and Simulink outputs are a match. For large testbench, consider generating a SystemVerilog DPI test bench in conjunction with Simulink Coder.

- Co-simulation

  A co-simulation model automatically compares the output between the generated model and the generated HDL that is being simulated in an HDL simulator. Any mismatch between the generated model and HDL outputs will trigger an assertion. You can generate a co-simulation model in the **Test Bench** pane of the HDL Coder UI, or in HDL Workflow Advisor **3.2 Generate RTL Code and Testbench**.

  In addition to co-simulating generated code, you can also incorporate hand-written HDL into your Simulink model using the Cosimulation Wizard. Note that co-simulation requires HDL Verifier and a supported HDL simulator.

- FPGA-in-the-Loop (FIL)

  HDL Verifier also provides the capability to co-simulate with FPGA hardware. A FIL model automatically compares the output between the generated model, and the compiled HDL that is running on a compatible FPGA board. Same as co-simulation, any mismatch between the generated model and FPGA outputs will trigger an assertion. To generate a FIL model, set **Target workflow** to `FPGA-in-the-Loop` in HDL Workflow Advisor, **1.1 Set Target Device and Synthesis Tool**.

  Note that the FPGA runs synchronously with Simulink during a FIL simulation – the FPGA is enabled for a clock period whenever the Simulink model advances a time step. To capture free-running FPGA signals, consider using the FPGA data capture workflow instead.

4 Targeting FPGA Hardware

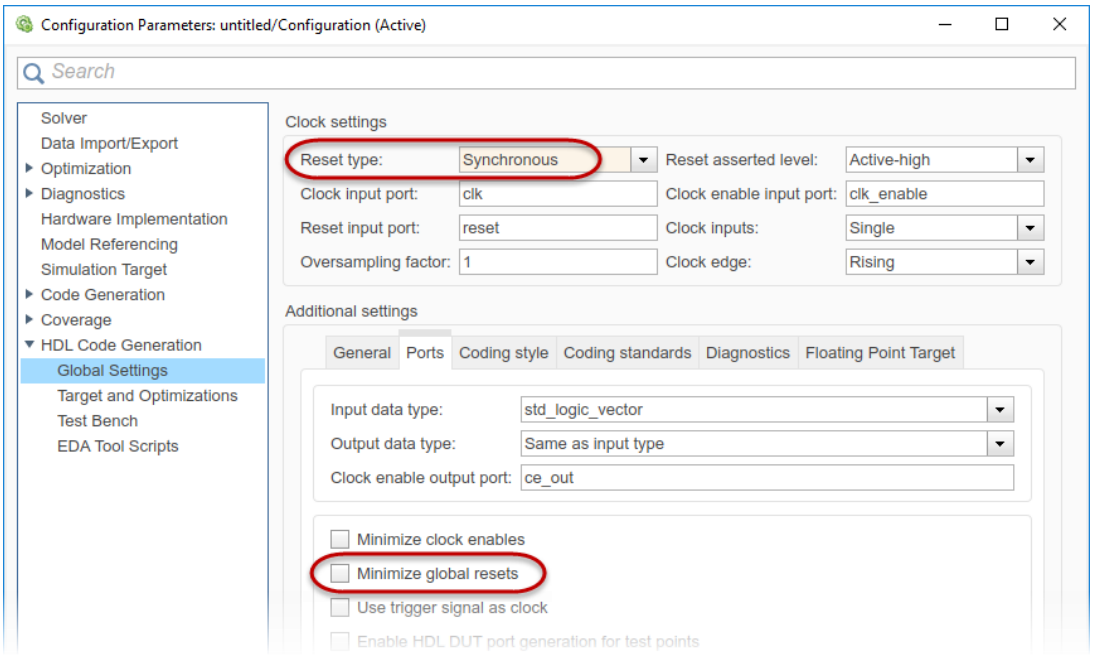
Understanding how your design maps to hardware, specifically your target device, is key to generating HDL that will meet your needs when implemented downstream. This section contains some tips that apply to common FPGA target devices.

4.1 General techniques

4.1.1 Global reset type

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.

