HDL Coder™ Evaluation Reference Guide (R2023a-R2023b)

Table of Contents

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 opening the MATLAB Project *HdlExampleMenu.prj* in the "Examples" directory.

1. Getting Started

In this section, you will find information on HDL-supported blocks, configuring models for HDL designs and how to perform basic HDL operations.

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 domain-specific IP blocks are available in the Simulink Library Browser if you have the optional toolboxes installed. They can be found in the following libraries. Note that Fixed-Point Designer™ is required by HDL Coder.

- Fixed-Point Designer HDL Support
- Communications Toolbox™ HDL Support
- DSP HDL Toolbox™ (new in R2022a)
- DSP System Toolbox™ HDL Support
- Vision HDL Toolbox™
- Wireless HDL Toolbox™

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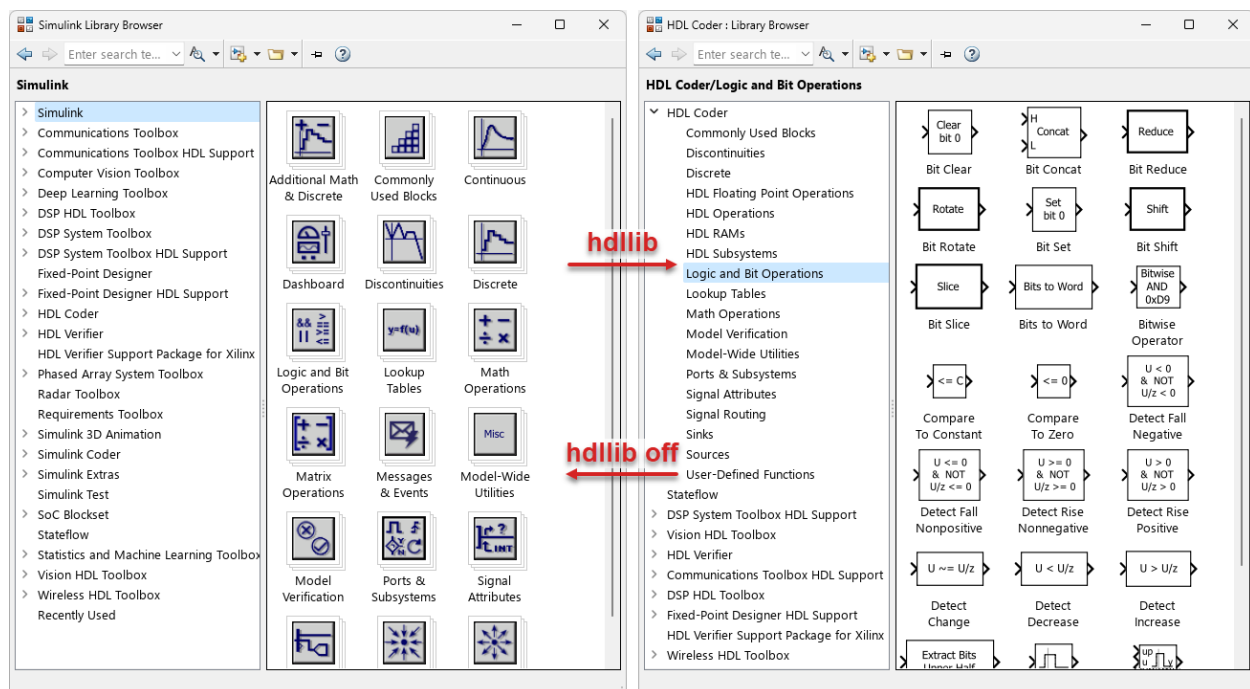
1.2 Model setup

When working with a new or existing 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 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 displaying sample time color and signal data type. Modified parameters are displayed in the MATLAB command window when you apply the `hdlsetup` command.
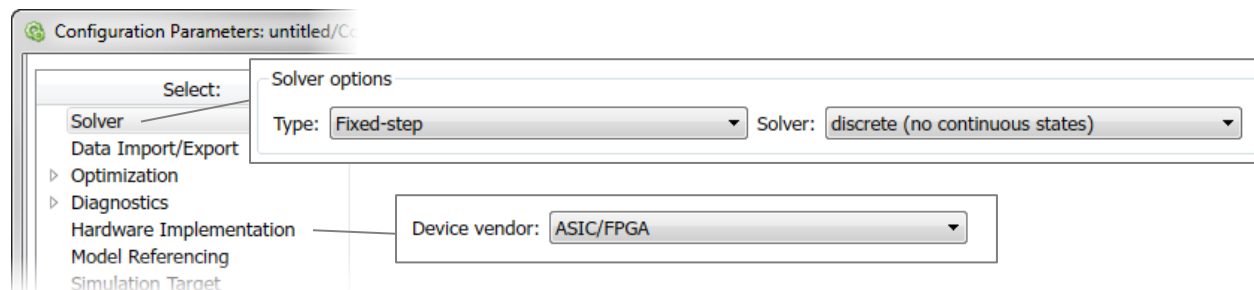
In addition, you can create a Simulink model using templates pre-configured for HDL code generation. On the **Simulink Start Page**, scroll down to the HDL Coder section to choose from a variety of starting points:
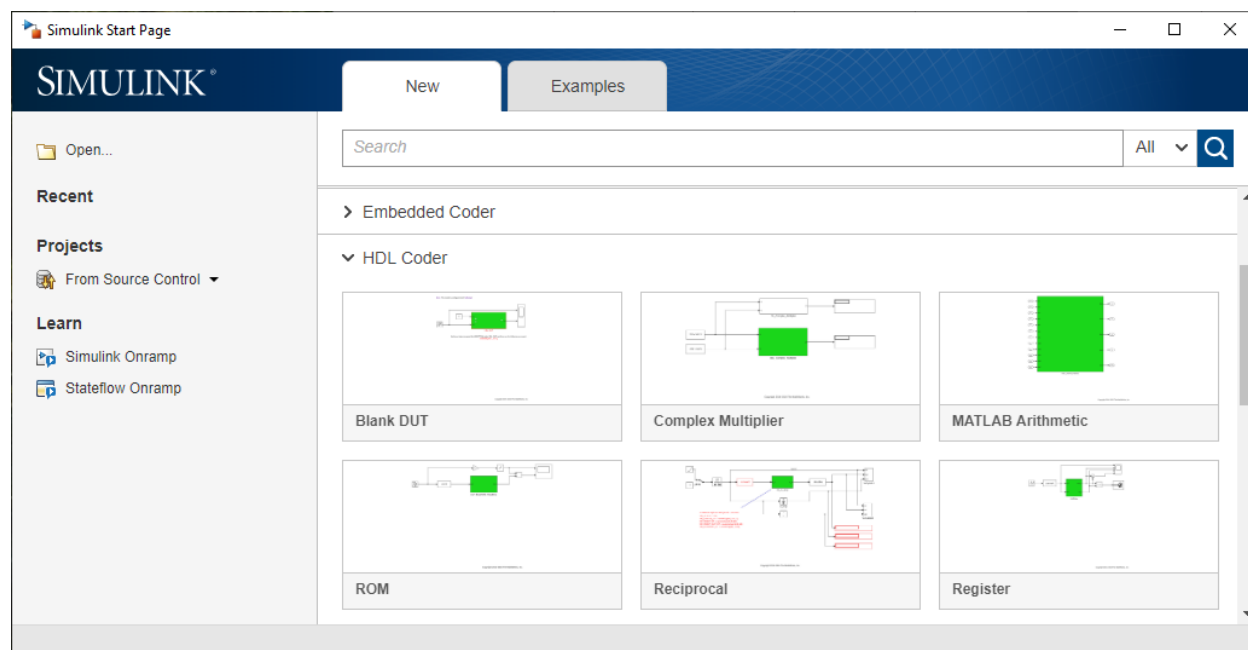
Consider creating a custom setup function if you prefer different parameter settings than those used by `hdlsetup` or HDL Coder templates, such as selecting another solver for designs containing mixed-signal components. You may also customize HDL code generation settings using the command `hdlset_param`, such as changing the default target language to Verilog. See section 3.5 for more information on scripting commands.

3

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, 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 on the Simulink Toolstrip. To activate the **HDL Code** tab, select **Apps > HDL Coder**, within the Code Generation group.
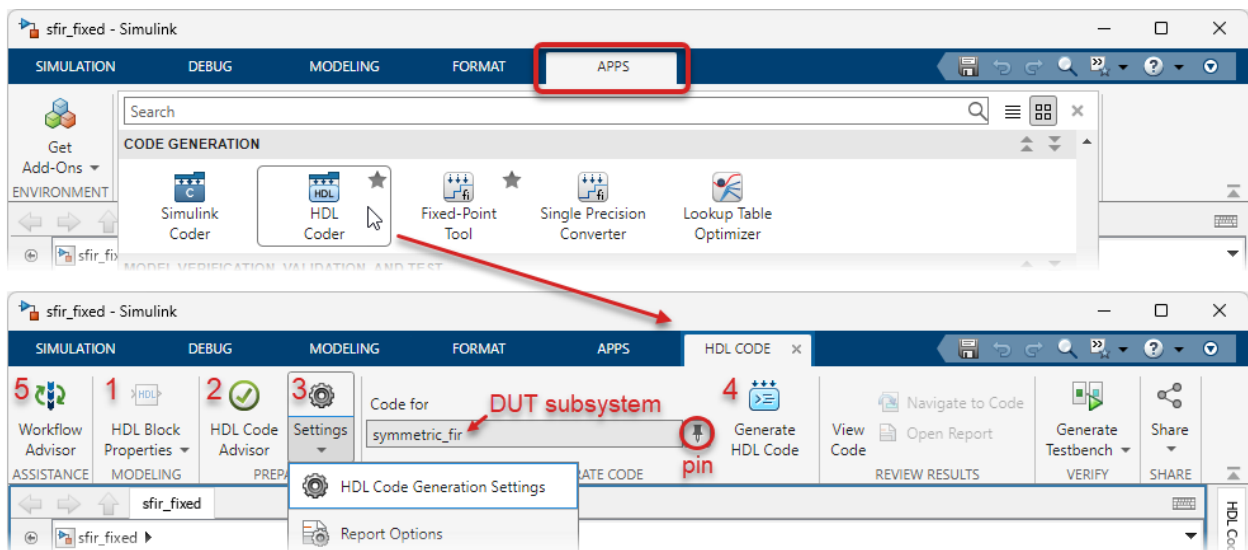
1. **HDL Block Properties**

   Configure HDL properties for the selected block or subsystem that takes effect when you generate code.

2. **HDL Code Advisor**

   Analyze your model for HDL compatibility and efficiency using the HDL Code Advisor – see 3.1 for details.

3. **HDL Code Generation Settings**

   Configure model-wide HDL properties that takes effect when you generate code.

4. **Generate HDL Code**

   Generate HDL code for the currently selected subsystem. You can also lock code generation to a specific subsystem using the pin button.

5. **Workflow Advisor**

   Launch the HDL Workflow Advisor to perform operations such as FPGA synthesis – see 3.2 for details.

2. Simulink Modeling Best Practices for HDL Designs

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. For instance, hardware requires one or more clocks, it often involves modeling multiple data rates and data flow control, and numerical decisions at the algorithm level can greatly impact the resulting hardware performance. Finally, there will be some functionality better suited for design using MATLAB Function or Stateflow blocks.
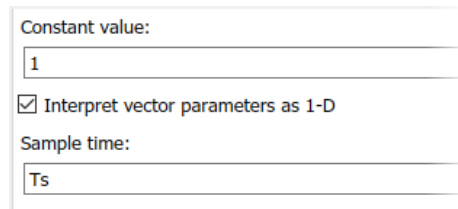
2.1 Clock, sample rate and data flow control

This section explains how clock and sample rates are represented in Simulink (2.1.1) and blocks you can use to change sample rate (2.1.2). Also discussed are ways to control data flow such as local enable & reset (2.1.3), conditional subsystems (2.1.4) and data valid interface (2.1.5).

2.1.1 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 time step in Simulink maps to 1 clock cycle in HDL. You can use a relative mapping (e.g. a sample time of 1 = 1 HDL clock cycle) or an absolute mapping (e.g. a sample time of 10e-9 = one 10 ns clock cycle 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. See 2.1.2 for more details on multi-rate modeling.

- Some optional 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.

- If you model your sample time to be equal to the actual clock speed, that does not mean it will achieve this clock speed during RTL synthesis and implementation. Delays from logic gates and wires get introduced in synthesis and implementation, and the slowest path will dictate your fastest achievable clock frequency. HDL Coder can estimate some of these delays (see 3.4.2) and provide direct feedback from RTL synthesis (4.1.2).

2.1.2 Multi-rate modeling

- A multi-rate model consists of multiple synchronous sample rates. Use sample time colors and the sample time legend (Toolstrip: **Debug > Information Overlays > Sample Time**) to help visualize different sample rates in the model. The fastest sample rate is always annotated in red.
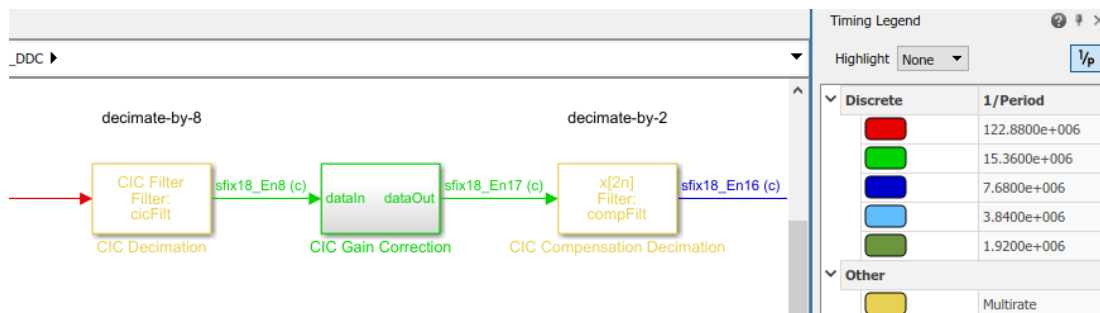
- A single clock is generated for multi-rate models by default. Blocks operating at slower sample rates are gated with clock enable signals that are active once every N clock cycles. You can also specify HDL Coder to generate multiple synchronous clock signals – see Using Multiple Clocks in HDL Coder.
- Multi-rate designs can also be represented in a single-rate model. Slower data rates can be modeled with a data/valid signal pair, while the model simulates at the clock rate. See 2.1.5 for more information on modeling with data valid.
- Use only the following blocks and block settings to change sample rate:
  - Rate Transition block. See the guidelines below for block settings.
  - Downsample block (DSP System Toolbox)

Input processing:	Elements as channels (sample based)	▼
Rate options:	Allow multirate processing	▼

  - Upsample, Repeat blocks (DSP System Toolbox)

Input processing:	Elements as channels (sample based)	▼

  - Additional blocks that change rate as part of their functionality, such as decimating / interpolating FIR and CIC filters, serializer / deserializer and dual-rate dual port RAM.
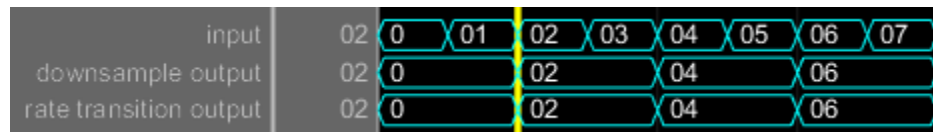- Downsampling

  Example: downsample_upsample_example.slx

  - Downsampling using the Rate Transition block must use the following settings:

    ☑ Ensure data integrity during data transfer

    ☑ Ensure deterministic data transfer (maximum delay)

  - 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 the bypass logic in the HDL. Optionally, HDL Coder can 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 or the Rate Transition block with **Ensure data integrity during data transfer** unchecked. Both become a wire in the HDL.
  - You may also use the Rate Transition block with the following settings for upsampling, as recommended in previous releases. The resulting rate transition has 1 cycle of latency.

    ☑ Ensure data integrity during data transfer

    ☑ Ensure deterministic data transfer (maximum delay)

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.
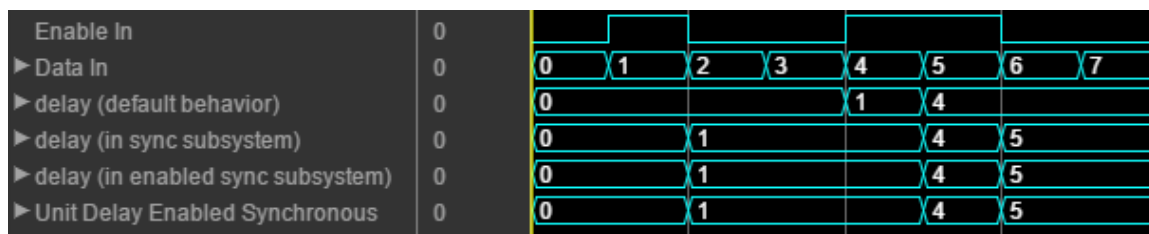
- Blocks such as Delay and Discrete FIR Filter provide optional enable and reset ports. However, these ports do not act synchronously by default. To model synchronous enable/reset, place a State Control block in the subsystem (at the same level or higher), and configure it as **Synchronous**.

- 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 or delay line, 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.
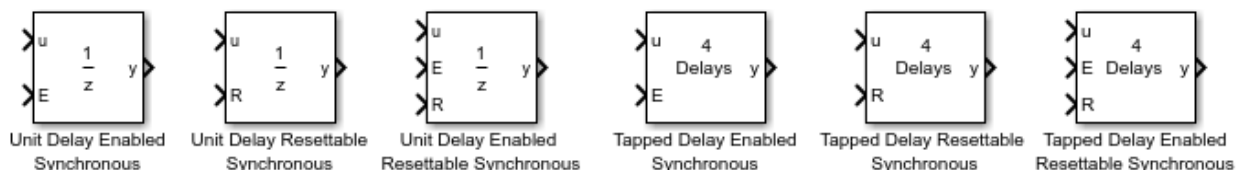
  Tip: If your design contains the Unit Delay Enabled block from older MATLAB releases, consider replacing it with the Unit Delay Enabled Synchronous block. The older version does not work in a synchronous subsystem, and is no longer available in the Simulink library browser.

