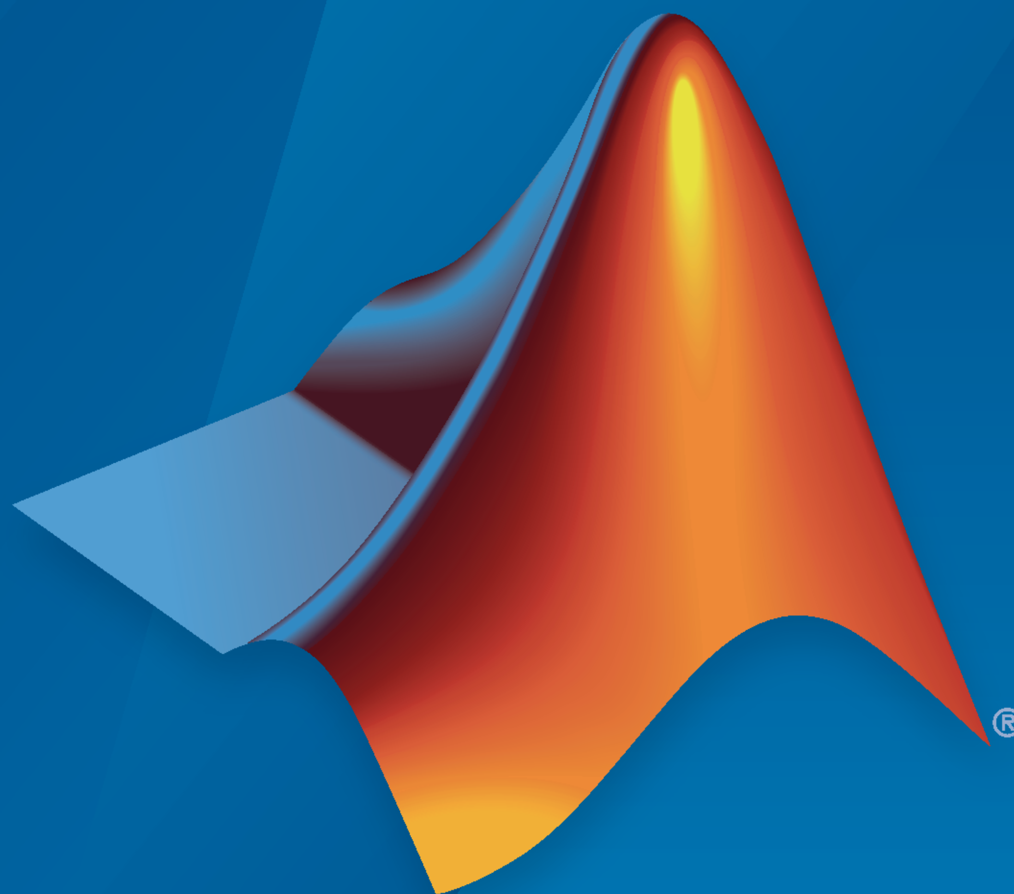
HDL Coder™

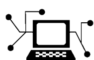
Getting Started Guide



MATLAB® & SIMULINK®

R2022a

How to Contact MathWorks



Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

HDL Coder™ Getting Started Guide

© COPYRIGHT 2012–2022 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

March 2012	Online only	New for Version 3.0 (Release 2012a)
September 2012	Online only	Revised for Version 3.1 (Release 2012b)
March 2013	Online only	Revised for Version 3.2 (Release 2013a)
September 2013	Online only	Revised for Version 3.3 (Release 2013b)
March 2014	Online only	Revised for Version 3.4 (Release 2014a)
October 2014	Online only	Revised for Version 3.5 (Release 2014b)
March 2015	Online only	Revised for Version 3.6 (Release 2015a)
September 2015	Online only	Revised for Version 3.7 (Release 2015b)
October 2015	Online only	Rereleased for Version 3.6.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 3.8 (Release 2016a)
September 2016	Online only	Revised for Version 3.9 (Release 2016b)
March 2017	Online only	Revised for Version 3.10 (Release 2017a)
September 2017	Online only	Revised for Version 3.11 (Release 2017b)
March 2018	Online only	Revised for Version 3.12 (Release 2018a)
September 2018	Online only	Revised for Version 3.13 (Release 2018b)
March 2019	Online only	Revised for Version 3.14 (Release 2019a)
September 2019	Online only	Revised for Version 3.15 (Release 2019b)
March 2020	Online only	Revised for Version 3.16 (Release 2020a)
September 2020	Online only	Revised for Version 3.17 (Release 2020b)
March 2021	Online only	Revised for Version 3.18 (Release 2021a)
September 2021	Online only	Revised for Version 3.19 (Release 2021b)
March 2022	Online only	Revised for Version 3.20 (Release 2022a)

Contents

1 About HDL Coder

HDL Coder Product Description 1-2
Key Features 1-2

HDL Language Support and Supported Third-Party Tools and Hardware 1-3
VHDL and Verilog Language Support 1-3
Third-Party Synthesis Tools and Version Support 1-3
FPGA-in-the-Loop Hardware 1-3
Generic ASIC/FPGA Hardware 1-4
IP Core Generation Hardware 1-5
Simulink Real-Time FPGA I/O: Speedgoat Target Computer 1-5
FPGA Turnkey Hardware 1-6

2 Getting Started with HDL Coder

Tool Setup 2-2
Synthesis Tool Path Setup 2-2
HDL Simulator Setup 2-3
Xilinx System Generator Setup for ModelSim Simulation 2-3
Altera DSP Builder Setup 2-4
FPGA Simulation Library Setup 2-4
C/C++ Compiler Setup 2-5

3 Tutorials

Basic HDL Code Generation Workflow 3-2

Create HDL-Compatible Simulink Model 3-3
Use Blank DUT Template 3-3
Choose Blocks from HDL Coder Library 3-4
Develop Algorithm for DUT 3-5
Create Test Bench for Design 3-6
Simple Counter Model 3-6
Simulate and Verify Design Functionality 3-7
Generate HDL Code from Simulink Model 3-8

Generate HDL Code from Simulink Model **3-9**
Models for HDL Code Generation **3-9**
Simple Counter Model **3-9**
Generate HDL Code **3-10**
View HDL Code Generation Files **3-11**
Inspect Generated HDL Code **3-12**
Validate HDL Behavior Using Validation Model **3-14**
Verify Generated HDL Code **3-15**

Verify Generated HDL Code from Simulink Model **3-16**
What is an HDL Test Bench? **3-16**
Simple Counter Model **3-16**
Verification Methods **3-17**
Generate HDL Test Bench **3-17**
View HDL Test Bench Files **3-18**
Run Simulation and Verify Generated HDL Code **3-18**
Deploy Generated HDL Code on Target Device **3-19**

HDL Code Generation and FPGA Synthesis from Simulink Model **3-21**
Simulink HDL Workflow Advisor **3-21**
Simple Counter Model **3-21**
Set Up Tool Path **3-22**
Open the HDL Workflow Advisor **3-22**
Generate HDL Code **3-23**
Perform FPGA Synthesis and Analysis **3-24**
Run Workflow at Command Line with a Script **3-25**

Generation of Clock Bundle Signals in HDL Coder **3-26**
MATLAB Code and Clock Relationship **3-26**
Simulink Model and Clock Relationship **3-27**

Get Started with MATLAB to HDL Workflow **3-29**

Basic HDL Code Generation and FPGA Synthesis from MATLAB **3-35**

Generate HDL Code from MATLAB Code Using the Command Line Interface **3-42**

Generating Modular HDL Code for Functions **3-46**

System Design with HDL Code Generation from MATLAB and Simulink **3-53**

About HDL Coder

- “HDL Coder Product Description” on page 1-2
- “HDL Language Support and Supported Third-Party Tools and Hardware” on page 1-3

HDL Coder Product Description

Generate VHDL and Verilog code for FPGA and ASIC designs

HDL Coder generates portable, synthesizable VHDL® and Verilog® code from MATLAB® functions, Simulink® models, and Stateflow® charts. The generated HDL code can be used for FPGA programming or ASIC prototyping and design.