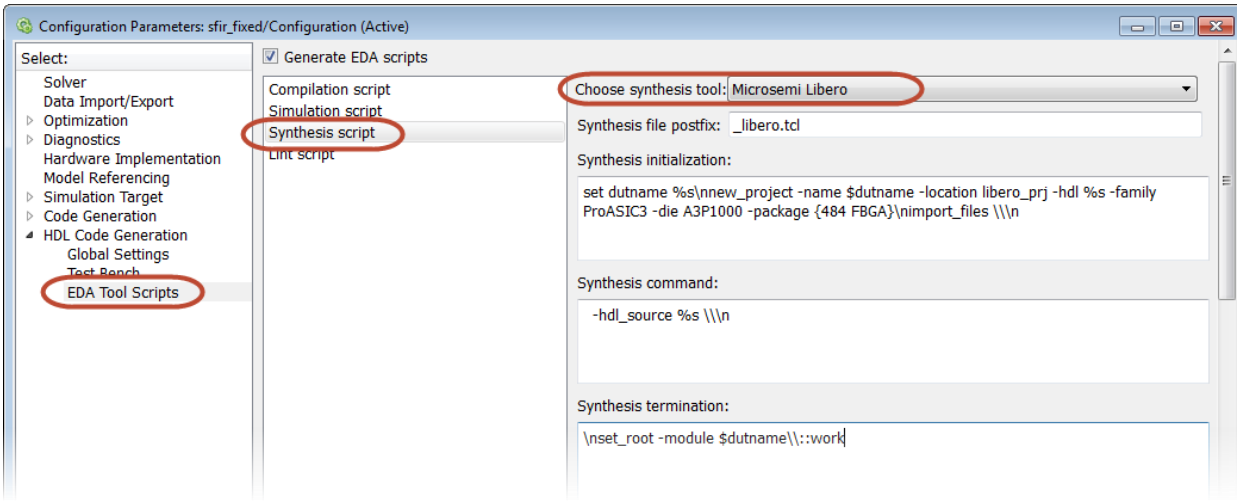
- For Xilinx FPGA devices, use `synchronous` global reset.
- For Intel FPGA devices, use `asynchronous` global reset.
- The **Reset type** setting can be found in the **Global Settings** pane of the HDL Coder UI, or in HDL Workflow Advisor **3.1.3 Set Advanced Options**.
- If your design does not require global reset, you can enable the **Minimize global resets** setting under **Additional settings > Ports**.



4.1.2 Synthesis tool integration

- If you use Xilinx ISE / Vivado, Intel Quartus Prime or Microsemi Libero SoC, you can create an FPGA project, synthesize your design, analyze timing and more using HDL Workflow Advisor, without having to launch the synthesis tool separately.
- To specify timing constraints or customize synthesis settings, supply a constraint or Tcl file in **4.1 Create Project**. If you specify a positive value in **1.2 Set Target Frequency**, a clock constraint is automatically generated and applied to the FPGA project.

  Note: when synthesis tool and target frequency are specified, latency may be introduced to the generated design due to Adaptive Pipelining. See section 3.3.4 for details.
- If your synthesis tool is not supported by HDL Workflow Advisor, you can generate a synthesis Tcl script that can be sourced to create an FPGA project. To enable the Tcl script generation and customize its contents, select the appropriate synthesis tool in **EDA Tool Scripts** > **Synthesis script** on the HDL Coder UI.



4.1.3 Register DUT I/Os for timing analysis

Add registers to the inputs and outputs of your DUT before performing post-synthesis timing analysis. This will ensure all elements in your design are included for timing analysis, and will give you a more accurate critical path estimation.

4.2 Block RAM mapping

One of the most common sources of area inefficiency when first targeting hardware is large register banks. Often these can be more efficiently implemented on the FPGA device by targeting its block RAM resources.

4.2.1 RAM

The following RAM blocks can be found in the Simulink Library Browser under **HDL Coder > HDL RAMs**:

- Single Port RAM (System)
- Simple Dual Port RAM (System)
- Dual Port RAM (System)
- Dual Rate Dual Port RAM