- 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.

2.1.4 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.

- Using enabled subsystems in *synchronous* mode, as described in the previous section, is highly recommended.

- Perform the following steps for a triggered or enabled subsystem in classic mode (i.e. not synchronous):

  o 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).

  o Enabled and triggered subsystems contain a similar combinatorial path as the downsampling case described in the previous section. To generate more efficient HDL (i.e. no bypass logic), place a delay at the output of these subsystems.

- Limit logic within each enabled subsystem to a reasonable amount, in order to prevent large fanout and timing issue during downstream synthesis & implementation.

2.1.5 Data valid interface
Example: data_valid_example_testbench.mlx, data_valid_example.slx

In hardware, it is common for a data signal to have gaps between samples. Such a signal is typically accompanied by a *data valid* signal that tells you whether to process or discard a data sample. The presence of data valid often requires additional design considerations and efforts.

7

- Continuous data with sample rate = (1/N * clock rate) may be modeled as multi-rate as described in 2.1.2; it may also be represented in a single-rate model, using a data valid signal that toggles high once every N clock cycles. This approach gives you greater control, and it works well with the data valid interface provided by various HDL Optimized library blocks.
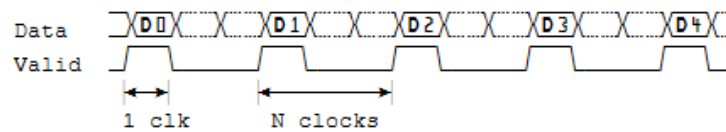

  - If you choose to share design resources such as multipliers, it works with the fact that valid only goes high once every N clock cycles, so that you can reuse one multiplier for N operations. For example, resource sharing for the FIR blocks in DSP HDL Toolbox can be specified as **Minimum number of cycles between valid input samples**. Ensure that the input data to such a block does not change faster than the specified number of cycles.

Filter structure:	Partly serial systolic
Specify serialization factor as:	Minimum number of cycles between valid input samples
Number of cycles:	N
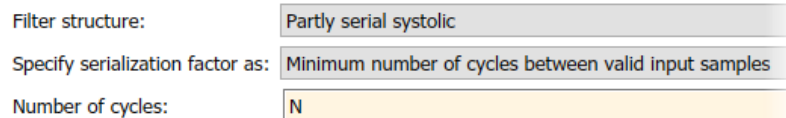
- Use a single-rate model when working with data with irregular gaps. Examples of irregular gaps include input that is continuous during normal operation but pauses during configuration; or frame-based algorithm that produces outputs in burst.
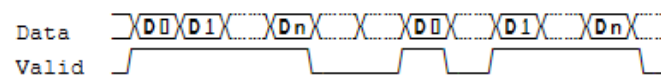
- Working with pipeline registers:
  - Model data and data valid as a pair – when you add pipeline registers to the data path, match the latency in the valid path.
  - Enable Delay Balancing (described in 3.3.4). If HDL Coder inserts data path registers via optimization features, matching registers are automatically inserted to the valid path.
- Working with algorithm with states:
  - Unlike pipeline registers, design registers such as those within an FIR filter should not process invalid data samples. Model design registers with Synchronous Enabled Delay blocks (described in 2.1.3), connecting the data valid signal to the enable port on the blocks.
  - Synchronous enabled subsystems may be used for blocks that do not provide enable or valid ports, but care should be taken to avoid downstream fanout and timing issues. Follow guidelines in 2.1.3 and 2.1.4 on the use of enabled subsystem.
  - Many IP blocks provide data valid interfaces. Leverage them where appropriate.
- Verifying data valid signal paths:
  - Inserting random gaps into the input of your design is a good way to verify the implementation of your data valid signal path. After discarding invalid output samples, the output of your design should be identical to when continuous input is provided.
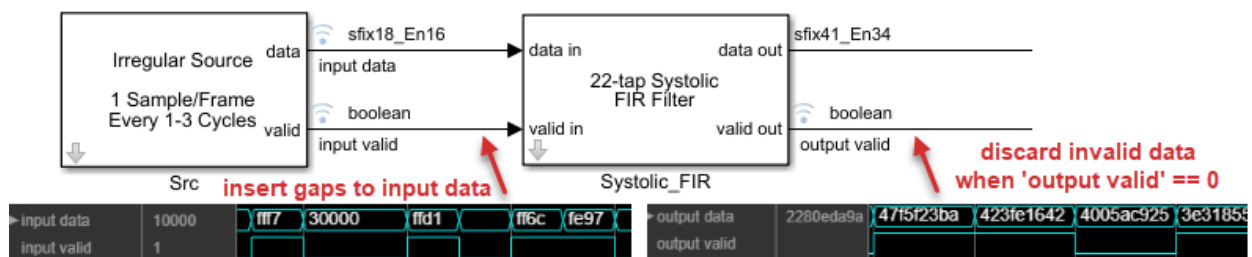
2.2 Fixed-point and floating-point settings

Using HDL Coder, you can create designs using fixed-point or floating-point data types that are hardware independent. This section outlines the best practices for designing with both data types in Simulink.

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.

- 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.

- Enable display for port data types and signal dimensions (Toolstrip: **Debug > Information Overlays > 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.

- 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.
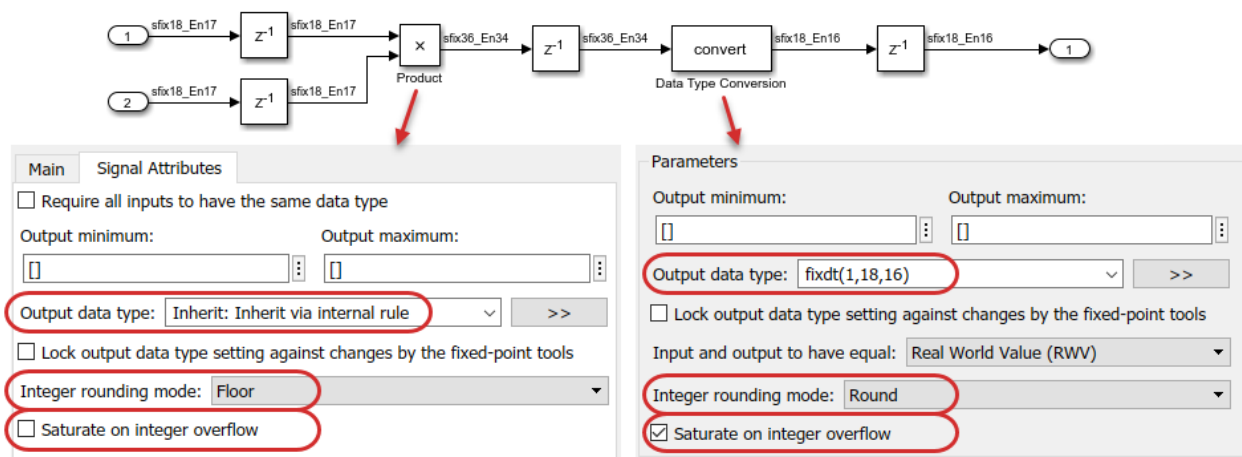
2.2.2 Native floating-point code generation

HDL Coder can generate synthesizable code for double, single and half-precision floating-point operations and data types. Floating-point is useful for high dynamic range calculations or to generate early prototype HDL. For more information on this feature, see [this video](#). Here are the best practices for using floating-point in your HDL designs:

- 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.

- Use a mix of 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.

- Configure target settings to request for vendor-specific floating-point primitives

Beginning in R2023a, you can request for vendor-specific floating-point primitives while using native floating-point. The model parameter **HDL Code Generation** > **Floating Point > Vendor Specific Floating Point Library** is enabled after you specify the synthesis tool and target device settings. For more information, see the documentation Generate HDL Code for Vendor-Specific FPGA Floating-Point Target Libraries.

- 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, such as 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 code that is 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 the MATLAB editor.

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

To define block properties such as data types and fimath, and specify function input parameter, click the **Edit Data** button to open the **Property Inspector** and **Symbols** panels:
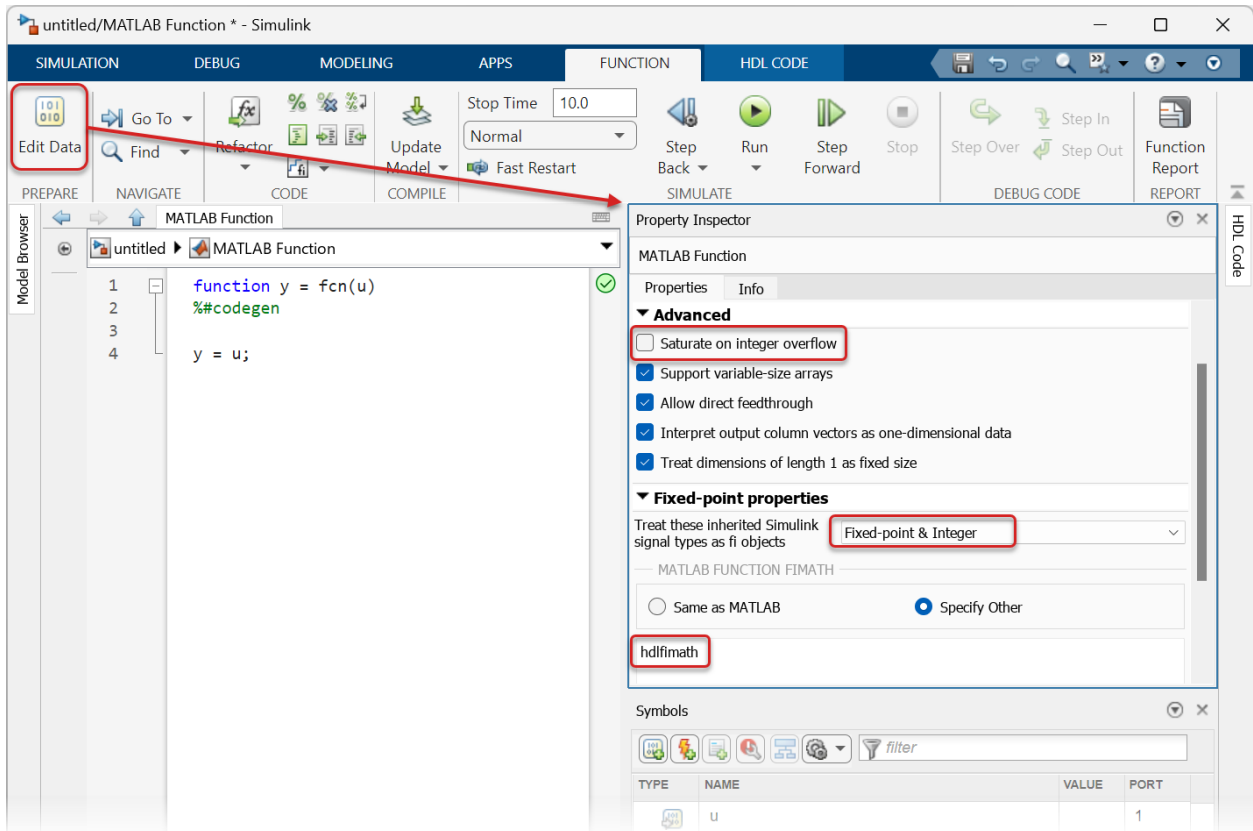
As discussed in the fixed-point section, rounding and saturation logic consume hardware resources and may impact timing performance. Unless they are required:

- o Turn off **Saturate on integer overflow**
- o 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 in the **Symbols** panel and change **Scope** to `Parameter` in the **Property Inspector** panel. Un-check the **Tunable** box under the **Advanced** section unless you intend to turn the parameter into a top-level input port in the HDL code.

11
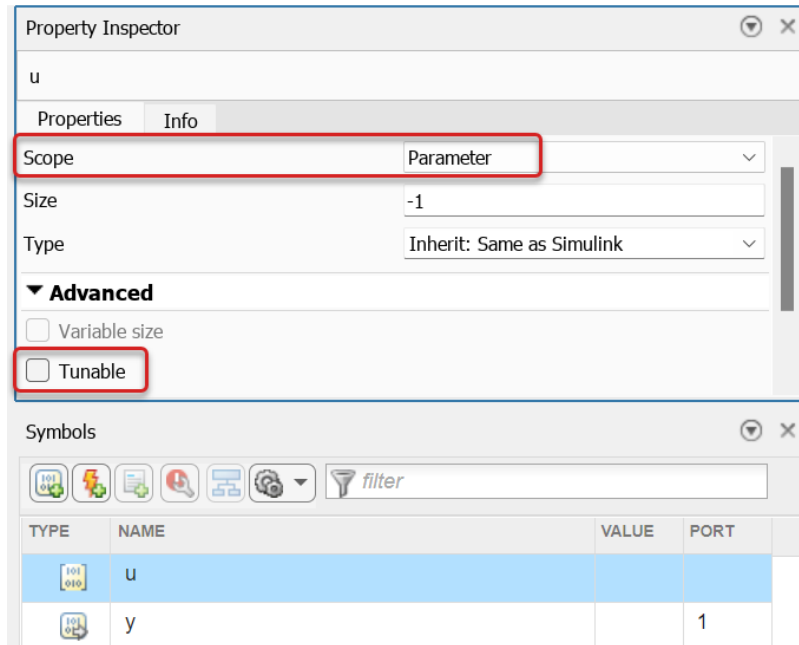
- Consider using the `MATLAB Datapath` block architecture if your MATLAB Function block contains algorithm that requires speed or area optimizations. See section 3.3.2 for more details.

2.3.2 Modeling registers with MATLAB code
Registers can be modeled a number of 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.