HDL Coder provides a workflow advisor that automates the programming of Xilinx®, Microsemi®, and Intel® FPGAs. You can control HDL architecture and implementation, highlight critical paths, and generate hardware resource utilization estimates. HDL Coder provides traceability between your Simulink model and the generated Verilog and VHDL code, enabling code verification for high-integrity applications adhering to DO-254 and other standards.

Support for industry standards is available through IEC Certification Kit (for ISO 26262 and IEC 61508).

Key Features

- Target-independent, synthesizable VHDL and Verilog code
- Code generation support for MATLAB functions, System objects and Simulink blocks
- Mealy and Moore finite-state machines and control logic implementations using Stateflow
- Workflow advisor for programming Xilinx, Microsemi, and Intel application boards
- Resource sharing and retiming for area-speed tradeoffs
- Code-to-model and model-to-code traceability for DO-254
- Legacy code integration

HDL Language Support and Supported Third-Party Tools and Hardware

In this section...

"VHDL and Verilog Language Support" on page 1-3
"Third-Party Synthesis Tools and Version Support" on page 1-3
"FPGA-in-the-Loop Hardware" on page 1-3
"Generic ASIC/FPGA Hardware" on page 1-4
"IP Core Generation Hardware" on page 1-5
" Simulink Real-Time FPGA I/O: Speedgoat Target Computer" on page 1-5
"FPGA Turnkey Hardware" on page 1-6

VHDL and Verilog Language Support

The generated HDL code complies with the following standards:

- VHDL-1993 (IEEE® 1076-1993)
- Verilog-2001 (IEEE 1364-2001)

Third-Party Synthesis Tools and Version Support

The HDL Workflow Advisor is tested with the following third-party FPGA synthesis tools:

- Intel Quartus® Prime Standard 20.1.1
- Intel Quartus Pro 20.2
- Xilinx Vivado® Design Suite 2020.2
- Microsemi Libero® SoC 12.6
- Xilinx ISE 14.7

When you use a synthesis tool that has been tested with the HDL Workflow Advisor and start the workflow, the Advisor generates a list of devices that are supported with that tool. If you use a third-party synthesis tool that is not tested with HDL Workflow Advisor, the Advisor does not update the device list to reflect the FPGA devices that you can use for that tool.

For example, the HDL Workflow Advisor has been tested with Intel Quartus Prime Standard and Intel Quartus Pro. If you use a tool has not been tested with the Advisor, such as Intel Quartus Prime Lite, the FPGA device list does not get updated in the Workflow Advisor.

To use third-party synthesis tools with HDL Coder, a supported synthesis tool must be installed, and the synthesis tool executable must be on the system path. For details, see "Tool Setup" on page 2-2.

FPGA-in-the-Loop Hardware

The FPGAs supported for FPGA-in-the-loop simulation with HDL Verifier™ are listed in the HDL Verifier documentation.

You can also add custom FPGA boards by using the FPGA Board Manager. See “FPGA Board Customization” for details.

For FPGA-in-the-Loop or Customization for USRP™ Device using the HDL Workflow Advisor, a supported synthesis tool must be installed, and the synthesis tool executable must be on the system path. For details, see “Tool Setup” on page 2-2.

Generic ASIC/FPGA Hardware

The following hardware is supported for the Generic ASIC/FPGA workflow:

Synthesis Tool	Device Family
Xilinx Vivado	Kintex7
	Artix7
	Kintex UltraScale+
	KintexU
	Spartan7
	Virtex UltraScale+
	Virtex UltraScale+ HBM
	Virtex UltraScale+ 58G
	Virtex7
	VirtexU
	Zynq
	Zynq UltraScale+
	Zynq Ultrascale+ RFSoC
Xilinx ISE	Virtex6
	Virtex5
	Virtex4
	Spartan-3A DSP
	Spartan 3E
	Spartan3
	Spartan6
Altera® Quartus II **Note** Altera Quartus II refers to the synthesis tool Intel Quartus Prime Standard.	Cyclone® IV
	Cyclone V
	Arria® II GX and GZ
	Stratix® IV
	Stratix V
	Arria 10
	Arria V GX
	MAX 10

Synthesis Tool	Device Family
	Cyclone 10 LP
Intel Quartus Pro	Arria 10
	Cyclone 10 GX
	Stratix 10
Microsemi Libero SoC	SmartFusion2
	RTG4
	IGLOO2
	PolarFire
	PolarFire SoC

IP Core Generation Hardware

The following hardware is supported for the IP Core Generation workflow:

Synthesis Tool	Target Platform
Xilinx Vivado	ZedBoard and with FMC-HDMI-CAM and FMCOMMS2/3/4/
	ZC706 and with FMC-HDMI-CAM and FMCOMMS2/3/4/ and FMCOMMS5
	ZC702 with FMC-HDMI-CAM
	Zynq ZC706 evaluation kit
	Zynq ZC702 evaluation kit
	PicoZed FMC-HDMI-CAM
	Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit
	Zynq UltraScale+ RFSoC ZCU111 Evaluation Kit
	Zynq UltraScale+ RFSoC ZCU216 Evaluation Kit
	Kintex-7 KC705 development board
Intel Quartus Pro	Intel Arria 10 SoC development kit
Altera Quartus II **Note** Altera Quartus II refers to the synthesis tool Intel Quartus Prime Standard.	Intel Arria 10 SoC development kit
	Cyclone V SoC development kit Rev. C and Rev. D
	Arrow DECA Max 10 FPGA development board
	Arrow SoC Kit development board
	Arria 10 GX FPGA development kit
Microsemi Libero SoC	Microchip Polarfire® SoC Icicle Kit

Simulink Real-Time FPGA I/O: Speedgoat Target Computer

You use the `Simulink Real-Time FPGA I/O` workflow to target Speedgoat FPGA I/O modules. These I/O modules are part of Speedgoat® target computer systems. To run the `Simulink Real-Time FPGA I/O` workflow, install the Speedgoat I/O Blockset and the Speedgoat HDL Coder

Integration Packages. After you install the integration packages, you can choose the **Target platform** and then run the workflow to generate a Simulink Real-Time™ interface subsystem.

To learn about:

- The integration packages and how you can install them, see Speedgoat - HDL Coder Integration Packages.
- Speedgoat I/O modules that are supported with the HDL Workflow Advisor, seeSpeedgoat Real-Time FPGA Application Support from HDL Coder .

See “Speedgoat FPGA Support with HDL Workflow Advisor”.

FPGA Turnkey Hardware

The following hardware is supported for the FPGA Turnkey workflow:

- Altera Arria II GX FPGA development kit
- Altera Cyclone III FPGA development kit
- Altera Cyclone IV GX FPGA development kit
- Altera DE2–115 development and education board
- XUP Atlys Spartan-6 development board
- Xilinx Spartan-3A DSP 1800A development board
- Xilinx Spartan-6 SP605 development board
- Xilinx Virtex-4 ML401 development board
- Xilinx Virtex-4 ML402 development board
- Xilinx Virtex-5 ML506 development board
- Xilinx Virtex-6 ML605 development board

For FPGA development boards that have more than one FPGA device, only one such device can be used with FPGA Turnkey. This workflow does not support Xilinx Vivado.