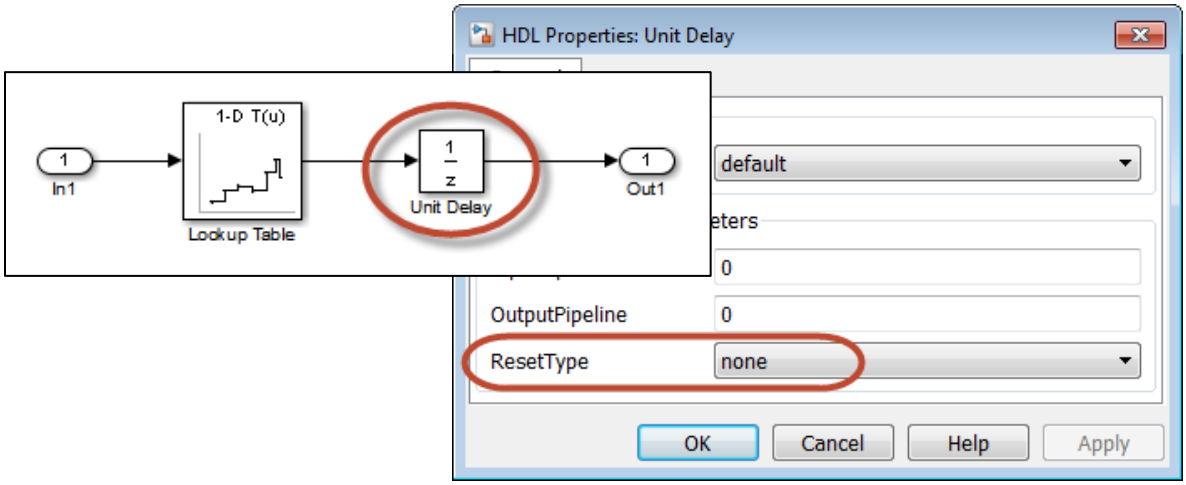
RAM blocks based on the `hdl.RAM` System object (e.g. Single Port RAM System) support vector input, non-zero initial value, as well as improved simulation speed for large RAM size.

The generated HDL for all RAM blocks uses *inference* instead of *instantiation*, so that the code can be mapped to FPGA block RAMs without having to use any vendor-specific libraries.

For Intel FPGA devices, set the parameter **RAM Architecture** to `Generic RAM without clock enable` for better mapping results. The parameter can be found under **Coding style** tab, on the **Global Settings** pane of the HDL Coder UI, or in HDL Workflow Advisor **3.1.3 Set Advanced Options**.

4.2.2 ROM

To create a ROM that can be mapped to block RAMs, use a Direct Lookup Table (n-D) block followed by a Delay block. Right-click on the Delay block, select **HDL Code > HDL Block Properties**, and set **ResetType** to `none`. Suppressing the reset on the Delay block will ensure best mapping results across different synthesis tools and FPGA devices. HDL Coder can also automate this delay insertion for you via Adaptive Pipelining – see section 3.3.4 for details.



In addition, follow these modeling guidelines for the Direct Lookup Table (n-D) block:

- For an n-bit address, specify all 2^n entries of the lookup table data, like the following example:

```
x = (0:99)';
pad = 2^nextpow2(length(x)) - length(x);
x_pad = [x; zeros(pad,1)];
```

- Make sure the parameter **Diagnostic for out-of-range input** is set to `Error`.
- The lookup table block requires an input of built-in data type such as `uint8`, `uint16`, etc. However, the extra bits will be optimized away when the HDL code is generated.

4.2.3 Additional tips

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.

When using block RAM (including blocks that use RAM internally, such as FFT), cosimulation and FIL may not match the original model after first simulation. This is because the RAM content is reset each time in Simulink, but not in the HDL simulator and FPGA.

4.3 DSP mapping

In addition to multipliers, DSP blocks in modern FPGA devices provide many other dedicated resources, such as adders and pipeline registers, for high-performance signal processing computations. The following guidelines will help you leverage those resources for designs containing common DSP operations.

4.3.1 General rules

- Reset type
  - Use the global reset type recommended in section 4.1.1, or set the HDL property **ResetType** to `none` for multiplier/adder pipeline registers. Using an “incorrect” reset type prevents DSP block resources from being fully utilized.
- Multiplier word size
  - 18x18-bit multiplication is a good starting point, as it usually maps well into FPGA DSP blocks. However, you should check the architecture of your target device to maximize the built-in multiplier resources. In particular:
    - Xilinx Virtex®-6 and 7 series provide 18x25-bit multipliers, while the UtraScale™ architecture provides 18x27-bit multiplier.
    - Intel FPGA DSP blocks can be configured as one big multiplier or several smaller ones. For example, Arria® 10 DSP supports one 27x27-bit, two 18x19-bit or three 9x9-bit multipliers. The exact word size depends on the device family and operating mode.