2. `hdl.Delay` or `dsp.Delay` are straightforward ways to model pipeline registers. `dsp.Delay` from the DSP System Toolbox library also provides a local reset option. However, they cannot be used 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);
```

12

2.4 Using a Stateflow chart

Stateflow provides a graphical language for designing finite state machines (FSM). It enables you to visually describe control logic and its interaction with the rest of your Simulink design. As code becomes more complex, Stateflow chart remains concise and easy to understand.


Hysteresis

Hysteresis + Debouncing
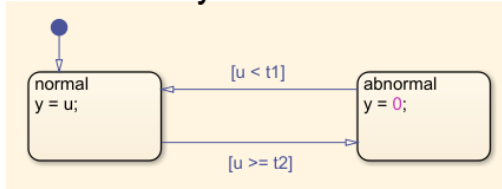
```
Chart1_1_output : PROCESS (is_Chart1, is_active_Chart1, t1_signed,
BEGIN
  is_Chart1_next <= is_Chart1;
  is_active_Chart1_next <= is_active_Chart1;
  IF is_active_Chart1 = to_unsigned(16#00#, 8) THEN
    is_active_Chart1_next <= to_unsigned(16#01#, 8);
    is_Chart1_next <= IN_normal;
    y_tmp <= u_signed;
  ELSE
    CASE is_Chart1 IS
      WHEN IN_abnormal =>
        IF u_signed < t1_signed THEN
          is_Chart1_next <= IN_normal;
          y_tmp <= u_signed;
        ELSE
          y_tmp <= to_signed(16#00#, 8);
        END IF;
      WHEN OTHERS =>
        --case IN_normal:
        IF u_signed >= t2_signed THEN
          is_Chart1_next <= IN_abnormal;
          y_tmp <= to_signed(16#00#, 8);
        ELSE
          y_tmp <= u_signed;
        END IF;
    END CASE;
  END IF;
END PROCESS Chart1_1_output;
```
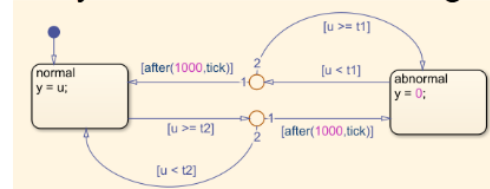
```
Chart2_1_output : PROCESS (is_Chart2, is_active_Chart2, t1_signed, t2_signed, temporalCounter_i1, u_signed)
  VARIABLE temporalCounter_i1_temp : unsigned(9 DOWNTO 0);
BEGIN
  temporalCounter_i1_temp := temporalCounter_i1;
  is_Chart2_next <= is_Chart2;
  is_active_Chart2_next <= is_active_Chart2;
  IF temporalCounter_i1 < to_unsigned(16#3FF#, 10) THEN
    temporalCounter_i1_temp := temporalCounter_i1 + to_unsigned(16#001#, 10);
  END IF;
  IF is_active_Chart2 = to_unsigned(16#00#, 8) THEN
    is_active_Chart2_next <= to_unsigned(16#01#, 8);
    is_Chart2_next <= IN_normal;
    temporalCounter_i1_temp := to_unsigned(16#000#, 10);
    y_tmp <= u_signed;
  ELSE
    CASE is_Chart2 IS
      WHEN IN_abnormal =>
        IF u_signed < t1_signed THEN
          IF temporalCounter_i1_temp >= to_unsigned(16#3E8#, 10) THEN
            is_Chart2_next <= IN_normal;
            temporalCounter_i1_temp := to_unsigned(16#000#, 10);
            y_tmp <= u_signed;
          ELSIF u_signed >= t1_signed THEN
            is_Chart2_next <= IN_abnormal;
            temporalCounter_i1_temp := to_unsigned(16#000#, 10);
            y_tmp <= to_signed(16#00#, 8);
          ELSE
            y_tmp <= to_signed(16#00#, 8);
          END IF;
        ELSE
          y_tmp <= to_signed(16#00#, 8);
        END IF;
      WHEN OTHERS =>
        --case IN_normal:
        IF u_signed >= t2_signed THEN
          IF temporalCounter_i1_temp >= to_unsigned(16#3E8#, 10) THEN
            is_Chart2_next <= IN_abnormal;
            temporalCounter_i1_temp := to_unsigned(16#000#, 10);
            y_tmp <= to_signed(16#00#, 8);
          ELSIF u_signed < t2_signed THEN
            is_Chart2_next <= IN_normal;
            temporalCounter_i1_temp := to_unsigned(16#000#, 10);
            y_tmp <= u_signed;
          ELSE
            y_tmp <= u_signed;
          END IF;
        ELSE
          y_tmp <= u_signed;
        END IF;
    END CASE;
  END IF;
  temporalCounter_i1_next <= temporalCounter_i1_temp;
END PROCESS Chart2_1_output;
```

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 **Modeling > Chart Properties** on the Simulink Toolstrip.
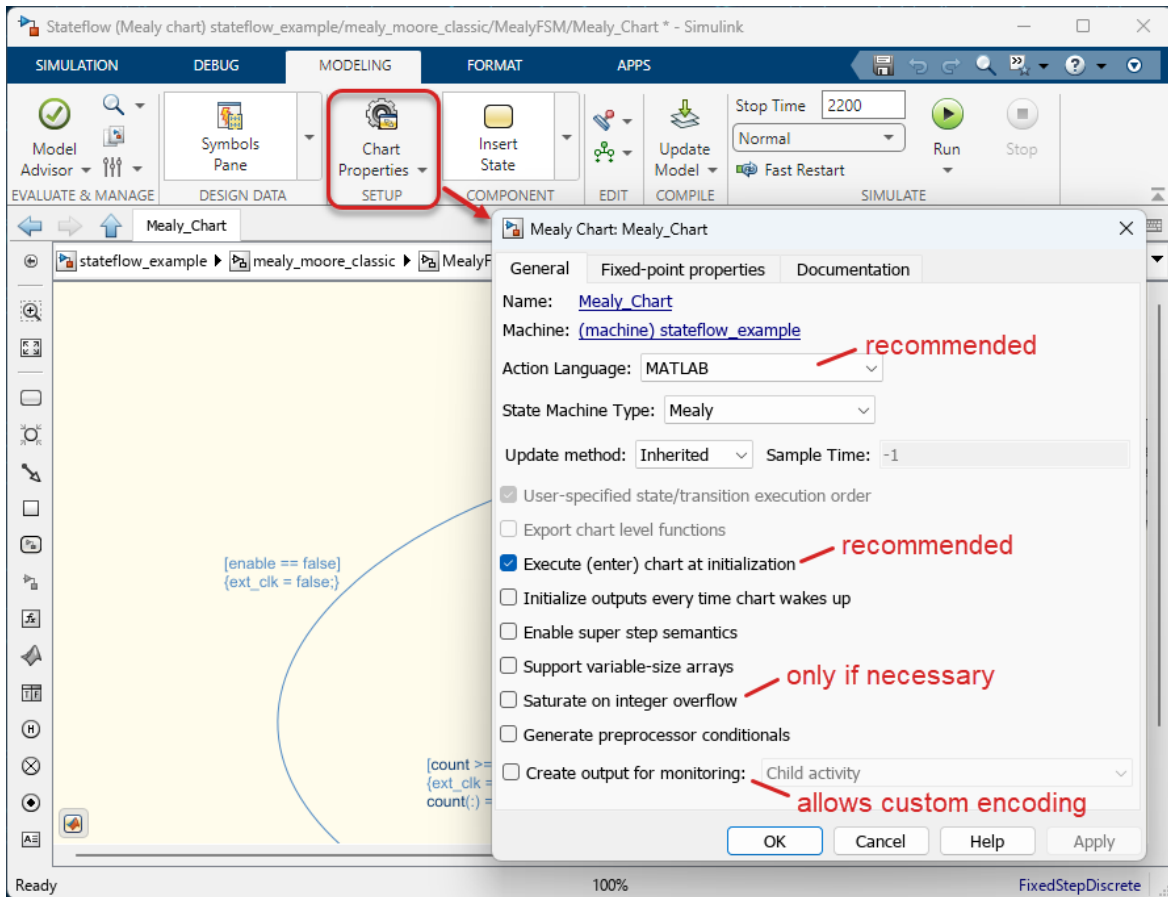
- Set **Action Language** to MATLAB (default). C action language is not recommended although it is allowed.

- When used inside synchronous subsystems (described in 2.1.4), **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.

- **Execute (enter) chart at initialization** is no longer required from R2021b, but it is recommended so that the FSM initializes to the state with default transition on reset. This option is always enabled for Moore chart.

- **Initialize Outputs Every Time Chart Wakes Up** is no longer required for Moore charts from R2021a. When this option is disabled, an output value is automatically retained (i.e. registered) when it is not explicitly assigned. For more information, see this documentation.

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 *Classic* state machine allows both Mealy and Moore semantics, although it is preferred to configure a state machine as either Mealy or Moore.

- 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.

- A *Moore* state machine generates outputs based on the current state only. Outputs are defined in states only. The generated code is generally more readable; there is also an option to register all outputs via the HDL property **ClockDrivenOutput** from R2022b.

- For additional information, see Advantages of Mealy and Moore Charts and Generate HDL for Mealy and Moore Finite State Machines
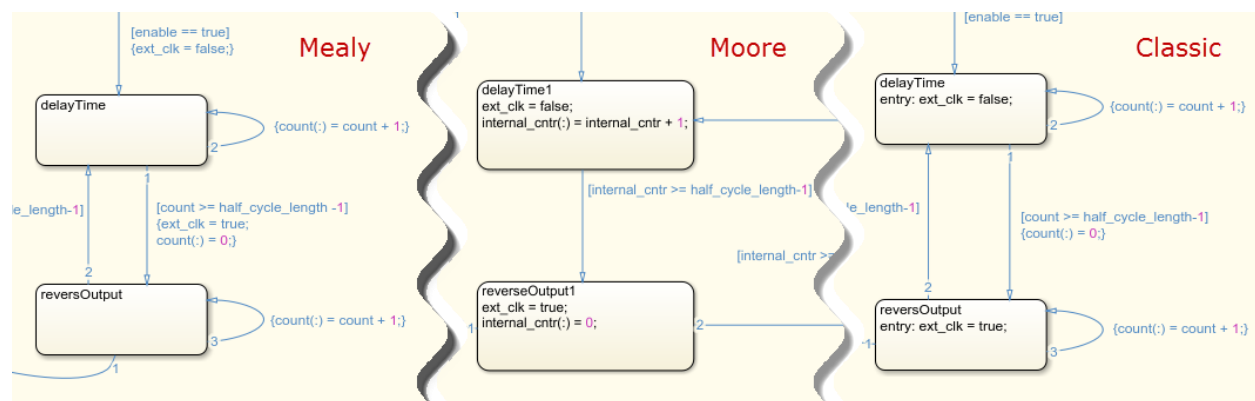
2.4.3 State Encoding Options
You may configure a state encoding scheme for the entire model, or just for an individual state machine using a custom enumeration definition.

- To specify state machine encoding scheme such as one-hot for the entire model, 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.
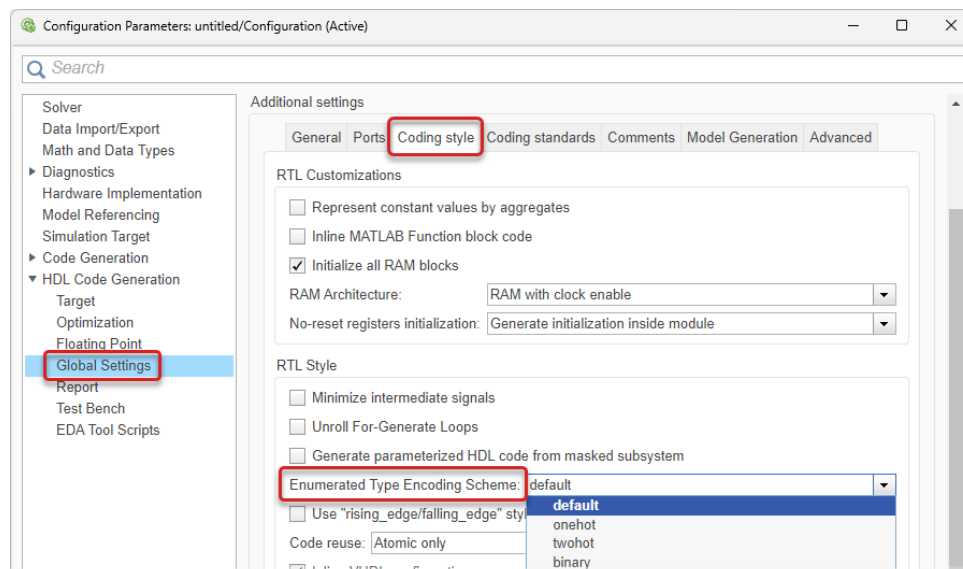
- When you enable **Create output for monitoring** for a chart (select **Modeling > Chart Properties** on the toolstrip), the active state is exposed as an output port. In addition, you can enable **Define enumerated type manually** to customize the enumeration.

In R2022b, the following enhancements allow you to fully control state encoding via custom enumeration.

  o You may remove the NONE state enumeration that is created by default when outputting active states, with some restrictions. It may prevent the consumption of extra hardware resources where the NONE state is never used (e.g. when there are exactly 2^N user-defined states). For more information, see this guideline.

  o You may use any order for the enumeration's numerical values, such as gray code. Previously, the numerical values must be monotonically increasing.

```
classdef ChartMooreModeType < Simulink.IntEnumType

    enumeration
        A(0),
        B(1),
        C(3),
        D(2)
    end
```

- When not using custom enumeration, states are encoded using the alphabetical order of state names.
- The state ranked last in alphabetical order is always used as the case statement default value in the generated code. For more information, see this guideline.

2.4.4 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.
- For Mealy and Classic charts, make sure inputs are properly registered to prevent long combinatorial data paths.
- Configure the model-level diagnostics **No unconditional default transition** to error (default value) to ensure proper reset behavior.
- Floating-point data types are not supported in Stateflow Charts. Consider manual typing of the signals to use integer or fixed-point prior to HDL Code generation.
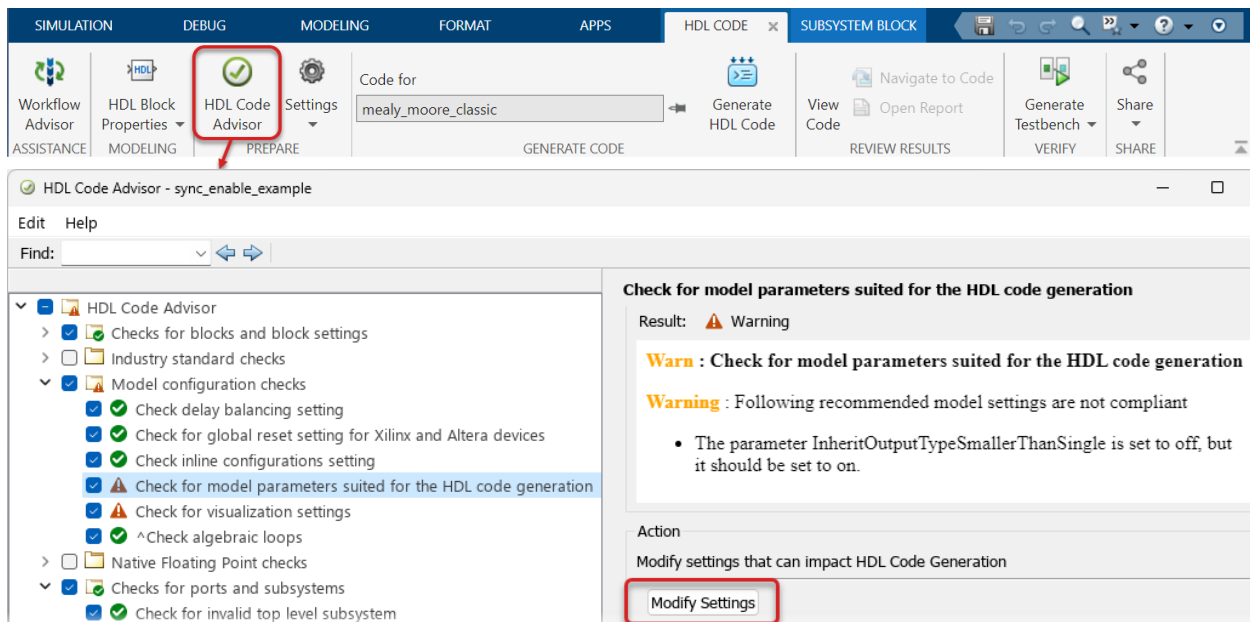
3. Code Generation Tools

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

3.1 HDL Code Advisor

Before generating code, you can use the HDL Code Advisor 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 Code Advisor UI.

To launch the HDL Code Advisor, set the DUT subsystem and click **HDL Code Advisor** on the HDL Code Toolstrip (see 1.4 on using the Toolstrip). Consult the documentation for the list of HDL Code Advisor checks.

3.2 HDL Workflow Advisor

HDL Coder provides a tool 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

- Generate an IP core with AXI interfaces for integration with a larger system design

- Verify your design on hardware using FPGA-in-the-Loop

- Generate FPGA bitstream and deploy your design on hardware for real-time prototyping and debugging

Follow these steps to start using HDL Workflow Advisor:

- First, use `hdlsetuptoolpath` to add paths for AMD Vivado™, AMD ISE™, Intel® Quartus® Prime Standard / Pro or Microchip® 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\2022.1\bin');

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

hdlsetuptoolpath('ToolName','Microchip Libero SoC','ToolPath',...
    'C:\Microchip\Libero_SoC_v2022.1\Designer\bin\libero.exe');
```

- Launch HDL Workflow Advisor by selecting / pinning the DUT subsystem, then click **Workflow Advisor** on the HDL Code Toolstrip.
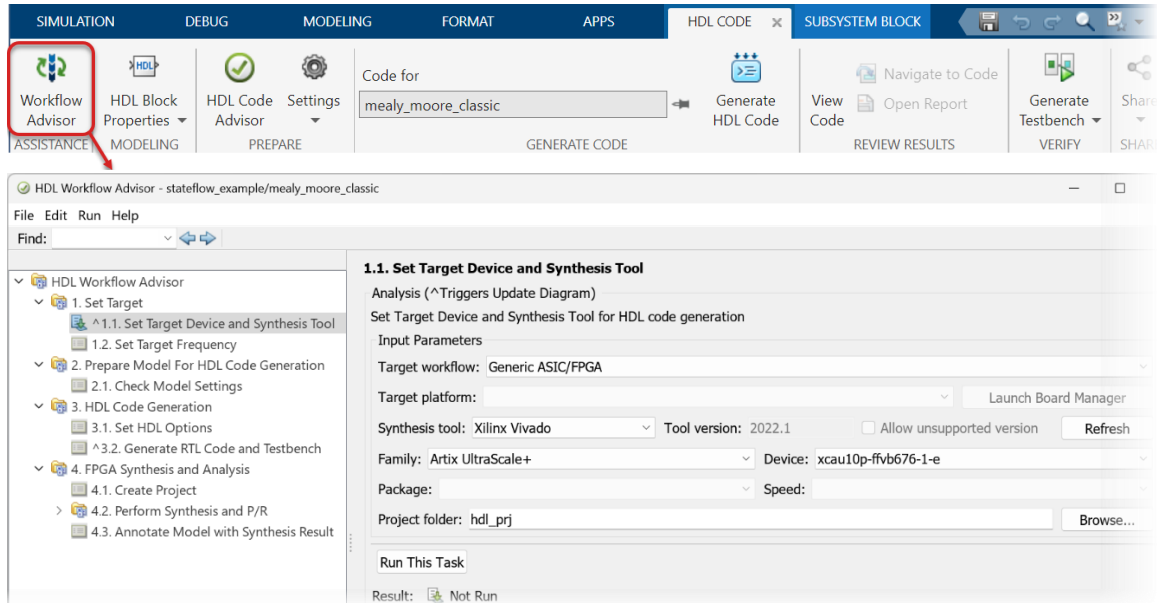
- In **1.1 Set Target Device and Synthesis Tool**, set your desired workflow, hardware target and synthesis tool. For design iterations through synthesis, use the default `Generic ASIC/FPGA` in **Target workflow**. Note: Some pipelining optimizations are activated when a synthesis tool / hardware target is selected. These optimizations may introduce latency in the generated design. See the next section for more information.

- 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**.

3.3 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.

- When an optimization setting introduces delays to a data path during code generation, HDL Coder automatically inserts matching delays to parallel data paths, so that the generated HDL produces the same numeric results as your original Simulink model, but at a later time step (clock cycle). Automatic delay balancing is enabled by default.

- Library blocks in DSP HDL Toolbox, Vision HDL Toolbox and Wireless HDL Toolbox are specifically designed for HDL code generation. The generated code from these blocks are bit- and cycle-accurate to their simulation behavior in Simulink. Optimization settings have no effect on these blocks.

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 selecting the block, then click **HDL Block Properties** on the HDL Code Toolstrip.
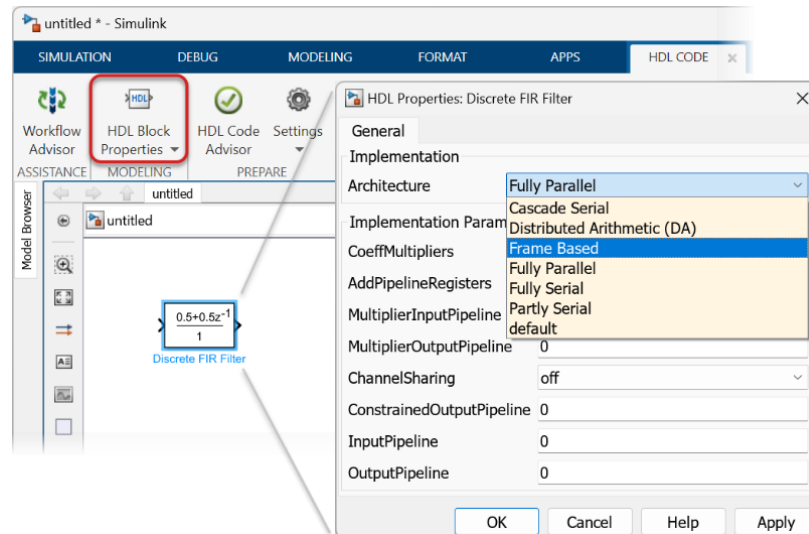

- Many blocks offer multiple architectures for different speed/area requirements. For example, the Discrete FIR block uses a Fully Parallel architecture by default, but you can select a Frame Based architecture to process multiple input samples in a single clock cycle.