Supported FPGA Device Families for Board Customization

You can also add custom FPGA boards using the FPGA Board Manager. HDL Coder supports the following FPGA device families for board customization; that is, when you create your own board definition file. See “FPGA Board Customization” (HDL Verifier).

Device Family	
Xilinx	Kintex7
	Artix7
	Spartan-3A DSP
	Spartan3
	Spartan3A and Spartan3AN
	Spartan3E
	Spartan6
	Virtex4

Device Family	
	Virtex5
	Virtex6
	Virtex7
Altera	Cyclone III
	Cyclone IV
	Arria II
	Stratix IV
	Stratix V

See Also

`hdlsetuptoolpath`

More About

- “Tool Setup” on page 2-2

Getting Started with HDL Coder

Tool Setup

In this section...

"Synthesis Tool Path Setup" on page 2-2
"HDL Simulator Setup" on page 2-3
"Xilinx System Generator Setup for ModelSim Simulation" on page 2-3
"Altera DSP Builder Setup" on page 2-4
"FPGA Simulation Library Setup" on page 2-4
"C/C++ Compiler Setup" on page 2-5

Synthesis Tool Path Setup

- "hdlsetuptoolpath Function" on page 2-2
- "Add Synthesis Tool for Current HDL Workflow Advisor Session" on page 2-2
- "Check Your Synthesis Tool Setup" on page 2-2
- "Supported Tool Versions" on page 2-3

hdlsetuptoolpath Function

To use HDL Coder with one of the supported third-party FPGA synthesis tools, add the tool to your system path using the `hdlsetuptoolpath` function. Add the tool to your system path before opening the HDL Workflow Advisor. If you already have the HDL Workflow Advisor open, see "Add Synthesis Tool for Current HDL Workflow Advisor Session" on page 2-2.

Add Synthesis Tool for Current HDL Workflow Advisor Session

Simulink to HDL Workflow

1. At the MATLAB command line, use the `hdlsetuptoolpath` function to add the synthesis tool.
2. In the HDL Workflow Advisor, in the **Set Target > Set Target Device and Synthesis Tool** step, to the right of **Synthesis tool**, click **Refresh**.

   The synthesis tool is now available.

MATLAB to HDL Workflow

1. At the MATLAB command line, use the `hdlsetuptoolpath` function to add the synthesis tool.
2. In the HDL Workflow Advisor, in the **Select Code Generation Target** step, to the right of **Synthesis tool**, click **Refresh list**.

   The synthesis tool is now available.

Check Your Synthesis Tool Setup

To check your Intel Quartus Prime Standard synthesis tool setup in MATLAB, try launching the tool with the following command:

```
!quartus
```

To check your Intel Quartus Pro synthesis tool setup in MATLAB, try launching the tool with the following command:

```
!qpro
```

To check your Xilinx Vivado synthesis tool setup in MATLAB, try launching the tool with the following command:

```
!vivado
```

To check your Xilinx ISE synthesis tool setup in MATLAB, try launching the tool with the following command:

```
!ise
```

To check your Microsemi Libero SoC synthesis tool setup in MATLAB, try launching the tool with the following command:

```
!libero
```

Supported Tool Versions

For supported tool versions, see “Third-Party Synthesis Tools and Version Support” on page 1-3.

HDL Simulator Setup

To open the HDL simulator from MATLAB, enter these commands:

MATLAB Command to Open HDL Simulator

HDL Simulator	Command to Open the Simulator
Cadence Incisive®	`nclaunch` (HDL Verifier)
Mentor Graphics® ModelSim®	`vsim` (HDL Verifier)

For example, to open the Mentor Graphics ModelSim simulator, enter this command:

```
vsim('vsimdir','C:\Program Files\ModelSim\questasim\10.5c\win64\vsim.exe')
```

To learn more about how to set up ModelSim, Questa®, or Incisive® for HDL simulation, or for cosimulation with HDL Verifier, see “HDL Simulator Startup” (HDL Verifier).

Add Simulation Tool for Current HDL Workflow Advisor Session

MATLAB to HDL Workflow

1. Set up your simulation tool.
2. In the HDL Workflow Advisor, in the **HDL Verification** > **Verify with HDL Test Bench** task, click **Refresh list**.

   The simulation tool is now available.

Xilinx System Generator Setup for ModelSim Simulation

To generate ModelSim simulation scripts for a design containing Xilinx System Generator blocks, you must:

- Have compiled Xilinx simulation libraries.
- Specify the path to your compiled libraries.

Required Libraries for Vivado and ISE

To generate ModelSim simulation scripts, you must have the following compiled Xilinx simulation libraries for your EDA simulator and target language:

- `unisim`
- `simprim`
- `xilinxcorelib`

To learn how to compile these libraries, refer to the Xilinx documentation.

- For Vivado, see `compile_simlib`.
- For ISE, see `compxlib`.

Specify Path to Required Libraries

Specify the path to your compiled Xilinx simulation libraries by setting the `XilinxSimulatorLibPath` parameter for your model.

For example, you can use `hdlset_param` to set `XilinxSimulatorLibPath`:

```
libpath = '/apps/Xilinx_ISE/XilinxISE-13.4/Linux/ISE_DS/ISE/vhdl/
   mti_se/6.6a/lin64/xilinxcorelib';
hdlset_param (bdroot, 'XilinxSimulatorLibPath', libpath);
```

Altera DSP Builder Setup

To generate code for a design containing both Altera DSP Builder and Simulink blocks, you must open MATLAB with Altera DSP Builder. For details, refer to the Altera DSP Builder documentation.

FPGA Simulation Library Setup

To map your design to an Altera or a Xilinx FPGA simulator library:

- Use Xilinx LogiCORE® IP Floating-Point Operator v5.0 or Altera floating-point megafunction IP cores.
- Specify the compiled simulation library and the target language for your EDA simulator. Use `XilinxCoreLib` simulation library for Xilinx LogiCORE IP and the EDA simulation library compiler for Altera megafunction IP.

  To learn how to compile this library, refer to the Xilinx `compxlib` documentation .
- Specify the path to your compiled Altera or Xilinx simulation libraries. Altera provides the simulation model files in **\quartus\eda\sim_lib** folder. Set the `SimulationLibPath` parameter for your DUT.

  For example, you can use `hdlset_param` to set `SimulationLibPath`:

  ```
  myDUT = gcb;
  libpath = '/apps/Xilinx_ISE/XilinxISE-13.4/Linux/ISE_DS/ISE/vhdl/
  ```

```
   mti_se/6.6a/lin64/xilinxcorelib';
hdlset_param (myDUT, 'SimulationLibPath', libpath);
```

You can also specify the simulation library path from the **HDL Code Generation > Test Bench** pane in the Configuration Parameters dialog box.

C/C++ Compiler Setup

HDL Coder locates and uses a supported installed compiler. For most platforms, a default compiler is supplied with MATLAB. For a list of supported compilers, see at https://www.mathworks.com/support/compilers/current_release/.