- When a multiplication exceeds the built-in word size, the synthesis tool either implements the “extra bits” in FPGA fabric, or splits the operation into multiple DSP blocks, resulting in lower speed and higher area.

- Fixed-point settings
  - Use full-precision fixed-point for any multiply/add operations, add output pipeline register(s), and then perform the necessary rounding/saturation after the final output register with a Data Type Conversion block. Refer to section 2.2.1 fixed-point settings for an illustration.
  - Alternatively, use rounding/saturation on a Product/Add block, and let HDL Coder automatically insert pipeline registers between the multiplier and output logic via Adaptive Pipelining – see section 3.3.4 for details.
- Pipeline registers
  - Using all available pipeline registers in the DSP blocks will give you highest clock frequency at the expense of increased latency. On the other hand, exceeding available pipeline registers may prevent synthesis tools from mapping all math operations within the DSP, resulting in sub-optimal area and timing performance. Choose pipelining level that matches the DSP architecture of your target device and your HW requirements; adjust them as needed based on synthesis results.
    - Xilinx DSP48 blocks provide up to 2 input registers and 1 output register. There is 1 level of pipeline register between the pre-adder, multiplier and final adder.
    - Intel FPGA variable precision DSP blocks provide 1-2 input registers and 1 output register. There is no pipeline register between the pre-adder, multiplier and output adders.

4.3.2 Multiply

- Pipelining guidelines for stand-alone multiply operations:
  - For Xilinx targets, use up to 2 input registers and 2 output registers.
  - For Intel FPGA targets, use 1-2 input registers and 1 output register.

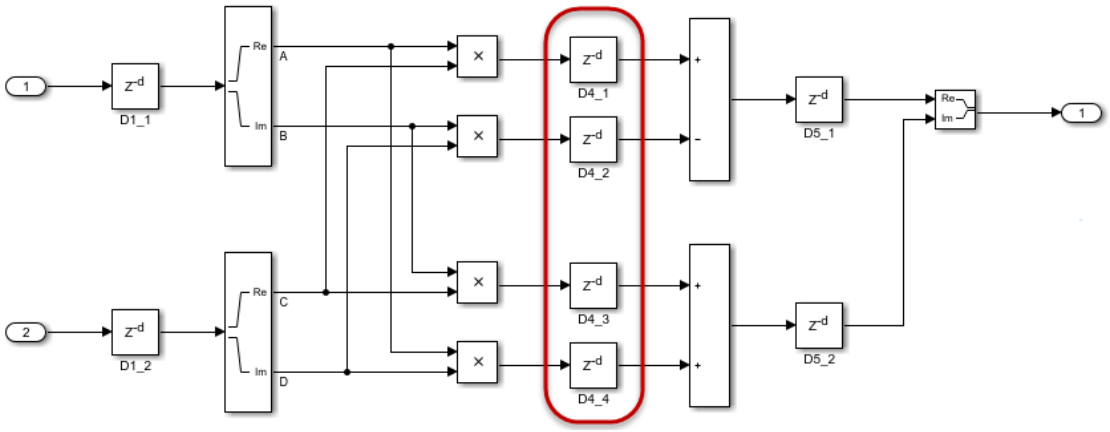
4.3.3 Complex multiply

Example: complex_multiply_example.slx

Under the hood, a complex multiply is a series of multiply and add/subtract operations. It can be implemented using 4 multipliers and 2 adders:

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

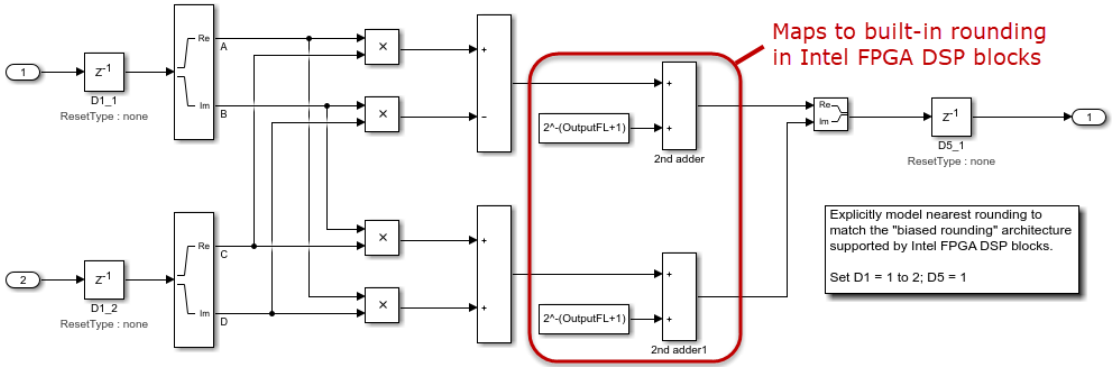
Instead of using the Simulink Product block, consider elaborating the real and imaginary components in order to accommodate the bit growth from the adders, and to pipeline the multiplier outputs before the adders, as shown in the figure below.