- 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.

- Sometimes it is desirable to simulate latency introduced by block-level optimizations in the Simulink model, instead of only in the generated HDL. You can do so by adding the expected amount of latency (i.e. delay blocks) behind the block in your design. The delays will be "absorbed" during code generation, resulting in HDL code that is cycle-accurate to your Simulink model. Consult the block documentation or generate preliminary code to determine the amount of latency required.

- Complex algorithm modeled using the MATLAB Function block may achieve better optimization results using the `MATLAB Datapath` architecture. This architecture option allows HDL Coder to perform optimizations across the MATLAB Function block boundary by transforming code within the MATLAB Function block into Simulink blocks.
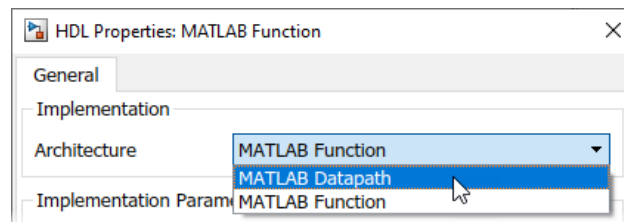
3.3.3 Subsystem-level optimizations

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.
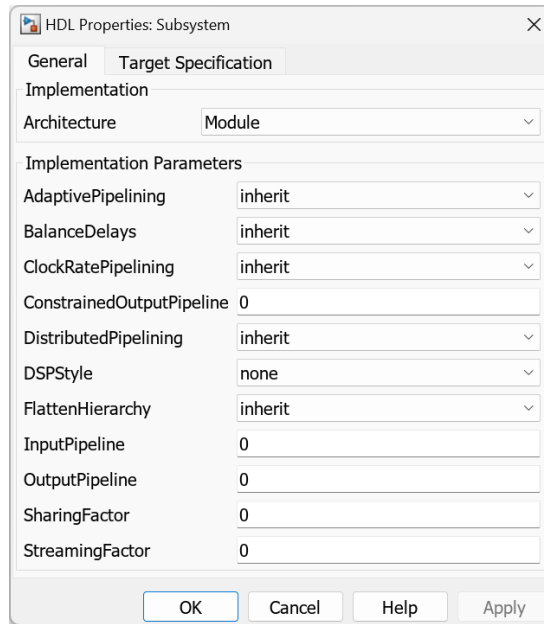
- Area optimizations

    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 delays in the design. Beginning in R2023a, distributed pipelining can move delays across hierarchical boundaries while preserving the subsystem hierarchies.

- 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 located in the Configuration Parameters under **HDL Code Generation > Optimization**.
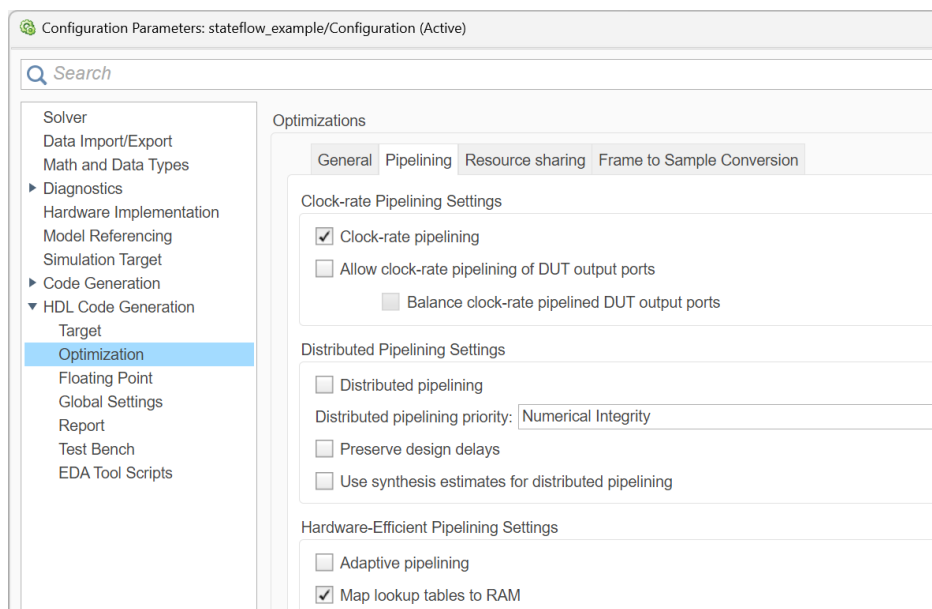
- Clock-rate pipelining

  o In a multi-rate design, registers in slower data paths do not produce new values every clock cycle. You can pack more logic between these registers if you use multicycle path constraints to relax their timing requirements during synthesis. Alternatively, you can sample slow data paths at the clock rate and use multiple pipeline registers to break up the logic. This allows your design to meet timing without relying on synthesis constraints. Clock-rate pipelining allows HDL Coder to do this automatically when performing other speed and area optimizations for a multi-rate design.

  o **Clock-rate pipelining** is enabled by default, but it only takes effect when another speed or area optimization has also been enabled. In addition to multi-rate models, it also works for single-rate models that have multiple HDL sample rates introduced by other optimization settings such as Oversampling Factor.

- Map lookup tables to RAM

  o In Simulink, lookup table blocks do not have latency. To implement them as RAM blocks in hardware, a pipeline register without reset is required at the output of each lookup table block (section 4.2.2). **Map lookup tables to RAM** allows HDL Coder to insert this register for you in the generated HDL. If your model already has a delay after the lookup table block, HDL Coder absorbs the delay without introducing additional latency.

  o **Map lookup tables to RAM** is available from R2021b. It is enabled by default, but automatic register insertion only takes place when **Synthesis Tool** is also set. The functionality was previously provided by **Adaptive pipelining**.

- Adaptive pipelining

  o Some Simulink operations 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 2.1.2). The adaptive pipelining optimization detect these patterns and automatically insert pipeline registers for you, using information such as target device and clock frequency to determine the optimal amount and location.

  o Beginning in R2021a, **Adaptive pipelining** is disabled by default. Enable the parameter (at the model or subsystem level) and specify a target device to use adaptive pipelining. To automate multiplier pipelining, also specify **Target Frequency** (>0). If your model is saved before R2021a with **Adaptive pipelining** enabled, you must manually enable the parameter again.

- Balance delays

  o Beginning in R2021b, **Balance Delays** has been removed from the Configuration Parameters UI to discourage the parameter from being disabled. If automatic delay balancing is unsuccessful, consider options such as increasing latency budget in a feedback loop or disabling delay balancing for stable paths only. Consult the Delay Balancing documentation for more details.

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:

```
Command Window                                                                            ⊙
### Generating HDL for 'biquad_model/DUT'.
### Starting HDL check.
### The code generation and optimization options you have chosen have introduced additional pipeline delays.
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additional delays.
### Output port 0: 3 cycles.
### Output port 1: 3 cycles.
### To highlight blocks that obstruct distributed pipelining, click the following MATLAB script: hdlsrc\biquad_mo
### To clear highlighting, click the following MATLAB script: hdlsrc\biquad_model\clearhighlighting.m
### Begin VHDL Code Generation for 'biquad_model'.
### MESSAGE: The design requires 2 times faster clock with respect to the base rate = 1.25e-05.
### Working on biquad_model/DUT/FSM as hdlsrc\biquad_model\FSM.vhd.
### Working on DUT_tc as hdlsrc\biquad_model\DUT_tc.vhd.
### Working on biquad_model/DUT as hdlsrc\biquad_model\DUT.vhd.
### Generating package file hdlsrc\biquad_model\DUT_pkg.vhd.
### Generating HTML files for code generation report at biquad_model_codegen_rpt.html.
### Creating HDL Code Generation Check Report DUT_report.html
### HDL check for 'biquad_model' complete with 0 errors, 0 warnings, and 2 messages.
### HDL code generation complete.
```

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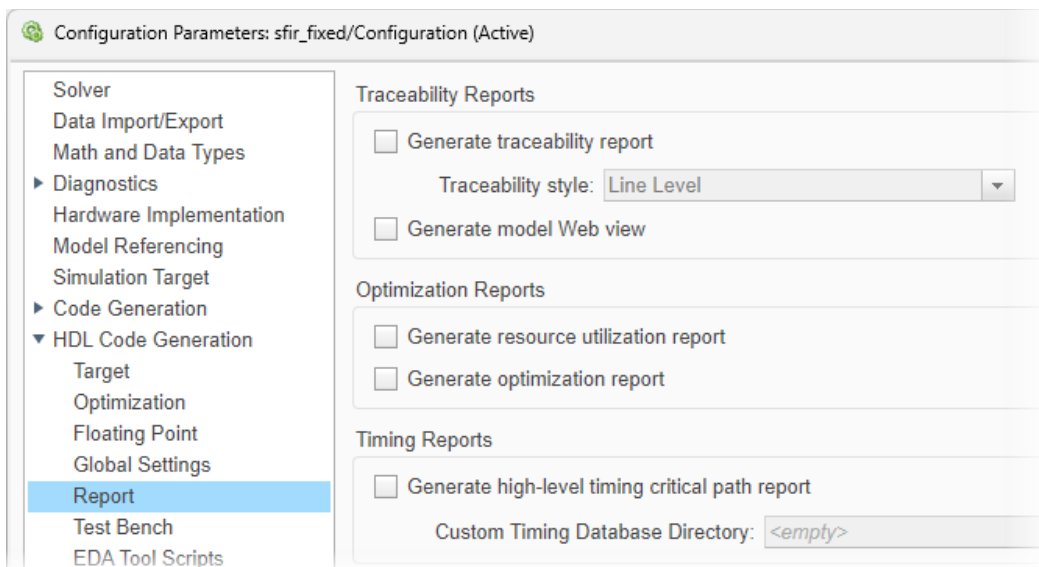
```
### Creating HDL Code Generation Check Report DUT_report.html
### HDL check for 'biquad_model' complete with 0 errors, 0 warnings, and 2 messages.
```

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**.

```
Configuration Parameters: sfir_fixed/Configuration (Active)

  Solver                          Traceability Reports
  Data Import/Export
  Math and Data Types                □ Generate traceability report
  ▶ Diagnostics
  Hardware Implementation                 Traceability style: Line Level            ▼
  Model Referencing                  □ Generate model Web view
  Simulation Target
  ▶ Code Generation               Optimization Reports
  ▼ HDL Code Generation
    Target                            □ Generate resource utilization report
    Optimization
    Floating Point                    □ Generate optimization report
    Global Settings
    Report                        Timing Reports
    Test Bench
    EDA Tool Scripts                  □ Generate high-level timing critical path report

                                         Custom Timing Database Directory: <empty>
```

- *Traceability report* links a Simulink block to its generated RTL code, and vice versa:
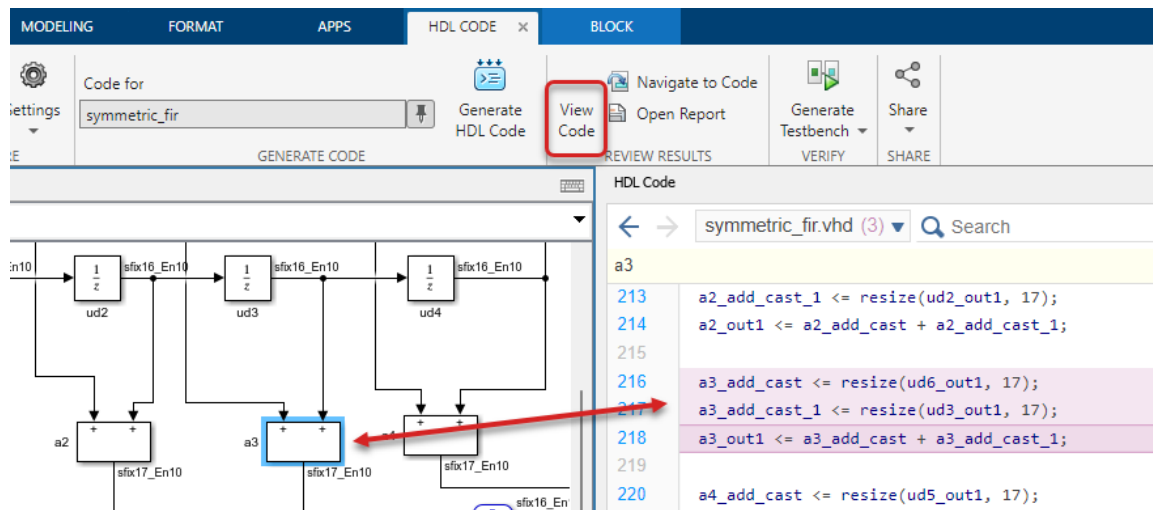
- *Resource utilization report* provides a high-level usage summary for resources such as multipliers, registers and RAMs. While the report is not specific to your target-device (e.g. 10 18x30 bit multiplier instead of the number of DSP48 on a AMD FPGA device), it provides a quick resource estimation 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.
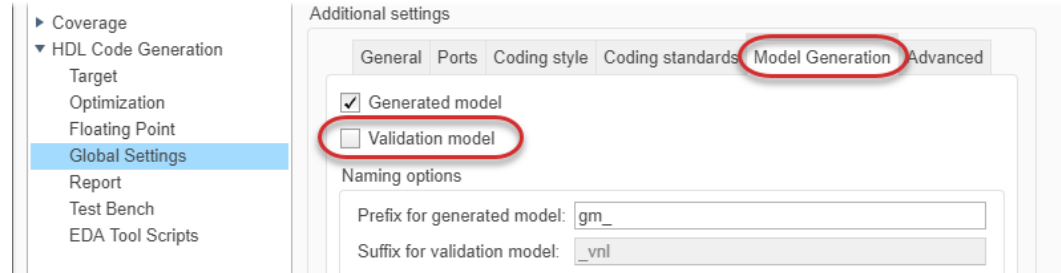
3.4.3 Generated and validation models

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.
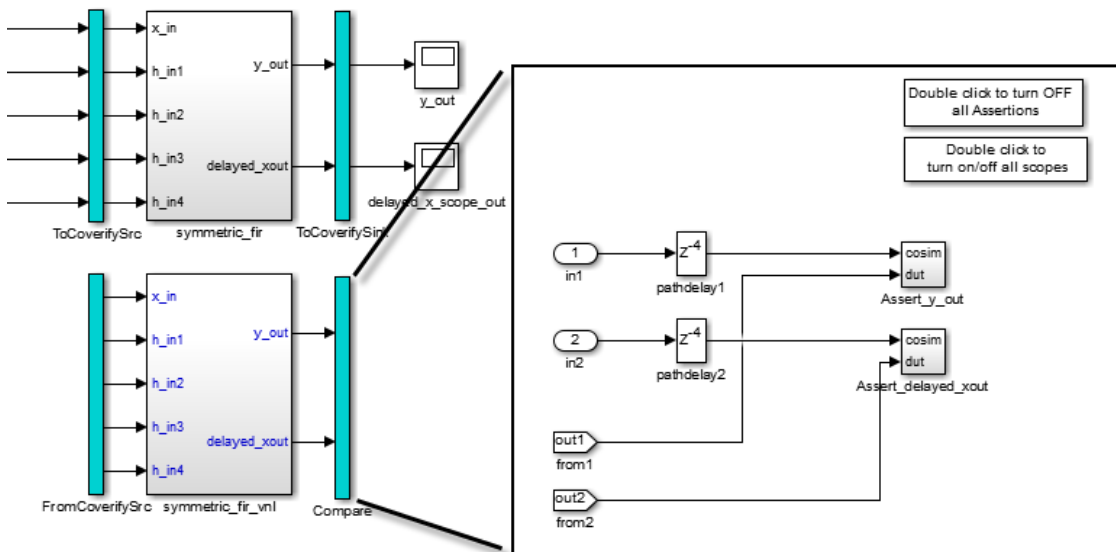
- Generated model
  - By default, a *generated model* is created and loaded in memory when you generate code from a Simulink model. It is a behavioral model that reflects the latency and numeric changes resulting from the HDL architecture or optimization settings you have chosen, and it is bit- and cycle-accurate 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.

23

- 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 Configuration Parameters, under **HDL Code Generation > Global Settings > Model Generation**, 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 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.
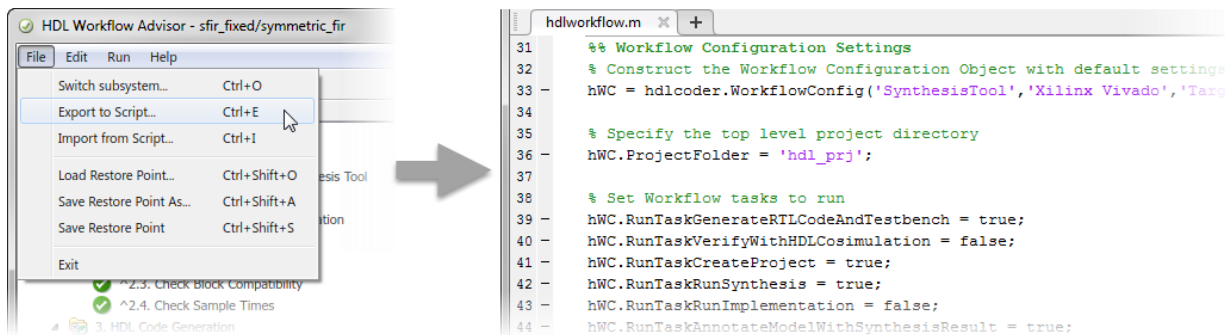
4. Targeting FPGA Hardware

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 AMD FPGA devices, use `synchronous` global reset.