See Also

`hdlsetuptoolpath`

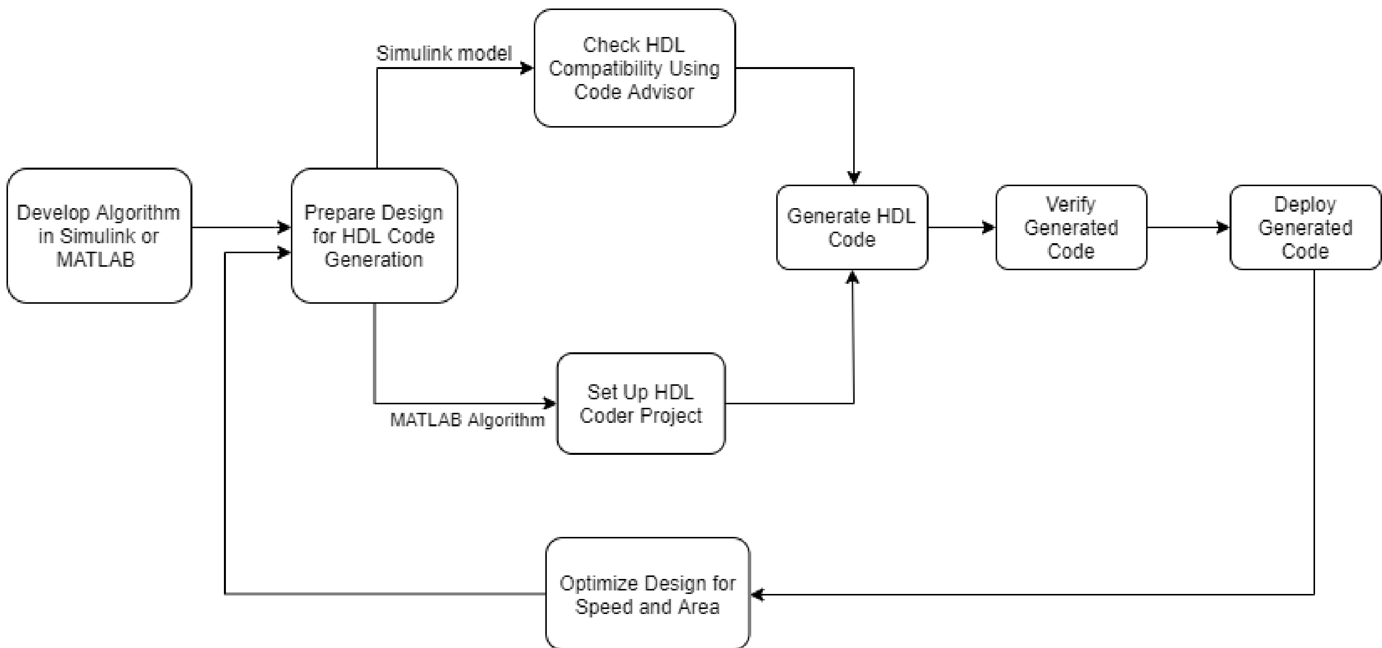
More About

- "Third-Party Synthesis Tools and Version Support" on page 1-3

Tutorials

- "Basic HDL Code Generation Workflow" on page 3-2
- "Create HDL-Compatible Simulink Model" on page 3-3
- "Generate HDL Code from Simulink Model" on page 3-9
- "Verify Generated HDL Code from Simulink Model" on page 3-16
- "HDL Code Generation and FPGA Synthesis from Simulink Model" on page 3-21
- "Generation of Clock Bundle Signals in HDL Coder" on page 3-26
- "Get Started with MATLAB to HDL Workflow" on page 3-29
- "Basic HDL Code Generation and FPGA Synthesis from MATLAB" on page 3-35
- "Generate HDL Code from MATLAB Code Using the Command Line Interface" on page 3-42
- "Generating Modular HDL Code for Functions" on page 3-46
- "System Design with HDL Code Generation from MATLAB and Simulink" on page 3-53

Basic HDL Code Generation Workflow



See Also

More About

- “Create HDL-Compatible Simulink Model” on page 3-3
- “Use Simulink Templates for HDL Code Generation”
- “Create and Set Up Your Project”
- “Check HDL Compatibility of Simulink Model Using HDL Code Advisor”
- “Generate HDL Code from Simulink Model” on page 3-9
- “Verify Generated HDL Code from Simulink Model” on page 3-16
- “HDL Code Generation and FPGA Synthesis from Simulink Model” on page 3-21
- “Basic HDL Code Generation and FPGA Synthesis from MATLAB” on page 3-35

Create HDL-Compatible Simulink Model

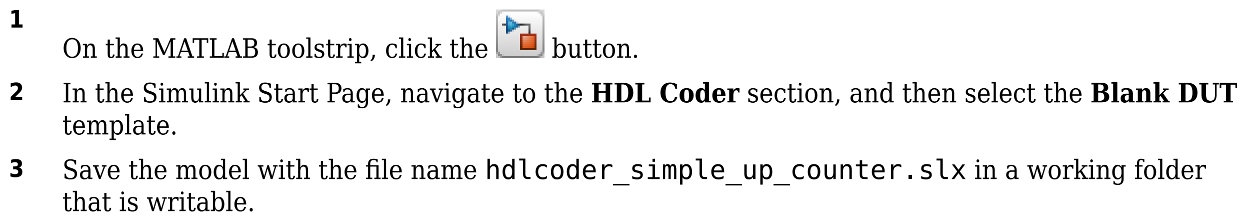
In this section...
“Use Blank DUT Template” on page 3-3
“Choose Blocks from HDL Coder Library” on page 3-4
“Develop Algorithm for DUT” on page 3-5
“Create Test Bench for Design” on page 3-6
“Simple Counter Model” on page 3-6
“Simulate and Verify Design Functionality” on page 3-7
“Generate HDL Code from Simulink Model” on page 3-8

This example illustrates how you can create a Simulink model for HDL code generation. To create a MATLAB algorithm compatible for HDL code generation, see “Guidelines for Writing MATLAB Code to Generate Efficient HDL and SystemC Code”.

The model is a simple counter algorithm that counts upward and wraps back to zero after it reaches the upper limit that you specify. To open the model directly without performing the steps, see “Simple Counter Model” on page 3-6.

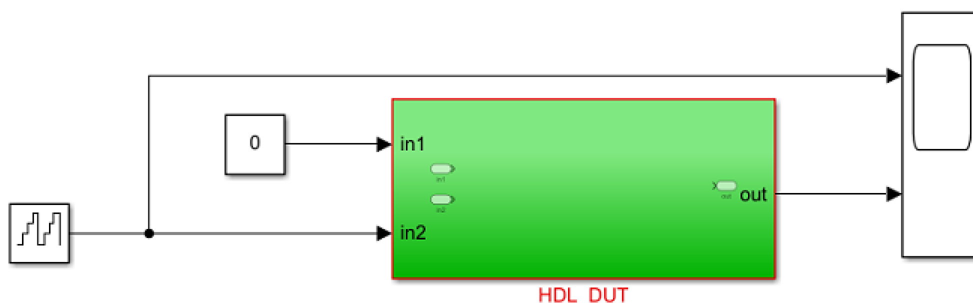
Use Blank DUT Template

To create a HDL-compatible Simulink model, use the **Blank DUT** template. The template is preconfigured for HDL code generation by using the `hdlsetup` function.



1. On the MATLAB toolstrip, click the button.
2. In the Simulink Start Page, navigate to the **HDL Coder** section, and then select the **Blank DUT** template.
3. Save the model with the file name `hdlcoder_simple_up_counter.slx` in a working folder that is writable.

Note: This model is configured with '**hdlsetup**'



Add your design targeted for ASIC/FPGA inside HDL_DUT and then run the following command:
makehdl('HDL_DUT')

The **Blank DUT** template has a `HDL_DUT` subsystem that corresponds to the Design-Under-Test (DUT) for which you generate HDL code. To verify the DUT functionality, the template contains a test bench outside the `HDL_DUT` subsystem that provides inputs to the DUT and logs output values. See "Partition Model into DUT and Test Bench".