- Pipelining guidelines:
  - For Xilinx targets, use up to 2 input registers (D1), 1 multiplier output register (D4) and 1 output register (D5).
  - For Intel FPGA targets, use 1-2 input registers (D1) and 1 output register (D5). Do not pipeline between the multiplier and adder (D4 = 0).

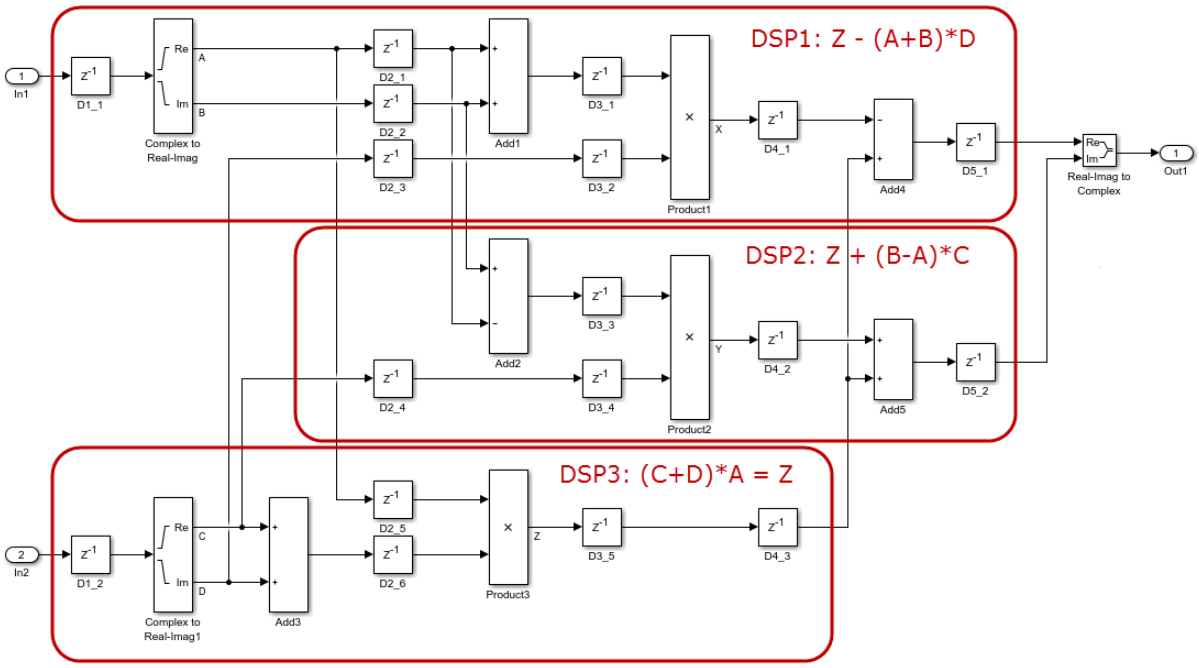
- Rounding support:

  The DSP blocks in many Intel FPGA families provide built-in support for biased rounding, which is the same as the *nearest* rounding mode in MATLAB. The example model below shows how to leverage that built-in rounding resources.