- For Intel FPGA devices, use `asynchronous` global reset, or `synchronous` global reset for newer device families such as Stratix® 10.

- For Lattice devices, you may use either type of 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**.
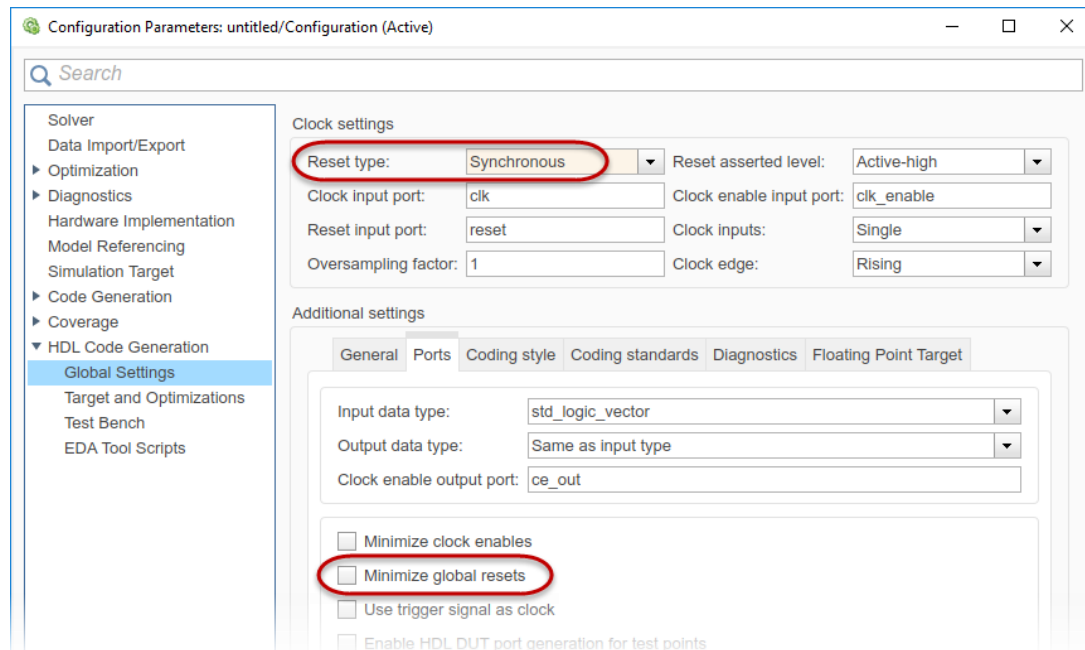

4.1.2 Synthesis tool integration

- If you use AMD Vivado / ISE, Intel Quartus Prime / Quartus Pro or Microchip Libero SoC, you can create an FPGA project, synthesize your design, analyze timing and more using HDL Workflow Advisor (section 3.2), 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 device are specified, latency may be introduced to the generated design due to global pipelining settings. See section 3.3.4 for details.

- If your synthesis tool is not supported by HDL Workflow Advisor, you can generate a Tcl script that can be sourced to create and synthesize a design. 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.

For example, the Tcl script generated using the following settings can be used with SynplifyPro® to target a Lattice CertusPro-NX FPGA.
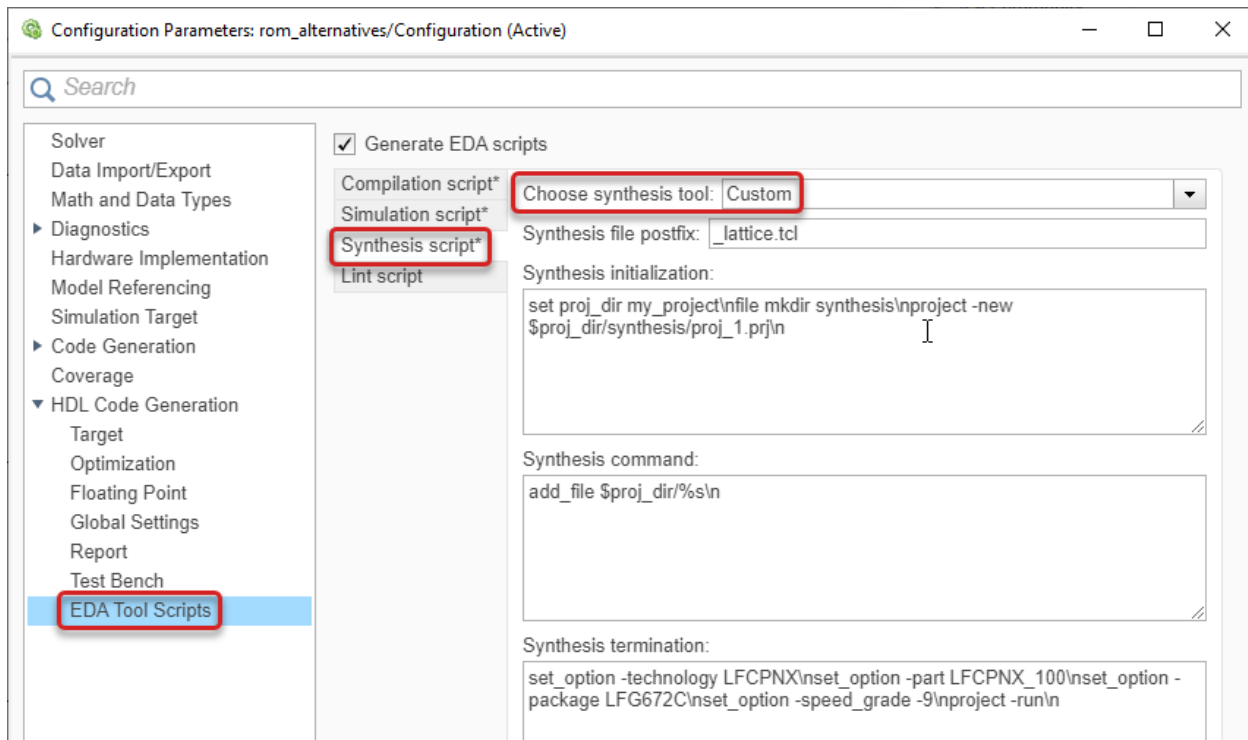
4.1.3 Register DUT I/Os for timing analysis
Add registers to the inputs and outputs of your DUT before performing post-synthesis timing analysis. This will ensure all elements in your design are included for timing analysis, and will give you an accurate critical path estimation.

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
- FIFO in classic or FWFT mode

Modeling guidelines:

- 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 some Intel FPGA devices, setting the parameter **RAM Architecture** to `Generic RAM without clock enable` may yield better mapping results. The parameter can be found under **Coding style** tab, on the **Global Settings** pane of the HDL Coder UI.
- For AMD FPGA devices, you can direct the use of specific RAM types via synthesis attributes. To do this, select the RAM block, click **HDL Block Properties** on the HDL Code Toolstrip, then set the parameter

27

RAMDirective to `distributed`, `block` or `ultra`. Note that UltraRAM mapping requires certain operating conditions and supported devices. See the [parameter documentation](parameter%20documentation) for more details.

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:
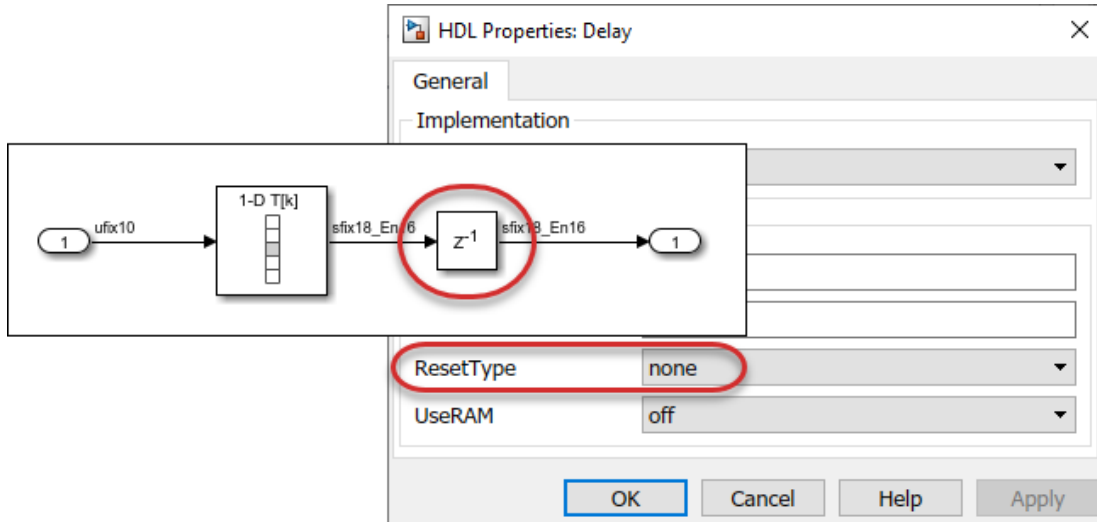
- Select the Delay block, click **HDL Block Properties** on the HDL Code Toolstrip, then set **ResetType** to `none`. Suppressing the reset on the Delay block will ensure best mapping results across different synthesis tools and FPGA devices. HDL Coder can also automate this delay insertion for you via the **Map lookup tables to RAM** parameter – see section 3.3.4 for details.