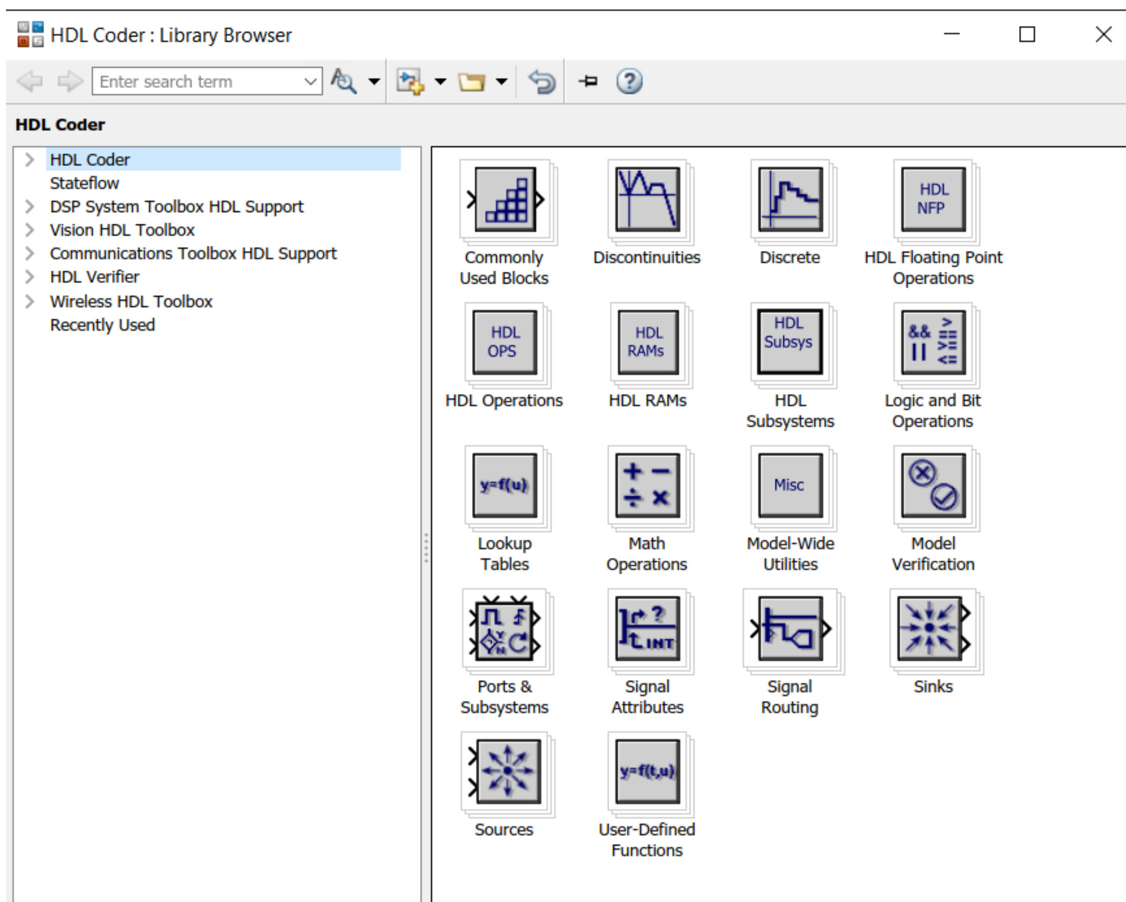
Choose Blocks from HDL Coder Library

To design your counter algorithm, use blocks from the **HDL Coder** Block Library. Blocks in this library are preconfigured for HDL code generation. To filter the Simulink Library Browser to show block libraries that support HDL code generation:

1. On the **Apps** tab, select **HDL Coder**.
2. From the **HDL Code** tab, select **HDL Block Properties** > **Open HDL Block Library**.

Alternatively, at the command line, enter `hdllib`.

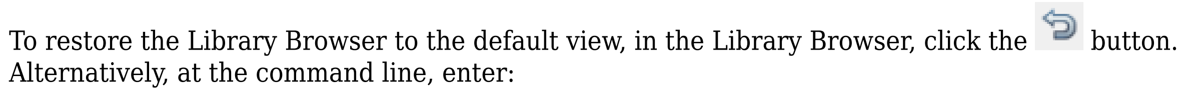
```
hdllib
```



Blocks in the **HDL Coder** Library are available with Simulink. If you do not have HDL Coder, you can simulate the blocks in your model, but cannot generate HDL code.

You can find additional HDL-supported blocks in these block libraries:

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



To restore the Library Browser to the default view, in the Library Browser, click the button. Alternatively, at the command line, enter:

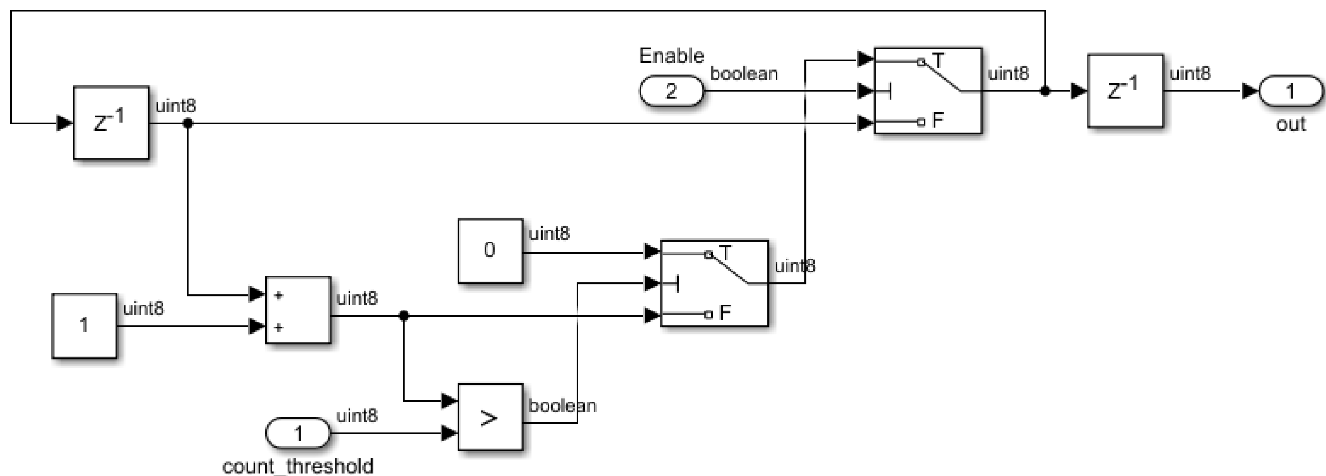
```
hdllib('off')
```

Develop Algorithm for DUT

1 Double-click the `HDL_DUT` subsystem. Drag blocks from the **HDL Coder** library to your model. This table lists the blocks to add to your model for designing the counter. To learn about what a block does and to specify its block parameters, double-click the block.

Block	Library	Number of Blocks	Block Parameters
Constant	Sources	2	• Constant values: 1 and 0 • Output data type: uint8
Switch	Signal Routing	2	Criteria for passing first input: u2 > Threshold
Delay	Discrete	2	Delay length: 1
Add	Math Operations	1	Accumulator data type: Inherit: Same as first input
Relational Operator	Logic and Bit Operations	1	Relational operator: >

2 Rename the input ports `In1` and `In2` to `count_threshold` and `Enable` respectively. Place the blocks in your model and connect them.



The `Enable` signal specifies whether the counter counts upward from the previous value. When the `Enable` signal is logical high, the counter counts up from zero to the `count_threshold` value. When

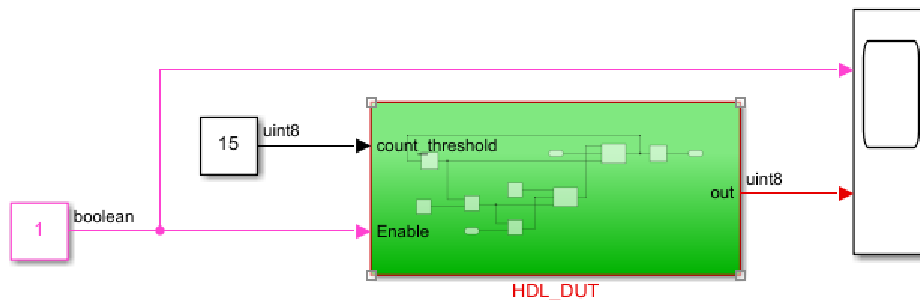
the value of out becomes equal to the count_threshold value, the counter wraps back to zero and starts counting again. When the Enable signal becomes logical low, the counter holds the previous value.