A complex multiplier can also be implemented using 3 multipliers and 5 adders:

(A + Bj)*(C + Dj) = (Z - X) + (Y + Z)j, where X = (A + B)*D, Y = (B - A)*C, Z = (C + D)*A



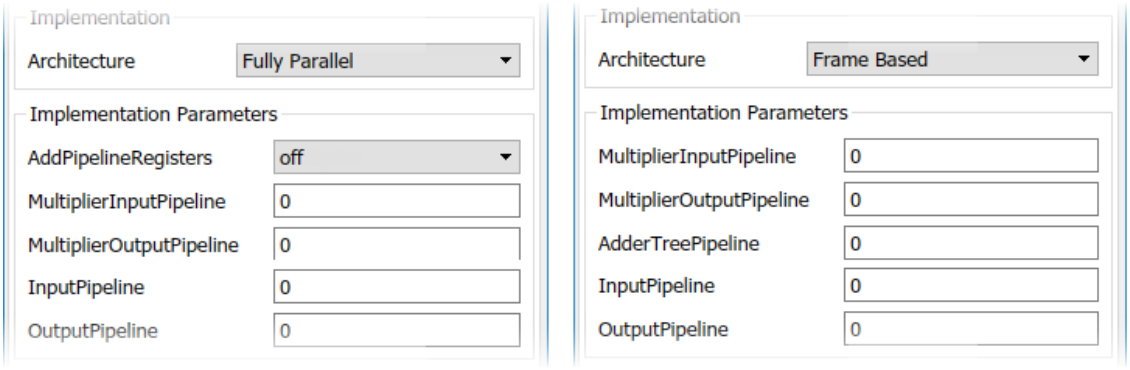
- For Xilinx FPGA targets, this architecture uses one less DSP48 block compared to the “4 multipliers, 2 adders” approach, by leveraging the pre-adders available in DSP48. In exchange, it may require more register resources and higher latency. The maximum clock speed may also be reduced.
- For Intel FPGA targets, this architecture may save DSP resources for certain word sizes and device families, but the “4 multipliers, 2 adders” approach is a better choice in many cases. Consult the device family DSP documentation on the most optimal complex multiply architecture.
- Pipelining guidelines:
  - For Xilinx targets, use 1-2 input registers, 1 multiplier input register, 1 multiplier output register and 1 output register (D1 to D5 = 1). More input registers (D1 = 2) result in a better-pipelined DSP48 but also more register resources in the fabric.
  - For Intel FPGA targets, use 1-2 input register (D1) and 1 output register (D5). Do not pipeline between the pre-adder, multiplier and adder (D2, D3, D4 = 0).

4.3.4 FIR filter

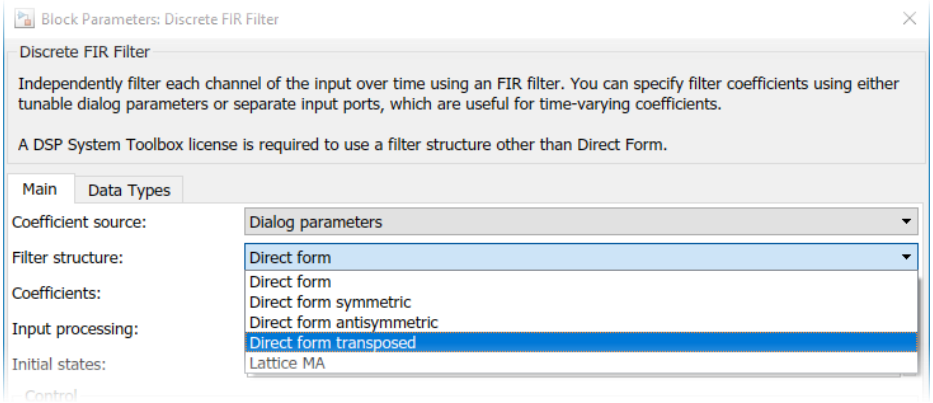
Example: fir_filter_example.slx

The following guidelines apply to the Discrete FIR Filter, FIR Decimation, and FIR Interpolation blocks. Note that the architecture and pipelining options affect the generated code only. If you prefer bit and cycle-accurate simulation behavior, use the Discrete FIR Filter HDL Optimized block (section 4.3.5).