- 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)];
```

- Set the parameter **Diagnostic for out-of-range input** of the Lookup Table block to `Error`.

You may also model a ROM using a System object-based RAM block with initial value, like the example below. Use this approach when targeting Lattice FPGA devices. Follow the RAM block guidelines in section 4.2.1.
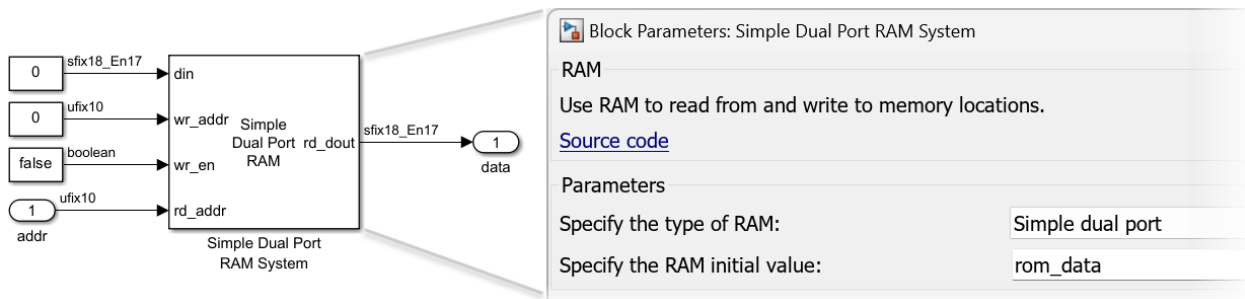
4.2.3 Additional tips

- Most FPGA RAM blocks support multiple aspect ratios. For example, an 18Kbit RAM can be configured as 9-bit x 2048, 18-bit x 1024, 36-bit x 512 etc. Exceeding the word size for a given aspect ratio, even by 1-2 bits, will double the RAM block usage on the FPGA - e.g. a 20-bit x 1024 RAM in Simulink are implemented using two 18Kbit FPGA RAM blocks. To optimize RAM resources, check the available RAM types and configurations of your target device, and stay within the supported aspect ratios.

- 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), co-simulation 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:
    - AMD Virtex™-6 and 7 series provide 18x25-bit signed multipliers, while the AMD UltraScale™ architecture provides 18x27-bit signed 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.
    - Lattice FPGA DSP blocks also support 9x9, 18x27 and 18x36-bit multiplications.
  - 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.
    - AMD FPGA 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.
    - For Lattice FPGA, the Nexus family DSP blocks provide 1 input register, 1 output register and 1 register between the multiplier and final adder.

4.3.2 Multiply
Example: multiply_rounding_example.slx

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

29

- o For Intel FPGA targets, use 1-2 input registers and 1 output register.
- o For Lattice FPGA targets, use 1 input register and 1 output register.
- Rounding support:
  - o The DSP blocks in many AMD FPGA families provide built-in support for convergent rounding after the multiplication. The rounding operation uses the post-adder and pattern detect logic within the DSP, as shown in the following example.
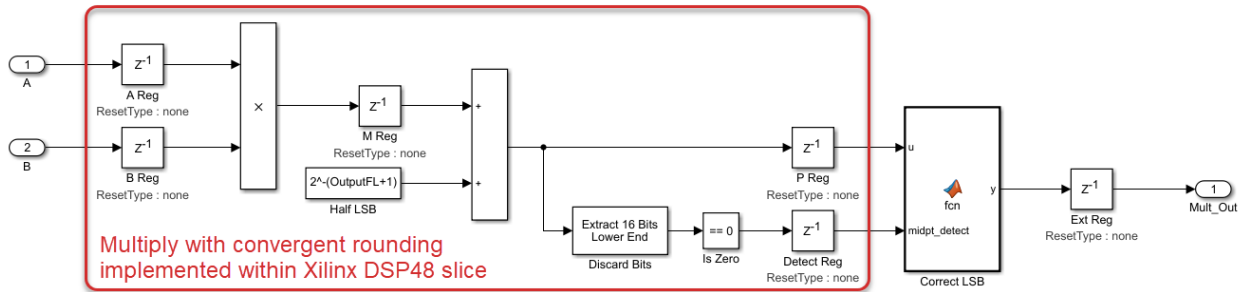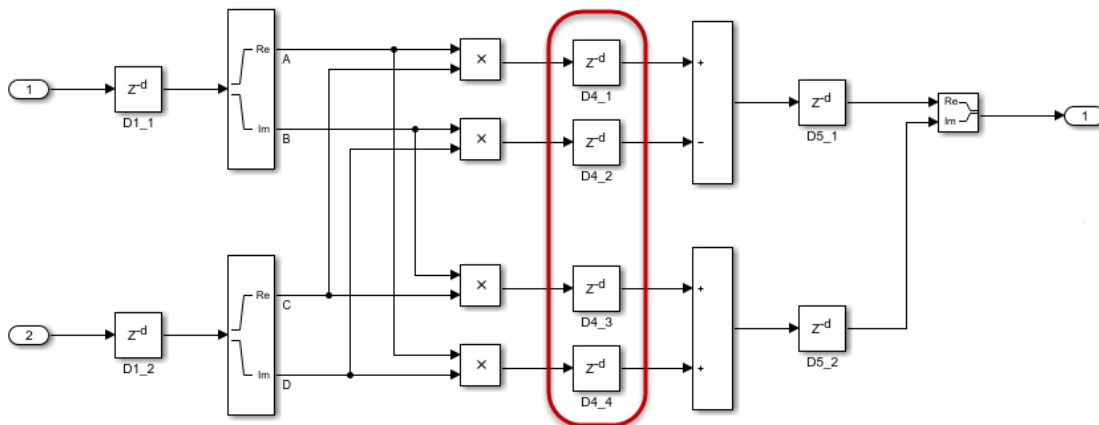

4.3.3 Complex multiply
Example: complex_multiply_example.slx

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:
  - o For AMD FPGA targets, use up to 2 input registers (D1), 1 multiplier output register (D4) and 1 output register (D5).
  - o 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).
  - o For Lattice FPGA targets, use 1 input register (D1), 1 multiplier output register (D4) and 1 output register (D5).

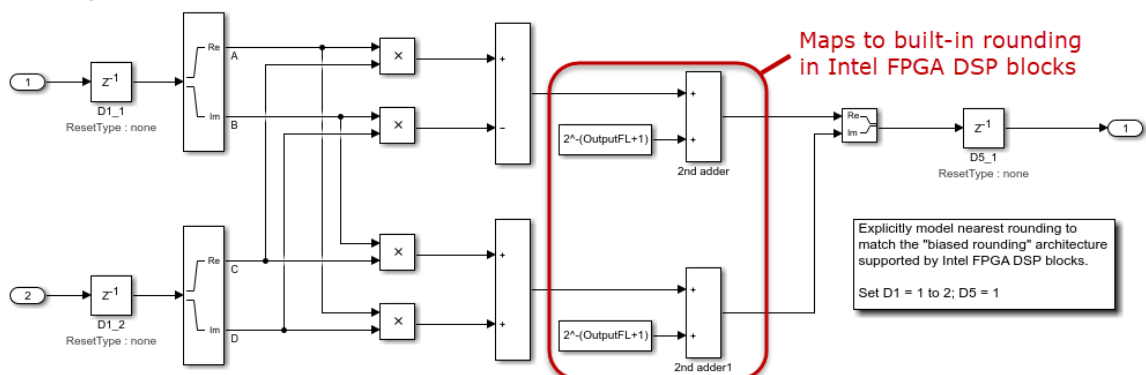A complex multiplier can also be implemented using 3 multipliers and 5 adders:

$$(A + Bj)*(C + Dj) = (Z - X) + (Y + Z)j, \text{ where } X = (A + B)*D, Y = (B - A)*C, Z = (C + D)*A$$

DSP 1: Z-(A+B)*D

DSP 2: Z+(B-A)*C

DSP 3: (C+D)*A = Z
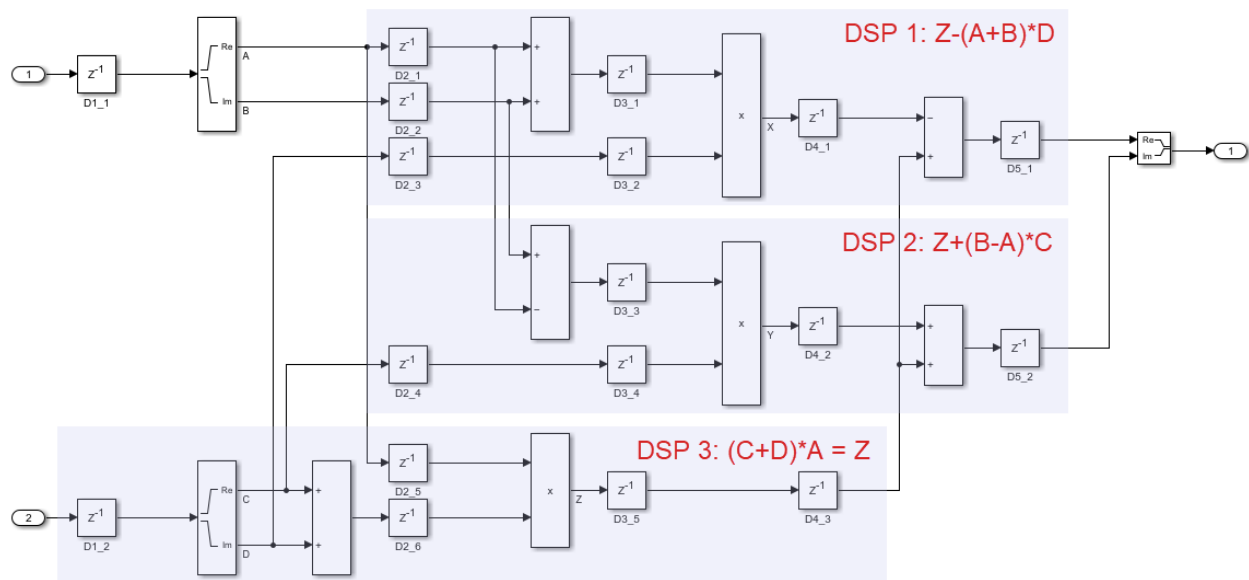
- For AMD FPGA targets, this architecture uses one less DSP block compared to the "4 multipliers, 2 adders" approach, by leveraging the pre-adders available in DSP. In exchange, it may require more register resources and higher latency. The maximum clock speed may also be reduced.

- 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.

- For Lattice FPGA targets, avoid this architecture. The "4 multipliers, 2 adders" approach results in better hardware utilization.

- Pipelining guidelines:
  - For AMD FPGA 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).

Rounding logic at the output of a complex multiply may be performed without incurring additional DSP resources, as demonstrated in the following examples.

- 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.

Maps to built-in rounding in Intel FPGA DSP blocks

Explicitly model nearest rounding to match the "biased rounding" architecture supported by Intel FPGA DSP blocks.

Set D1 = 1 to 2; D5 = 1

31

- For AMD FPGA devices, the "4 multipliers, 2 adders" architecture maps to 4 DSP blocks with unused adder resources. Those adders can be leveraged to implement nearest rounding by shifting the order of operations to the left, as shown in the example below.
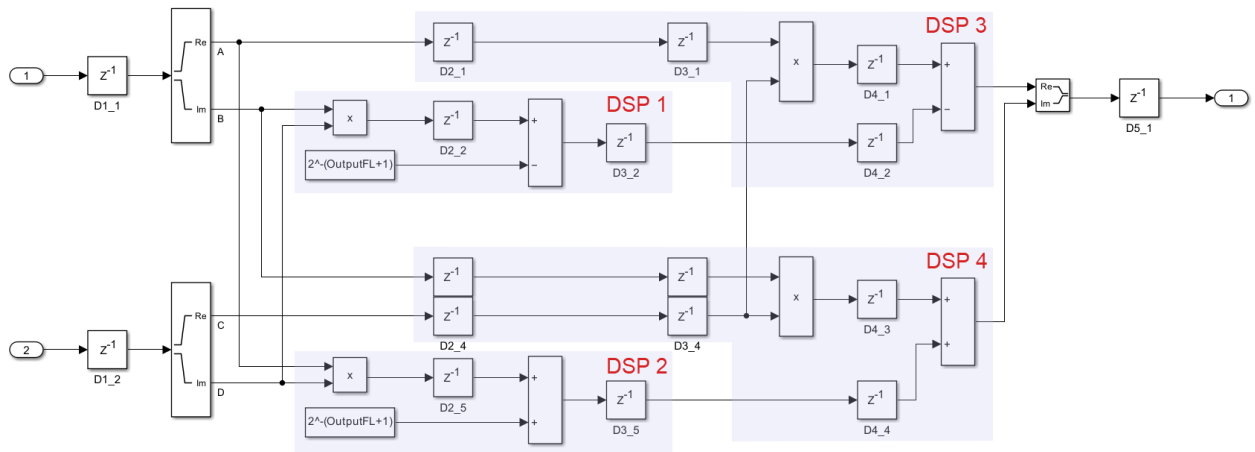
4.3.4 FIR filters and FFTs (DSP HDL Toolbox)

DSP HDL Toolbox (new since R2022a) provides signal processing IP blocks that are optimized for FPGA and ASIC targets. The blocks use architectures that leverage FPGA DSP and RAM resources efficiently, with built-in pipelining and data valid interface for versatile sample control. Compared to similar HDL-compatible blocks in DSP System Toolbox, DSP HDL Toolbox library blocks offer more options and superior hardware performance, while providing bit and cycle accurate simulation behavior in Simulink.

Follow the guidelines below to achieve best synthesis results:

- Reset type

  Most blocks provide a local, synchronous reset port if your design requires a way to reset internal states. The option **Use HDL global reset** is available for some blocks but not preferred, as it may produce large fanout and impact DSP block mapping. If you use it, follow the guidelines on proper global reset type in section 4.3.1. See the documentation for additional reset considerations.

- Word size

  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. You can control multiplier word size as followed:

  - For FIR filters, an M-bit input data and N-bit coefficient creates MxN-bit multipliers.
  - For FFT/IFFT using the streaming radix 2^2 architecture, an M-bit input data creates MxM-bit multipliers for all FFT stages if the **Divide butterfly outputs by two** option is used. This scaling option optimizes DSP block mapping at the cost of reduced precision. When the option is not used, the multiplier word size grows by 1-bit per stage.
  - For FFT/IFFT using the burst radix 2 architecture, an M-bit input data creates an MxM-bit multiplier if the **Divide butterfly outputs by two** option is used. Otherwise, the multiplier is PxP-bit, where P = M+log2(FFT size).

- Pipeline registers

  Optimization settings described in section 3.3 do not impact internal pipelining for DSP HDL Toolbox library blocks. However, the model target device setting may be used to determine optimal internal pipelining placement. You can 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**.

The following guidelines apply to the [Discrete FIR Filter](#) block in Simulink and the [FIR Decimation](#) / [FIR Interpolation](#) blocks in DSP System Toolbox. Note that the architecture and pipelining options affect the generated code only. If you prefer bit and cycle-accurate simulation behavior, use the FIR filter blocks in DSP HDL Toolbox (section 4.3.4).

- 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:

| Implementation			Implementation	
Architecture	Fully Parallel ▼		Architecture	Frame Based ▼
Implementation Parameters			**Implementation Parameters**	
AddPipelineRegisters	off ▼		MultiplierInputPipeline	0
MultiplierInputPipeline	0		MultiplierOutputPipeline	0
MultiplierOutputPipeline	0		AdderTreePipeline	0
InputPipeline	0		InputPipeline	0
OutputPipeline	0		OutputPipeline	0

- 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.

Block Parameters: Discrete FIR Filter	✕
Discrete FIR Filter	
Independently filter each channel of the input over time using an FIR filter. You can specify filter coefficients using either tunable dialog parameters or separate input ports, which are useful for time-varying coefficients.	
A DSP System Toolbox license is required to use a filter structure other than Direct Form.	

Main	Data Types
Coefficient source:	Dialog parameters ▼
Filter structure:	Direct form ▼
Coefficients:	Direct form
Input processing:	Direct form symmetric
Initial states:	Direct form antisymmetric
	Direct form transposed
	Lattice MA

  - For AMD FPGA 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 AMD FPGA devices, use up to 2 input pipelines, 1 multiplier input pipeline and 1 multiplier output pipeline.

- o 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:
  - o Use direct form structure; transposed is not supported.
  - o For AMD FPGA targets, use 1-2 multiplier input pipelines and 1 multiplier output pipeline.
  - o 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:
  - o For AMD 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.
  - o For Intel FPGA targets, use 1-2 multiplier input registers, 1 multiplier output register and 2 registers between levels of adder tree (**AdderTreePipeline**).
  - o 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:
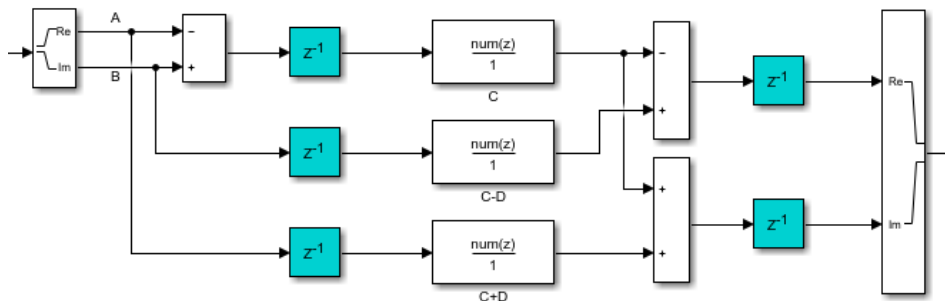
4.4 Using AXI Interfaces

Using the IP core generation workflow in HDL Workflow Advisor, you can map the data ports of your Simulink design to a variety of AXI4 interfaces, and generate an IP core that can be deployed to FPGA, SoC and ASIC devices. Additionally, you can deploy the IP core to supported SoC platforms and prototype your design with live data using MATLAB.

- Scalar, vector or bus data ports can be mapped to AXI4 or AXI4-Lite interface for lightweight data transfer. The ports are connected to memory-mapped registers with address offsets that are specified or allocated automatically. In addition:
  - o A design may not contain both AXI4-Lite and AXI4 interfaces. You may only choose one type.
  - o Scalar ports larger than 32-bit will be split into multiple 32-bit words automatically. A strobe register is also generated for synchronization.
  - o Each element of a vector/bus port is assigned a unique address. For vector ports, a strobe register is also generated to synchronize the vector read/write with the DUT logic.
  - o You may specify an initial value for input ports mapped to AXI4 registers.
  - o If you use **Enable readback on AXI4 slave write registers** with a design containing multiple AXI4 / AXI4-Lite outputs, you may also need to customize the setting **AXI4 slave port to pipeline register ratio** to achieve timing closure.
- Streaming data can be mapped to AXI4-Stream interface using a simplified data + valid protocol. The optional Ready and TLAST signals can be generated or modeled explicitly. The simplified protocol differs from the standard AXI4-Stream in the handshaking between the Ready and valid signals.
  - o Use vector to implement bit width greater than 128 bits, which is the limit of Simulink fixed-point support. For example, 512-bits can be modeled as a [4x1] 128-bit vector.

- You can control the packing of vector and complex data using the **Sample packing dimension** and **Packing mode** parameters.
- The generated Ready logic responds to backpressure by disabling the DUT logic, which may not be suitable for some designs. When modeling the Ready signal explicitly, make sure the model optimization settings do not interfere with the signal timing. See Appendix A for a more detailed explanation and examples.
- TLAST is used to mark the end of a frame. If you do not model TLAST explicitly for AXI4-Stream Master, a counter is generated to monitor the valid signal and assert TLAST after a fixed, programmable frame size. Model TLAST explicitly to customize its behavior, e.g. if a frame transfer is interrupted due to backpressure, you can assert TLAST early to resynchronize the frame boundary.

- For video applications, you can use the streaming pixel protocol provided by the [AXI4-Stream Video interface](#). The above guidelines regarding AXI4-Stream Ready signal also apply to AXI4-Stream Video.

- The [AXI4 master interface](#) provides a simplified protocol for designs that require data access to / from external memory. Data width up to 1024 bits is supported.
  - Use vector to implement bit width greater than 128 bits, which is the limit of Simulink fixed-point support. For example, 512-bits can be modeled as a [4x1] 128-bit vector.
  - The addresses for the DUT ports **rd_addr** and **wr_addr** are added to a programmable base address register to form the full AXI4 Master address.
  - Use a data FIFO together with your read / write control logic to properly handle backpressure. Before initiating a burst, make sure the FIFO has sufficient space for the entire burst. This prevents a stall mid-burst, which locks out other AXI4 masters from accessing the target slave device.

- Using HDL Verifier, you can interact with AXI4-Lite registers, RAM blocks and external memory from MATLAB during real-time operations. See section 5.5 for more information.

5. Verification Workflow

Design verification is critical to any hardware design; it is also time-consuming and labor-intensive. This section describes tools that can help you achieve verification goals in different stages of your design flow, from system-level simulations in Simulink, RTL test benches to FPGA hardware debugging.

5.1 Design verification tools in Simulink

Design mistakes are harder to debug and fix when they are found late in a development cycle. Thoroughly test your Simulink model before generating code help prevent costly mistakes from escaping downstream and reduce your overall design costs.

5.1.1 General tips

- A reference algorithm in MATLAB is often frame/vector based, while a Simulink design that implements the algorithm for hardware would only process one or a few samples per time step. When comparing the Simulink design against its MATLAB reference:
  - The Signal From Workspace block can be used to easily convert a large vector in MATLAB to a streaming data source in Simulink.
  - Simulink signal outputs can be captured as MATLAB vectors using To Workspace blocks or signal logging. Logged signals can also be viewed as digital waveforms in the Logic Analyzer Scope (see next section) or compared in the Simulink Data Inspector.
- Implementing hardware architecture before converting a design to fixed-point, where it is feasible to do so, allows the hardware architecture to be verified without quantization noise obscuring potential design errors.
- If a design contains data valid interface, insert invalid sample gaps to verify the data valid signal path. See section 2.1.5 for an example.
- Try accelerator modes in Simulink to improve simulation performance for large models or long / iterative simulations.

5.1.2 Logic Analyzer scope
Example: view_wave_example.slx

The Logic Analyzer scope allows you to easily view Simulink signals as digital waveforms. To use the scope, select and **log** one or more signals using the Simulink Toolstrip or pop-up cue. After simulating the model, click the **Logic Analyzer** button to view the waveforms.

The Logic Analyzer scope is available with DSP System Toolbox, HDL Verifier, or SoC Blockset.
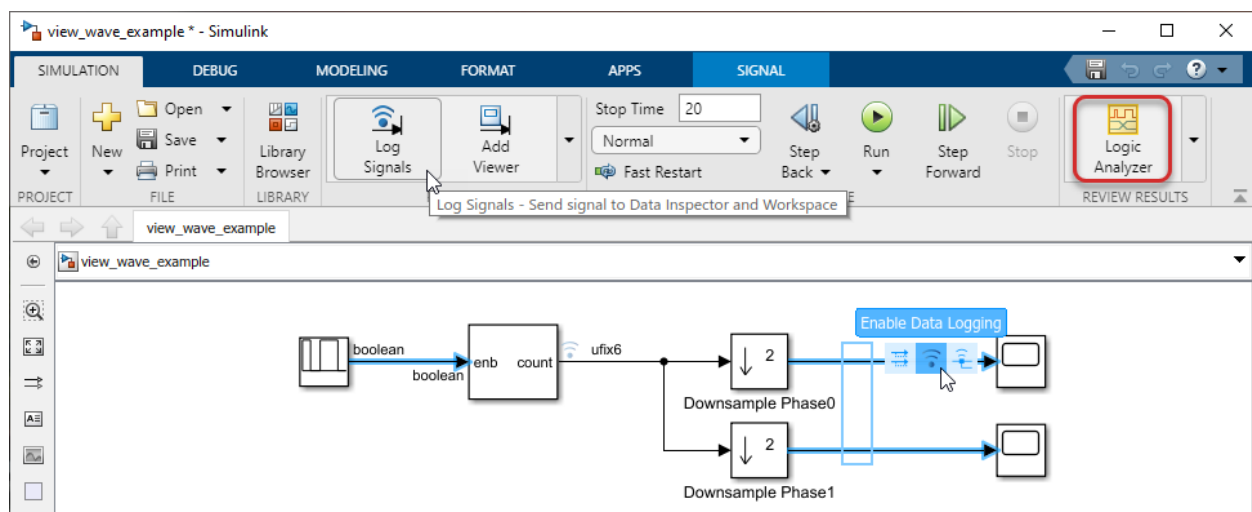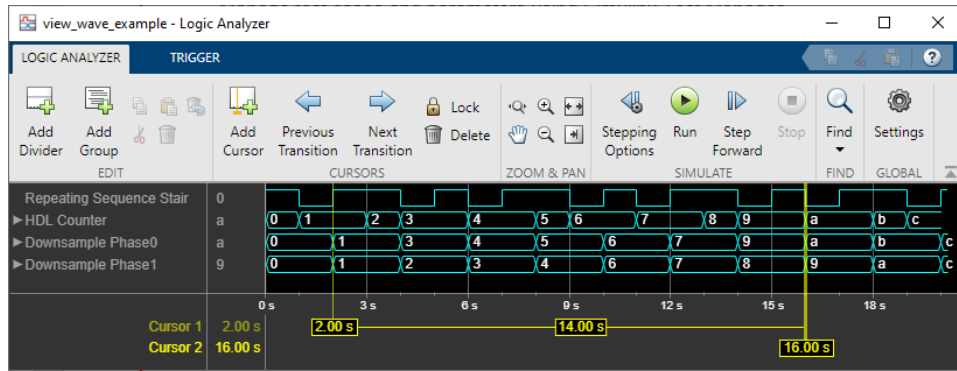
5.1.3 Validation & Verification Tools

- As your Simulink designs mature and testing needs grow, you can use Simulink Test to author, execute and organize test cases and test harnesses for your DUT. You can scale up regression testing with batch workflow and parallel execution, as well as integrate with continuous integration (CI) systems.

- Simulink Coverage can track how well your tests exercise your model, and merge the results from multiple runs. Tracing coverage is used to show what parts of your design are not exercised by the current suite of tests. It can also show which tests are only covering functionality that is already covered by other tests, so you can possibly eliminate some tests that are not contributing toward your overall verification goals.

- See the verification & validation solutions page for additional tools that help you link requirements to designs and tests, comply with high-integrity standards and more.

5.2 Verifying generated HDL with automatic test bench

Once you have generated HDL code from your Simulink model, you can use the same test cases you created to test your Simulink design to verify the generated HDL code.

5.2.1 Test bench options

- HDL test bench

  When you generate an HDL test bench, the generated model (described in 3.4.3) is simulated to log the input and output values of the DUT. When the test bench is run in an HDL simulator, a PASS or FAILED message will be printed to indicate whether the HDL outputs match the logged outputs from Simulink.

- SystemVerilog DPI test bench (requires HDL Verifier)

  A SystemVerilog DPI test bench is similar to an HDL test bench. However, instead of logging data from Simulink, C code is generated from the model to provide stimulus and reference output for the HDL DUT. This method greatly reduces test bench generation time for models with long simulation time, as shown in the example Verify HDL Design Using SystemVerilog DPI Test Bench.

- HDL co-simulation (requires HDL Verifier)

  A co-simulation model automatically compares the output between the generated model and the generated HDL that is being simulated in a supported HDL simulator. The HDL code being simulated appears as a co-simulation block in Simulink. Any mismatch between the generated model and HDL outputs will trigger a simulation assertion.

- FPGA-in-the-Loop / FIL (requires HDL Verifier)

  FIL is hardware co-simulation. Like HDL co-simulation, a FIL model automatically compares the output between the generated model and the generated HDL that is compiled and run on an FPGA board. Any mismatch will trigger an assertion. Communication with the FPGA is done via JTAG, Ethernet or PCI Express, depending on the hardware used.

  Note: In a FIL simulation, the hardware DUT I/O connects directly to the Simulink model, and the DUT is only enabled in a clock cycle when Simulink is ready to transfer data. See section 5.5 for other options to interact with a free-running FPGA design from MATLAB.

For additional comparison among these test bench options and their product requirements, see the documentation Choose a Test Bench for Generated HDL Code.

37

- Generate HDL / SystemVerilog DPI test bench or a co-simulation model using one of the following methods:
  - o On the **HDL Code** Toolstrip, click or pull down the **Generate Testbench** option after generating HDL code for the DUT.
  - o Open the **Test Bench** pane of the HDL Coder UI. Select the test bench type, set **Simulation tool** to `Mentor Graphics Modelsim`, `Cadence Incisive` or `Xilinx Vivado Simulator`. Click **Apply**, then click **Generate Test Bench**. For a detailed example, see Verify Generated Code Using HDL Test Bench from Configuration Parameters.
  - o In HDL Workflow Advisor, select the test bench type as part of the `HDL Code Generation Settings` in **3.1 Set HDL Options**, enable `Generate test bench` in **3.2 Generate RTL Code and Testbench,** then run the task. If you choose to generate a co-simulation model, step **3.3 Verify with HDL Cosimulation** will appear, giving you the option to automatically verify the HDL co-simulation outputs.
- Generate an FPGA-in-the-Loop model in HDL Workflow Advisor:
  - o Set **Target workflow** to `FPGA-in-the-Loop` in **1.1 Set Target Device and Synthesis Tool**. Subsequent steps are updated automatically to prepare for FIL simulation.
  - o See the tutorial FIL Simulation with HDL Workflow Advisor for Simulink for a detailed example.
- Note: test bench generation is disabled when the top-level model is configured as DUT, as described in section 1.3.

5.3 Co-simulating hand-written HDL code

In addition to verifying generated code, co-simulation can also be used to import hand-written HDL into Simulink, so that you may qualify the functionality of existing IP blocks and incorporate them into system-level verification. Co-simulating hand-written HDL uses HDL Verifier only; HDL Coder is not required.

- Use the Cosimulation Wizard to set up a co-simulation model. The wizard guides you through all necessary steps such as connecting to your HDL simulator, compiling the HDL files, configuring the I/O ports, mapping timescale between Simulink and the HDL simulation etc. See the tutorial Verify Raised Cosine Filter Design Using Simulink for a detailed example.

  Beginning in R2022b, you may also set up co-simulation via scripting. See cosimulationConfiguration for more information.

- Upon completing the wizard steps, the information you use to configure the co-simulation model are saved to a .mat file. You can launch the wizard using the saved information with the following command:

  ```
  >> cosimWizard('cosimWizard_DUT.mat')
  ```

- The following files produced by the wizard are needed for each co-simulation:
  - o A model containing the generated co-simulation block
  - o For co-simulation with AMD Vivado Simulator:
    - ▪ The `xsim.dir` folder and its content
  - o For co-simulation with Siemens® Questa® / ModelSim® and Cadence® Xcelium™:
    - ▪ `compile_hdl_design_hlp_<DUT name>.m` is used to compile the HDL files
    - ▪ `launch_hdl_simulator_<DUT name>.m`: is used to launch the HDL simulator for simulation
    - ▪ `hdlverifier_compile_design.do`/`.sh` contains the HDL compilation commands
    - ▪ `parameter_<DUT name>.cfg` is used to configure HDL parameters (generics)
- Note that absolute file paths are used in the .do/.sh script. If you plan to move the co-simulation files to a different folder or machine, modify the file paths accordingly.
- To use multiple co-simulation blocks in the same Simulink model, create an HDL wrapper for the individual modules as shown in the example Manchester Receiver Using Multiple Cosimulation Blocks. This prevents each co-simulation block from launching a separate instance of the HDL simulator.

5.4 Beyond test vector comparisons: generating SystemVerilog DPI and UVM test bench components

Engineers often verify RTL design by logging test vectors and expected outputs from the reference model in MATLAB or Simulink and comparing the logged values against the RTL outputs. While this works well for unit tests, verification of large system designs usually requires additional approaches such as constrained randomization to improve test coverage, and more analytical test benches to prove that system requirements are met. These additional tasks are typically labor-intensive and time-consuming.

You can model complex test bench elements in MATLAB and Simulink, then use HDL Verifier to accelerate and automate verification tasks via C code generation. Leveraging the Direct Programming Interface (DPI) in SystemVerilog, HDL Verifier allows you to easily integrate the generated C code with your verification environment, from VHDL, Verilog or SystemVerilog test benches to Universal Verification Methodology (UVM) environment. This technology can help improve your verification workflow in many ways, such as:

- Create a reference model to verify against a hand-coded design directly, without the need to log expected output vectors and maintain them.
- Reuse existing verification logic in MATLAB or Simulink, instead of developing them from scratch.
- Create parameterizable and reusable test bench components, including the generation of constrained random stimulus to improve verification coverage.

Generating SystemVerilog DPI and UVM components from MATLAB requires MATLAB Coder; generating SystemVerilog DPI components and UVM test bench from Simulink requires Simulink Coder.

5.5 Performing interactive testing on free-running FPGA and SoC boards

Hardware testing is an essential step in FPGA and SoC development. Programming memories, monitoring internal signals, collecting and verifying data samples as your design processes real-time data are some of the typical tasks. Using HDL Verifier, you can perform these tasks within MATLAB and leverage a variety of additional tools in your interactive hardware tests.

- FPGA data capture lets you capture a window of signal data from an FPGA / SoC board to MATLAB or Simulink over JTAG or Ethernet. You can view the captured signals in the Logic Analyzer scope (described in 5.1.2), or further process the data with MATLAB functions or Simulink blocks. Data capture can be configured interactively using the UI tool, or programmatically using the System Object API.

  You can use FPGA data capture with hand-written RTL design or generated code from HDL Coder, as demonstrated in the following examples. See the documentation for additional details.
    - Read Temperature Sensor Data from AMD FPGA Board
    - Read Audio Signal from Intel FPGA Board
    - Debug IP Core Using FPGA Data Capture

- Using the AXI Manager IP, you can read from and write to memory locations on your FPGA / SoC board, including AXI-Lite registers, RAM blocks and external DDR memories connected via an AXI memory controller. This allows you to interact with your hardware, such as program command registers, filter coefficients, and read/write image data directly within MATLAB.

  AXI Manager is supported over JTAG, Ethernet or PCI Express interface, depending on target hardware. You can use the feature with hand-written RTL design or generated code from HDL Coder, as demonstrated in the following examples.
    - MATLAB as AXI Manager with AMD FPGA and SoC Boards
    - Access FPGA External Memory on Intel FPGA Board

- FPGA data capture and MATLAB AXI Manager are supported for a range of FPGA devices. For details on hardware and interface support, see the Xilinx FPGA Board Support from HDL Verifier, Intel FPGA Board Support from HDL Verifier and Microchip FPGA Board Support from HDL Verifier documentations.

Appendix A. Modeling AXI4 Backpressure for IP Core Generation

This section describes the best practices in configuring HDL Coder optimization settings when your design contains an AXI4-Stream interface. While the guidelines uses AXI4-Stream as an example, they are also applicable for other interfaces with similar backpressure signaling.

A.1.1 AXI4-Stream ready signal

In the AXI4-Stream protocol, TREADY is used by the destination block to indicate whether it can accept a transfer in the current cycle. For example, a destination DMA engine may apply backpressure to its data source from time to time when it is unable to write to memory temporarily; a downstream IP block may require time to perform computations before it is ready to process additional data samples.

The Ready signal in the simplified streaming protocol serves the same purpose as AXI4-Stream TREADY, although it behaves slightly differently. If you do not model Ready in your design, HDL Coder automatically generates the signal and logic that responds to downstream backpressure by lowering the DUT clock enable. Since this implementation essentially disables all DUT logic, it may not be suitable for your design. In that case, you can model the handshaking explicitly according to the following timing diagram:
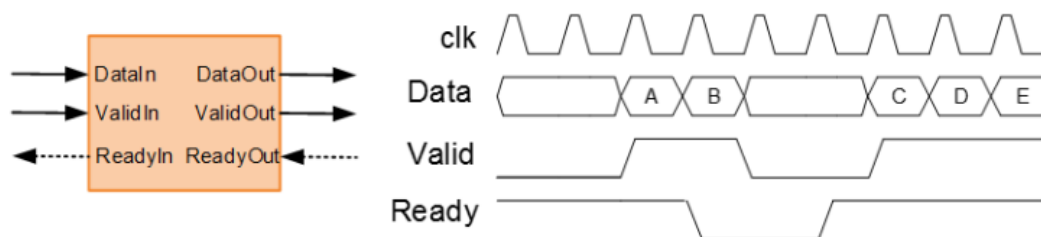
Specifically, valid must be de-asserted at most one clock cycle after downstream ready is de-asserted. In addition, any data samples produced within your design must be properly stored until the backpressure is released. This can be accomplished using a First-Word-Fall-Through (FWFT) FIFO, as shown below.
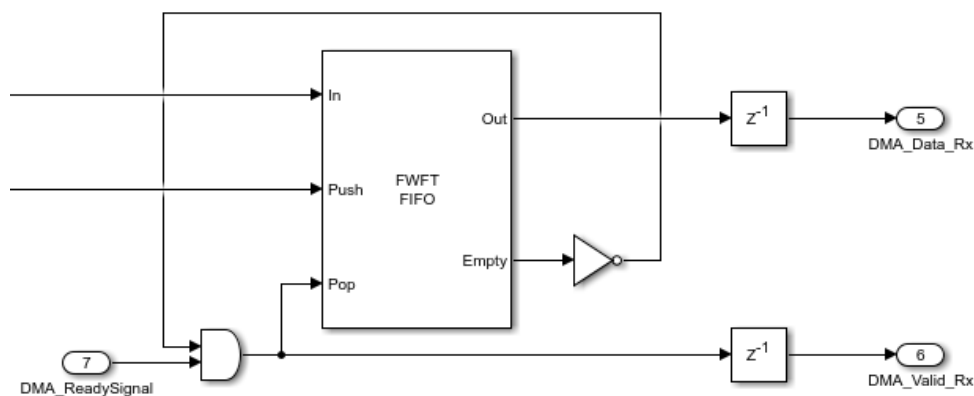
A.1.2 Effects of model-level optimization settings
Example: txrx_global_optim_example.slx

As described in section 3.3.4, several automatic pipelining options are applied at the model-level. For example, when *Adaptive Pipelining* introduces pipeline registers to select design elements such as multipliers, *Delay Balancing* adds matching registers to parallel data paths in order to maintain the behavior of your design. These optimization features help improve timing performance based on your hardware target settings.

However, model-level optimizations are applied to all signal paths within the DUT, including the AXI4-Stream valid and ready signals mentioned above. As a result, the strict timing requirements between these signals may be inadvertently violated, as shown in the following example.
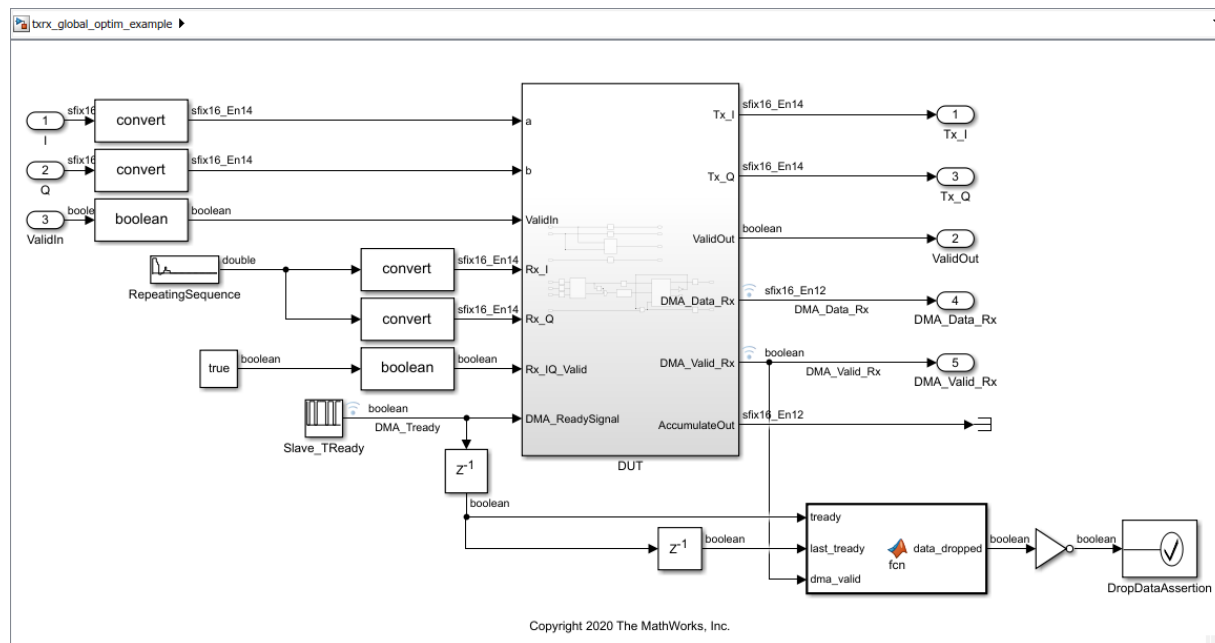
The example is a basic peak detector design that moves data only when a certain threshold is crossed. Ready is driven by a repeating, pseudo-random TRUE/FALSE sequence. Using the Logic Analyzer scope, you can observe the valid signal lowers 1 cycle after Ready is de-asserted, which conforms to the required protocol. A FIFO stores incoming data samples until the backpressure is released and data transfers can be resumed.
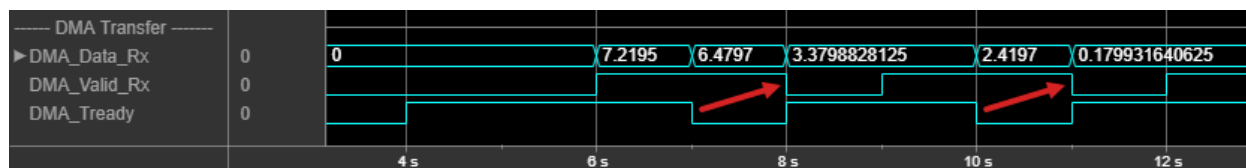
The design contains assertion logic that detects any violation of the valid/ready protocol, and stops the simulation if that occurs. Although the model simulates successfully without any assertion, the generated HDL design will fail using the default optimization settings.

The best way to understand the problem is by creating the optional optimization report (described in 3.4.2) when you generate the HDL code, and reviewing the generated model (described in 3.4.3). The option **Generate optimization report** can be found in the model Configuration Parameters under **HDL Code Generation > Report**.

- Generate HDL code for the DUT subsystem.
- Open the generated model that gets created during code generation, using the following command:
  ```
  >> gm_txrx_global_optim_example
  ```
  Once closed, you can locate it again in the *hdl_prj/hdlsrc/txrx_global_optim_example* folder.
- Simulate the generated model. You will encounter the following error:

41

Let's trace the error by performing the steps below.

- Log the following DMA signals in the generated model:
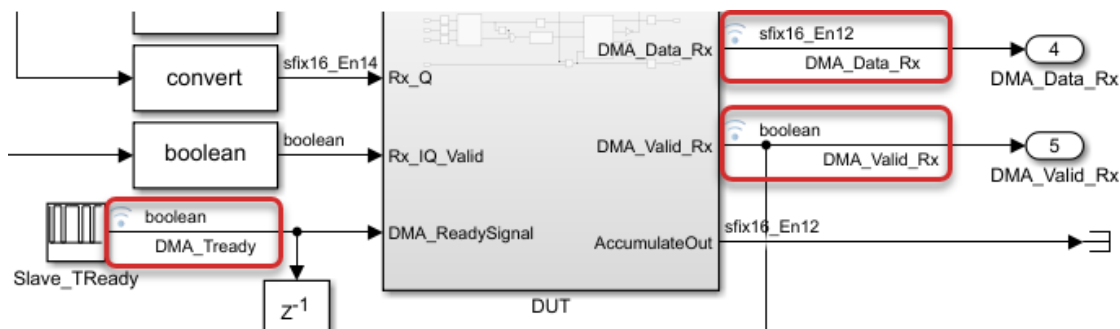
- Open the assertion block and uncheck **Stop simulation when assertion fails**.
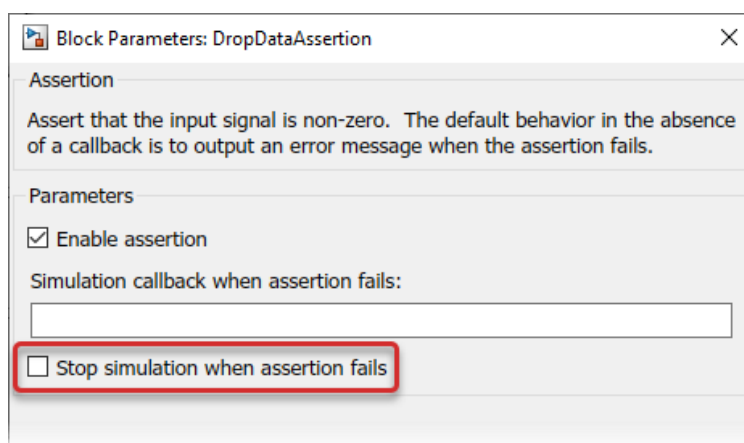
- Re-simulate the model, then open the Logic Analyzer scope and examine the signals at time 8. Unlike the original model, the valid signal stays high after Ready de-asserts in the previous clock cycle. Since the generated model is cycle-accurate to the generated HDL code, the HDL design will similarly fail.
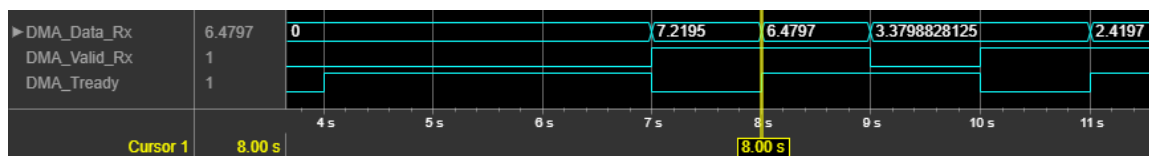
Further examination of the optimization report and generated model reveals the source of the problem. In the Receiver subsystem shown below, pipeline registers (colored in orange) are inserted around the multipliers as a result of *Adaptive Pipelining.*
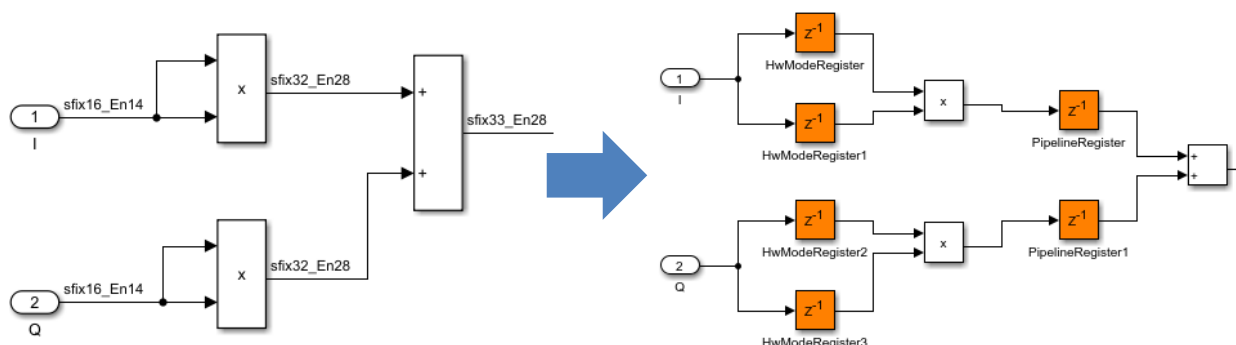
Matching pipeline registers are also inserted to data paths parallel to the Receiver subsystem via *Delay Balancing*. The additional delay to the Ready signal is the cause of the observed timing malfunction.
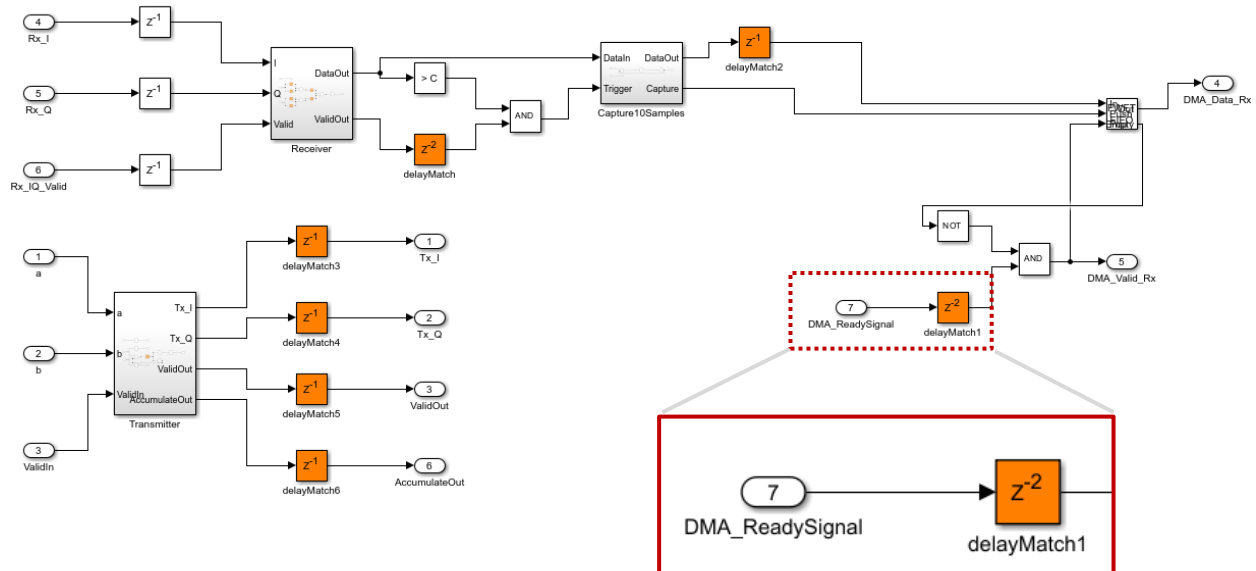
You can navigate the generated model and review the full list of inserted registers using the optimization report.

A.1.3 Localizing optimizations to algorithm blocks
Example: txrx_local_optim_example.slx

In the previous example, *Delay Balancing* is applied across the entire model, affecting parts of the design that didn't require automated pipelining. This example illustrates how you can localize the optimization to where it is needed.

In this example, the HDL Block Property **BalanceDelays** is disabled for the DUT subsystem and enabled for the Transmitter and Receiver subsystems. This instructs HDL Coder to insert pipelines locally to each subsystem.
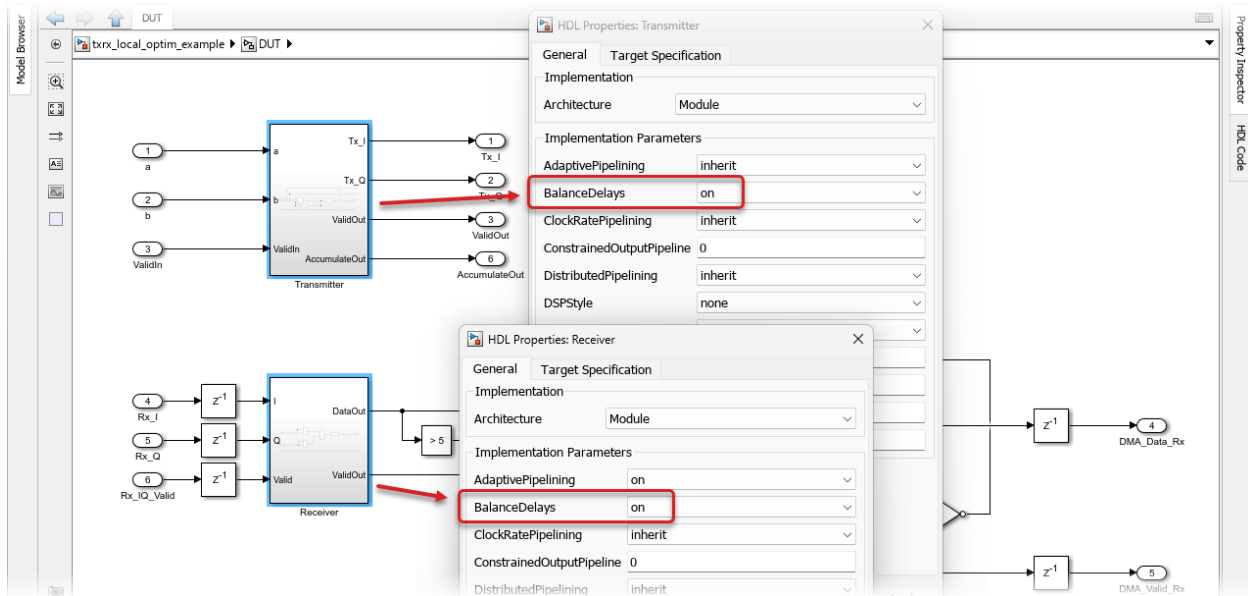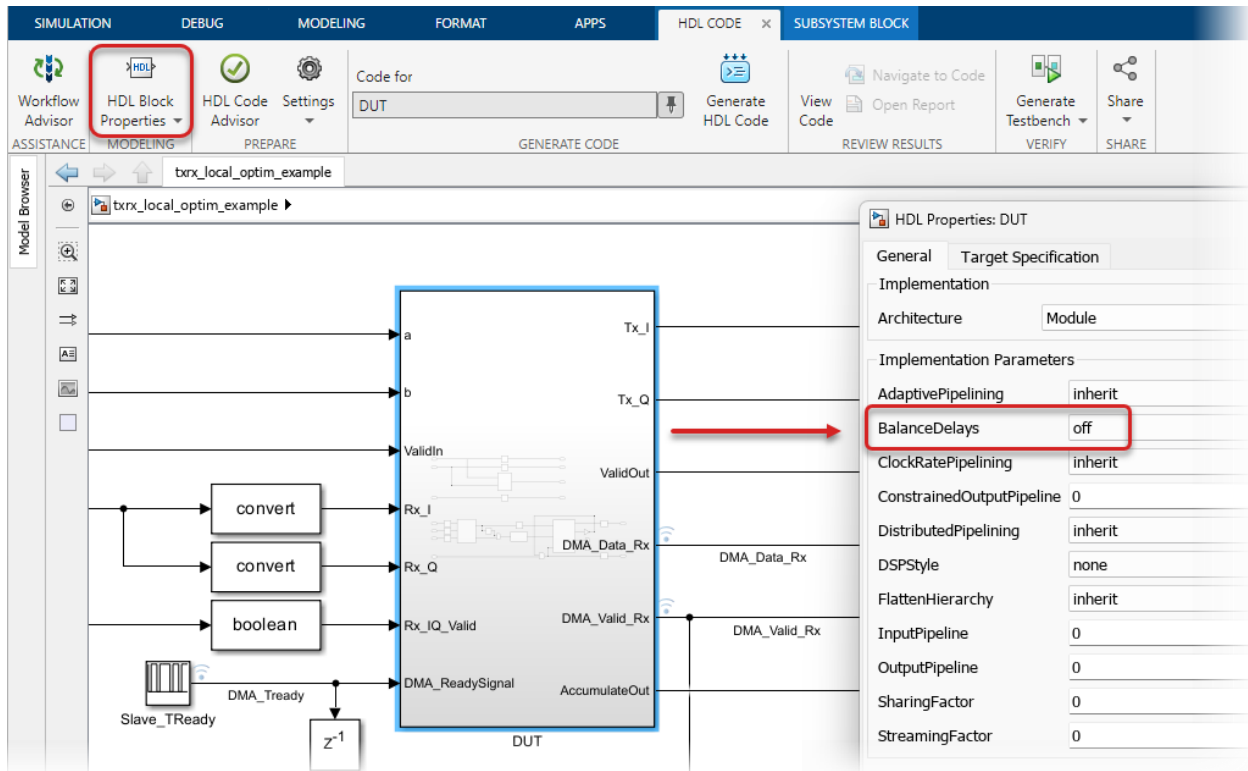
In addition, **Check for presence of unbalanced delays in the generated HDL code** has been changed from `Error` (default value) to `Warning`. The parameter can be found in the HDL Coder UI under **Global Settings** > **Additional settings** > **Advanced**.
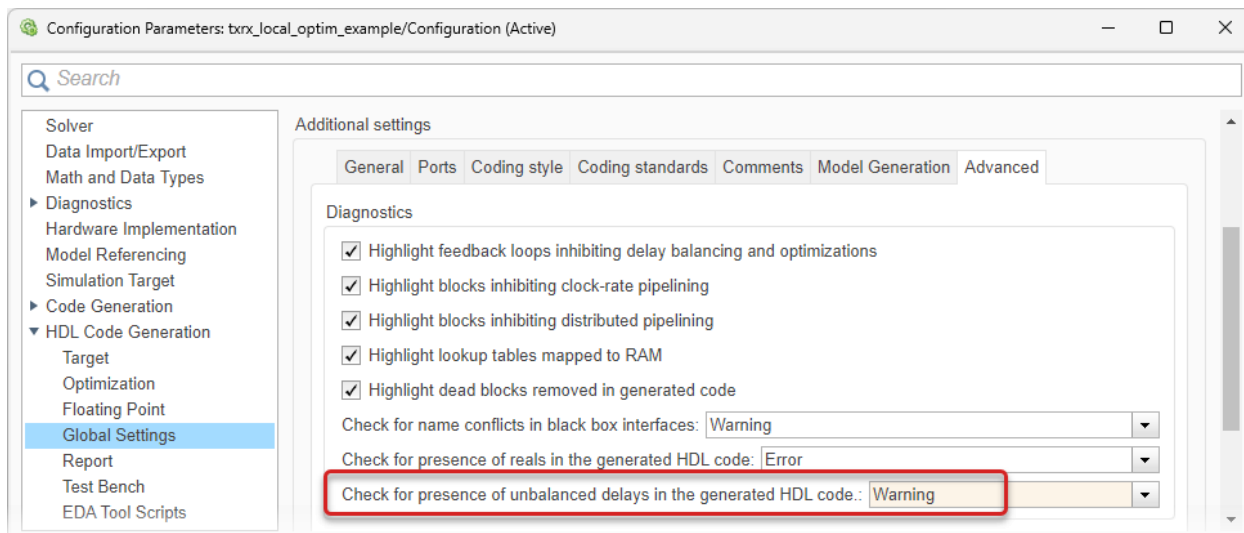
- Generate HDL code for the DUT subsystem and open the generated model as before.