Create Test Bench for Design

Navigate to the top level of the model and change the input settings.

- Constant block input to count_threshold: This input indicates the maximum value up to which the counter counts. This example shows how to design a 4-bit up counter. Set the **Constant value** to 15 (2^4 - 1), and set the **Output data type** to uint8.

  The output data type of this Constant block then matches the output data type of the Constant blocks inside the HDL_DUT subsystem.
- Counter Free-Running block input to Enable: Remove the Counter Free-Running block. Replace this block with a Constant block that has a value of 1, **Output data type** set to boolean, and **Sample time** of 1.



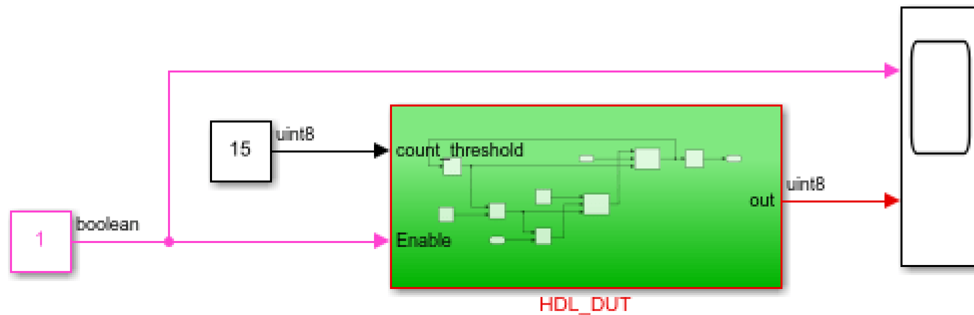
See also “Create a Simple Model”.

The preceding section shows the hdlcoder_simple_up_counter.slx model that you created by following the steps described above. To open the model in MATLAB, click the **Open Model** button.

Simple Counter Model

Open this model to see a simple counter. The model counts up from zero to a threshold value and then wraps back to zero. The threshold value is set to 15. To change the threshold value, change the value of the input to the count_threshold port. The Enable signal specifies whether the counter counts upward or holds the previous value. A value of 1 indicates that the counter counts upward continuously.

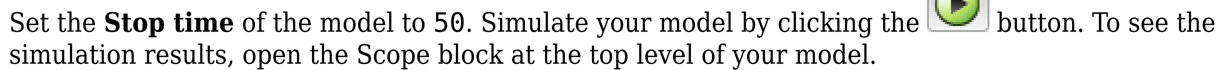
Note: This model is configured with 'hdlsetup'



Add your design targeted for ASIC/FPGA inside HDL_DUT and then run the following command:
makehdl('HDL_DUT')

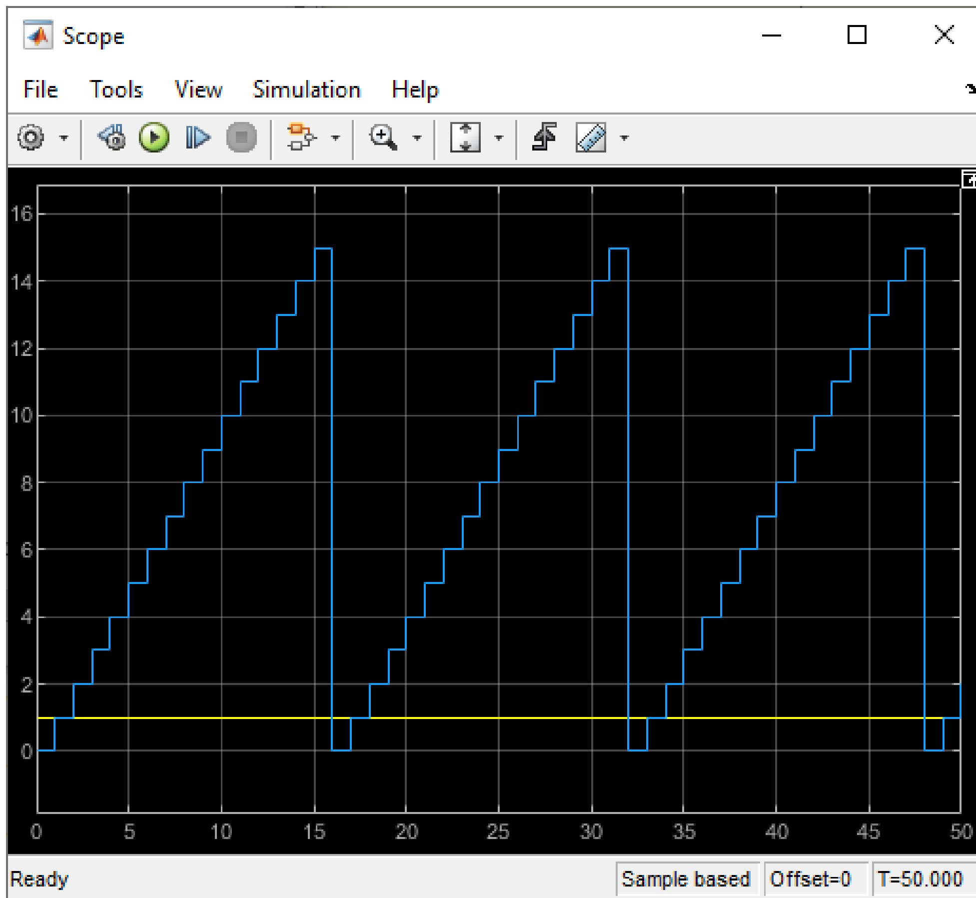
Copyright 2018-2021 The MathWorks, Inc.

Simulate and Verify Design Functionality



Set the **Stop time** of the model to `50`. Simulate your model by clicking the button. To see the simulation results, open the Scope block at the top level of your model.

The simulation results display the `Enable` signal generating a constant value of `1`. The `out` signal counts from `0` to `15`, wraps back to zero, and then counts up again.



Generate HDL Code from Simulink Model

Before you generate HDL code, you can verify that the model settings are compatible for HDL code generation. The counter model used in this example is compatible for HDL code generation. To verify and update your model for HDL compatibility, use the HDL Code Advisor. See "Check HDL Compatibility of Simulink Model Using HDL Code Advisor".

See "Generate HDL Code from Simulink Model" on page 3-9.

See Also

`hdllib` | `checkhdl` | `hdlsetup` | `hdlcodeadvisor`

More About

- "Use Simulink Templates for HDL Code Generation"
- "Verify Generated HDL Code from Simulink Model" on page 3-16
- "HDL Code Generation and FPGA Synthesis from Simulink Model" on page 3-21

Generate HDL Code from Simulink Model

In this section...

"Models for HDL Code Generation" on page 3-9
"Simple Counter Model" on page 3-9
"Generate HDL Code" on page 3-10
"View HDL Code Generation Files" on page 3-11
"Inspect Generated HDL Code" on page 3-12
"Validate HDL Behavior Using Validation Model" on page 3-14
"Verify Generated HDL Code" on page 3-15

This example shows how you can generate HDL code for a simple counter model in Simulink. This model is compatible for HDL code generation. To create this counter model, see "Create HDL-Compatible Simulink Model" on page 3-3.