- The `Fully Parallel, Partly Serial and Fully Serial` architecture options on the FIR blocks use a multiply-add or multiply-accumulate structure, which maps well into FPGA DSP blocks. In addition, symmetric FIR filters can also leverage the pre-adders available in most FPGA devices.
- Insert pipeline registers using the HDL block properties of the filter blocks. Select an architecture to view applicable pipelining properties:

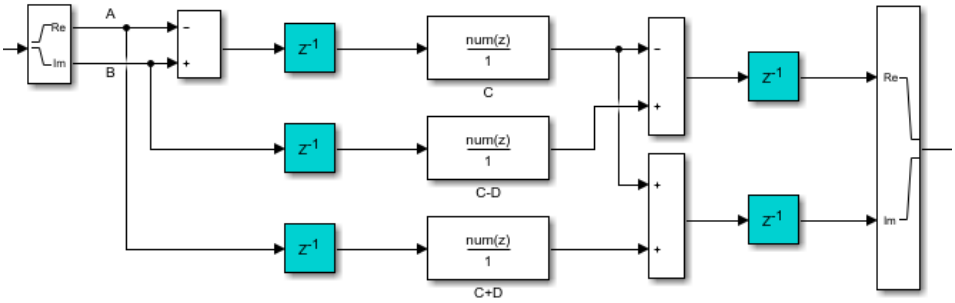


- Do not use Adaptive Pipelining (described in 3.3.4) with the following pipelining guidelines, as the automation feature may alter the explicit pipeline settings. Set **AdaptivePipelining** = `off` for the subsystem containing the filter block.
- Pipelining guidelines for a fully parallel FIR (non-symmetric / programmable):
  - Setting **Filter Structure** to `Direct form transposed` will provide the most efficient DSP block mapping. If a direct form FIR is used, set **AddPipelineRegisters** = on to generate a pipelined adder tree; this will improve timing at the expense of logic resources.



  - For Xilinx devices, use up to 2 multiplier input pipelines and 1 multiplier output pipeline.
  - For Intel FPGA devices, use 1-2 multiplier input pipeline.
- Pipelining guidelines for a symmetric, fully parallel FIR:
  - Use direct form structure with **AddPipelineRegisters** = on. While part of the adder tree will be implemented outside of DSP blocks, the transposed structure will likely result in more resources overall.
  - For Xilinx devices, use up to 2 input pipelines, 1 multiplier input pipeline and 1 multiplier output pipeline.

    - For Intel FPGA devices, use 1-2 input pipeline and 0-1 multiplier output pipeline. While the DSP block does not have a multiplier output register, Quartus Prime may be able to re-time the register elsewhere to improve overall timing.
- Pipelining guidelines for a serial FIR:
    - Use direct form structure; transposed is not supported.
    - For Xilinx targets, use 1-2 multiplier input pipelines and 1 multiplier output pipeline.
    - For Intel FPGA targets, use 1-2 multiplier input pipelines and 0-1 multiplier output pipeline. If Quartus Prime implements the output adder/accumulator outside of DSP block, use 1 multiplier output pipeline for better timing results.
- Pipelining guidelines for a symmetric, frame-based (vector) FIR:
    - For Xilinx FPGA targets, use 1-2 multiplier input pipelines and 2 pipelines between levels of adder tree (**AdderTreePipeline**). Use 1-2 multiplier output pipeline to tradeoff between DSP usage, logic area and timing performance.
    - For Intel FPGA targets, use 1-2 multiplier input registers, 1 multiplier output register and 2 registers between levels of adder tree (**AdderTreePipeline**).
    - Direct form or transposed structure does not apply for frame-based architecture.
- For complex inputs and complex coefficients, consider splitting the filter into 3 blocks with real coefficients for the most efficient DSP usage, such as the following:



4.3.5 HDL optimized FIR filter

Compared to the Discrete FIR Filter block, the *Discrete FIR Filter HDL Optimized* block provides a systolic architecture, control signals such as data valid, as well as latency and cycle-accurate simulation. Pipelining is automatically determined based on the coefficients and sharing factor parameters on the block, and the target device setting on the model.

Follow the guidelines below to achieve best synthesis results:

- Configure your target device in Configuration Parameters under **HDL Code Generation > Target**, or in HDL Workflow Advisor step **1.1 Set Target Device and Synthesis Tool**. HDL Coder uses the target device setting to determine internal pipelining placement for the block.
- Follow the guidelines on proper global reset type in section 4.3.1 if you enable the following option:
    - **Use HDL global reset** for fully parallel FIR
    - **Share DSP resources** for serial FIR
- Follow the guidelines on multiplier word size in section 4.3.1. Since built-in pipelining is optimized for implementing each multiply-add/accumulate operation with a single DSP block, larger word lengths may result in suboptimal performance.