  ```
  >> gm_txrx_local_optim_example
  ```

- Observe that the Ready signal is unchanged, while pipeline registers are added inside the Receiver subsystem.
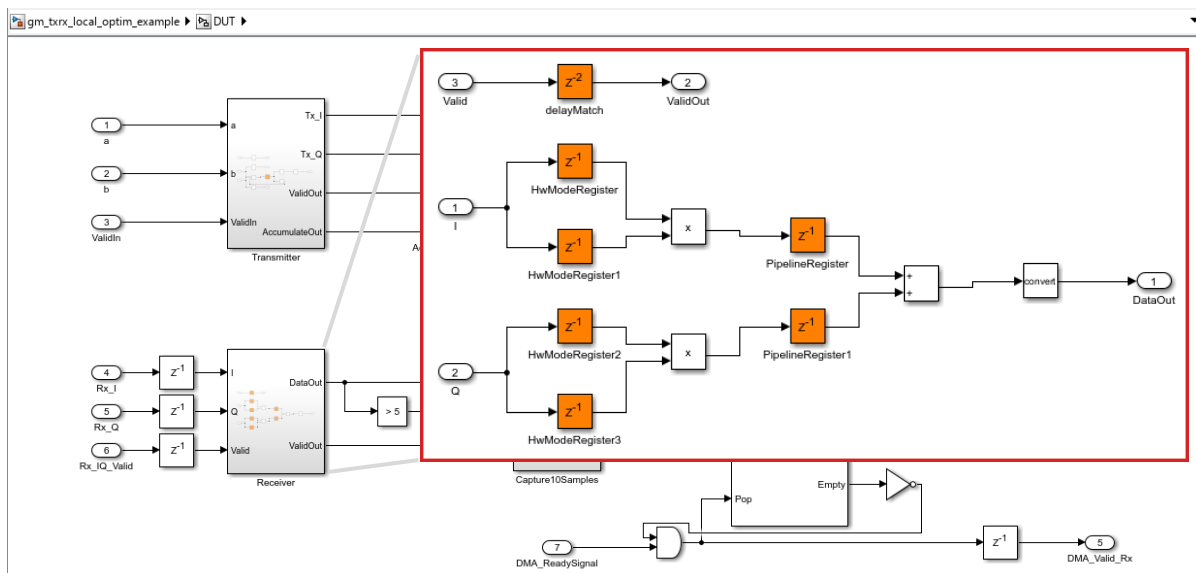
- Simulate the generated model and confirm that it runs without any assertion error.
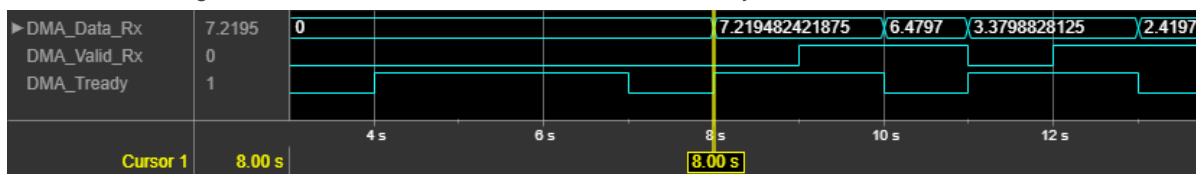
Delay Balancing has been successfully localized to the Transmitter and Receiver subsystems, without affecting the AXI4-Stream valid/ready signal protocol in the design.