Models for HDL Code Generation

You can either create your own HDL-compatible model such as the counter model or choose from:

- HDL Coder example models available in the `hdlcoderdemos` folder.

  ```
  cd (fullfile(matlabroot,'toolbox','hdlcoder','hdlcoderdemos'))
  ```

  These models are available on the MATLAB path. For example, you can choose the symmetric "FIR Filter Model". To use this model, enter:

  ```
  sfir_fixed
  ```

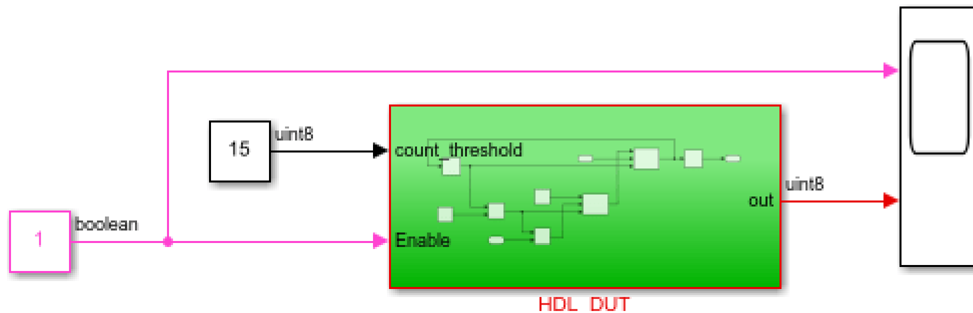
- Simulink templates for HDL code generation. You can use templates to model registers, ROM, basic arithmetic operations, complex multipliers, shift registers, and so on.

  To choose your template, on the MATLAB toolstrip, click the button, and then navigate to the **HDL Coder** section. See "Use Simulink Templates for HDL Code Generation".

Before generating HDL code, you can check and update the model for HDL compatibility by using the HDL Code Advisor. See "Check HDL Compatibility of Simulink Model Using HDL Code Advisor".

Simple Counter Model

Open this model to see a simple counter. The model counts up from zero to a threshold value and then wraps back to zero. The threshold value is set to `15`. To change the threshold value, change the value of the input to the `count_threshold` port. The Enable signal specifies whether the counter counts upward or holds the previous value. A value of `1` indicates that the counter counts upward continuously.



Generate HDL Code

For the counter model, the `HDL_DUT` subsystem is the DUT. To generate code for the DUT:

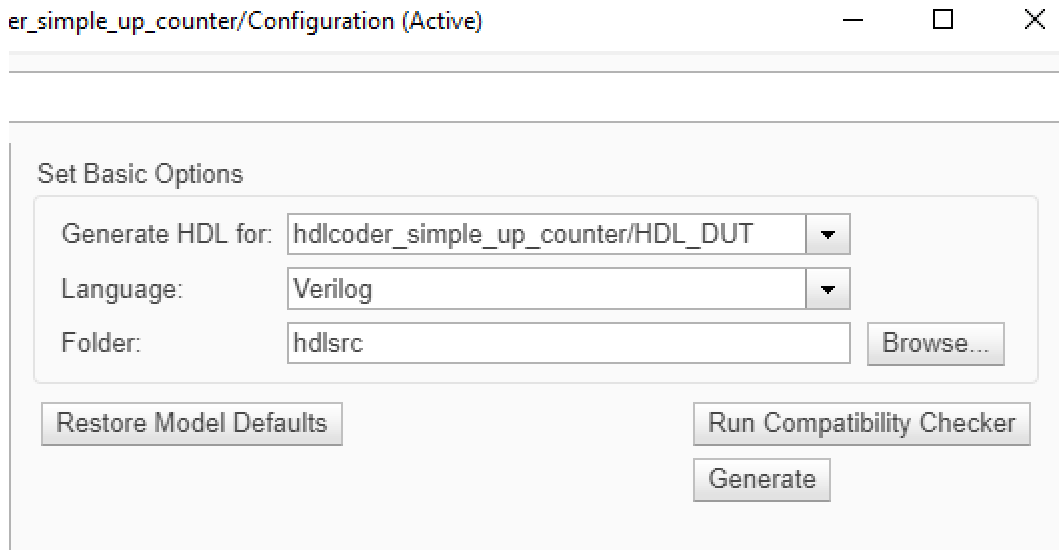
1. In the **Apps** tab, select **HDL Coder**.
2. Select the DUT Subsystem in your model, and make sure that this Subsystem name appears in the **Code for** option on the **HDL Code** tab. To remember the selection, pin this option. Click **Generate HDL Code**.

By default, HDL Coder generates VHDL code in the target `hdlsrc` folder.

Generate Verilog Code

To generate Verilog code for the counter model:

1. In the **HDL Code** tab, click **Settings**.
2. In the **HDL Code Generation** pane, for **Language**, select `Verilog`. Click **Apply** and then click **Generate**.



HDL Coder compiles the model before generating code. Depending on model display options such as port data types, the model can change in appearance after code generation. As code generation proceeds, HDL Coder displays progress messages in the MATLAB command line with links to the configuration set and the generated files. To view the files in the MATLAB Editor, click the links.

The process is completed and displays the message:

```
### HDL Code Generation Complete.
```

View HDL Code Generation Files

A folder icon for the `hdlsrc` folder appears in the current folder. To view the generated code and script files, double-click the `hdlsrc` folder, and then double-click the folder that has the same name as the model for which you generated HDL code.

- `HDL_DUT.vhd`: VHDL code that contains the entity definition and RTL architecture implementing the counter that you designed. If you generated Verilog code, you get a `HDL_DUT.v` file.
- `HDL_DUT_compile.do`: Mentor Graphics ModelSim compilation script.
- `HDL_DUT_map.txt`: Mapping file that maps generated entities or modules in the HDL code to subsystems in the model that generated them. See “Trace Code Using the Mapping File”.
- `HDL_DUT_report.html`: HDL check report displays HDL code generation status and warnings or messages.
- `gm_hdlcoder_simple_up_counter.slx`: Generated model that behaviorally represents the HDL code in the Simulink modeling environment.

HDL Coder creates a behavioral model of the HDL code called the generated model. The generated model name is the same as the original model and has the prefix `gm_`. The generated model is bit-true and cycle-accurate to the generated HDL code. This model shows the effect of block implementations and speed and area optimizations that you specified. See also “Speed and Area Optimizations in HDL Coder”.

To open the generated model for the counter, enter:

```
gm_hdlcoder_simple_up_counter
```

For the counter model, as optimizations are disabled, the generated model is identical to the original model.

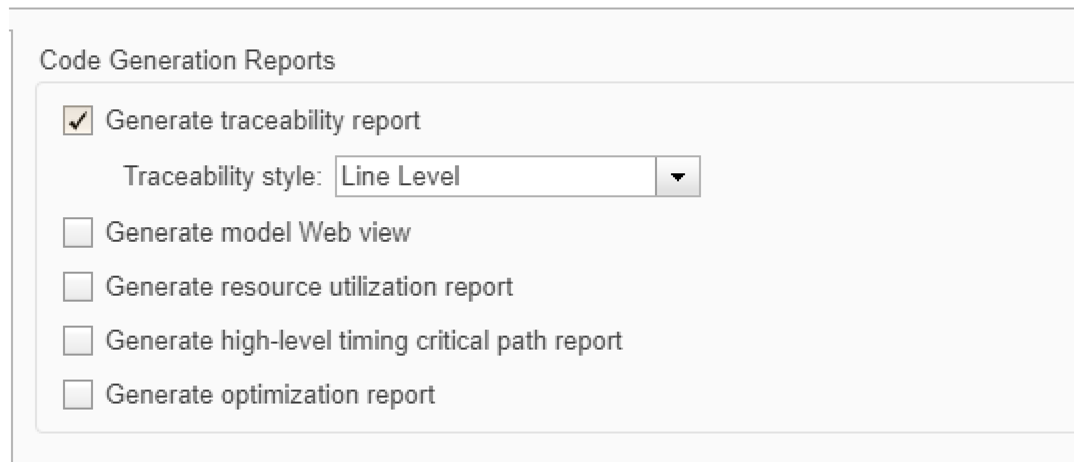
To view your generated HDL code alongside your model, you can use the Code view. After you generate HDL code for your model, the Code view displays the generated code to the right of your model. To manually open the Code view, open the **HDL Coder** app. On the Simulink toolstrip click the **View Code** button. Select the file that you want to display by using the drop-down list at the top of the Code view. You can dock or undock the Code view from the editor and minimize or expand the Code view using the down arrow in the upper right corner of the Code view.

Inspect Generated HDL Code

To identify the mapping between the source model and the generated HDL code more easily, generate a traceability report. Use the report to navigate from a block in your model to the generated code for that block and from the code to a block in your model.

To generate the traceability report:

1. In the **HDL Code** tab, click **Settings** > **Report Options**.
2. In the **HDL Code Generation** > **Report** pane, select **Generate traceability report**, and then generate HDL code for the `HDL_DUT` subsystem

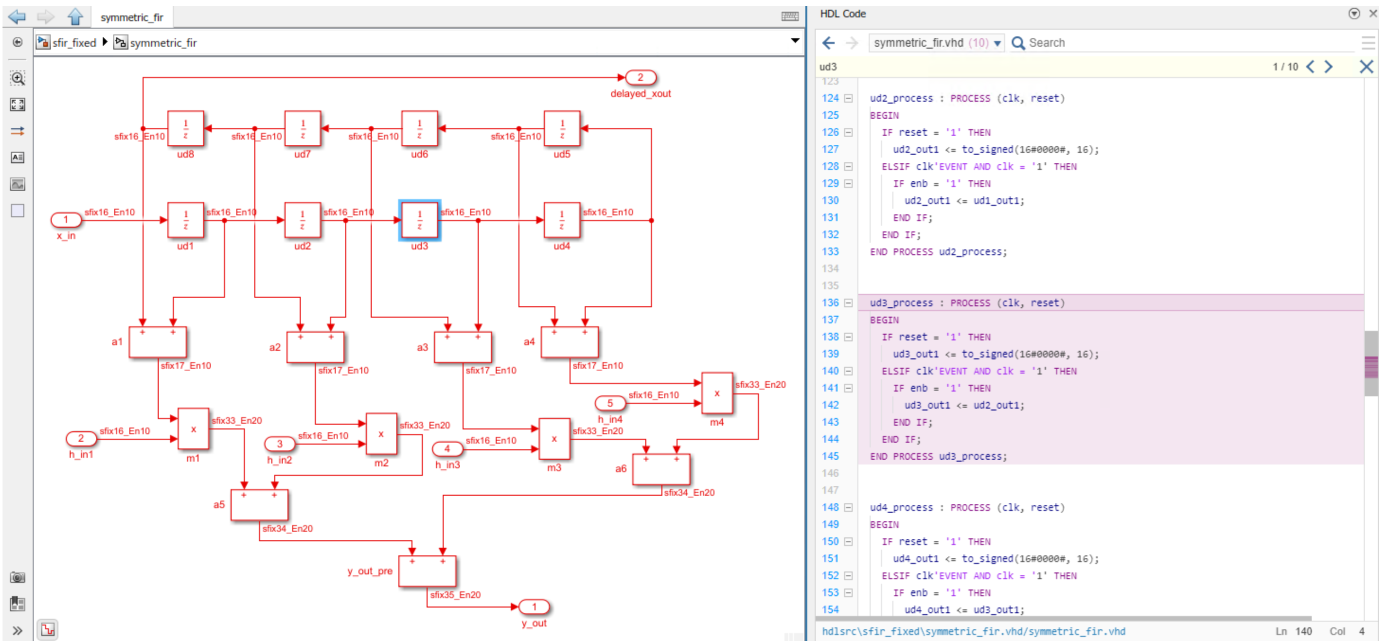


After you generate code, the Code Generation Report window opens. HDL Coder writes the code generation report files in the `hdlsrc\html\` folder of the build folder. If you close the report, you can navigate to this folder to reopen the report.

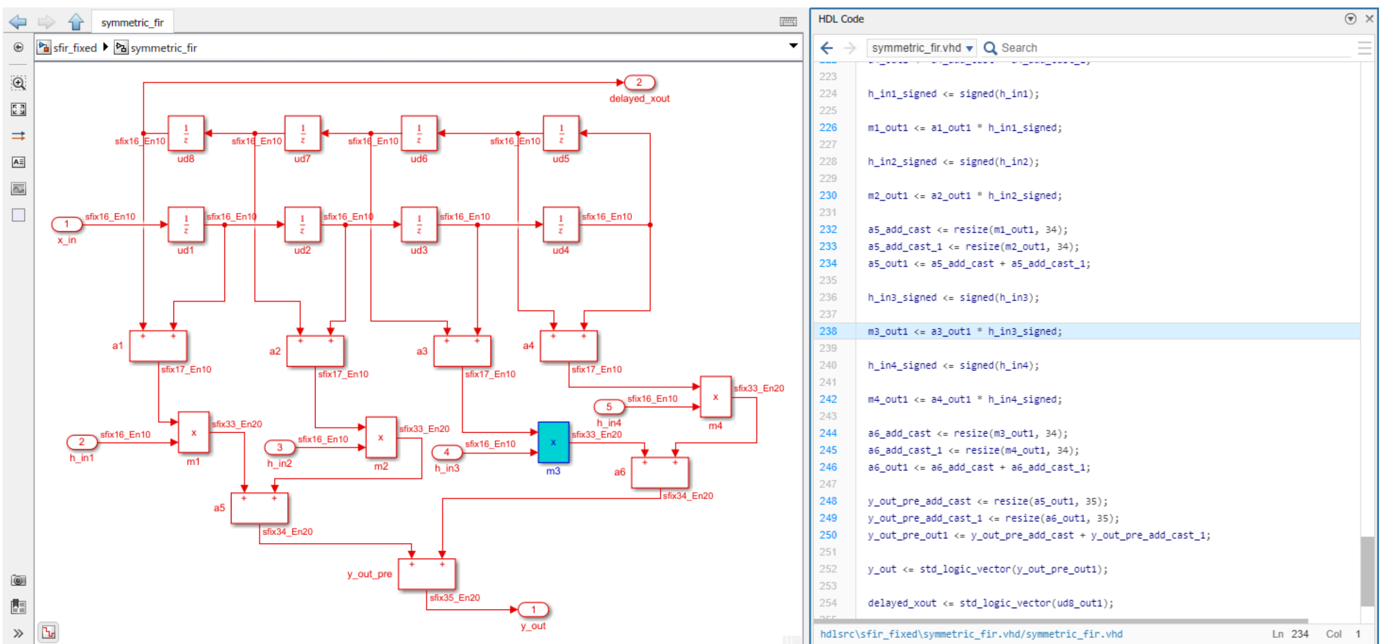
To navigate from the HDL code to the model, follow either of these workflows:

Use the Code view:

1. Click the Code view panel on the right that appears after generating HDL code or manually click the **View Code** button on the Simulink toolstrip of the **HDL Coder** app.
2. To navigate from model elements to their generated code, in your model, click a block. The Code view highlights the code for the block and scrolls to the highlighted code lines.



3 In the Code view, click the line number hyperlink or code comment link to highlight the block that the code line traces to. You can trace lines of code to the model elements from which they were generated.

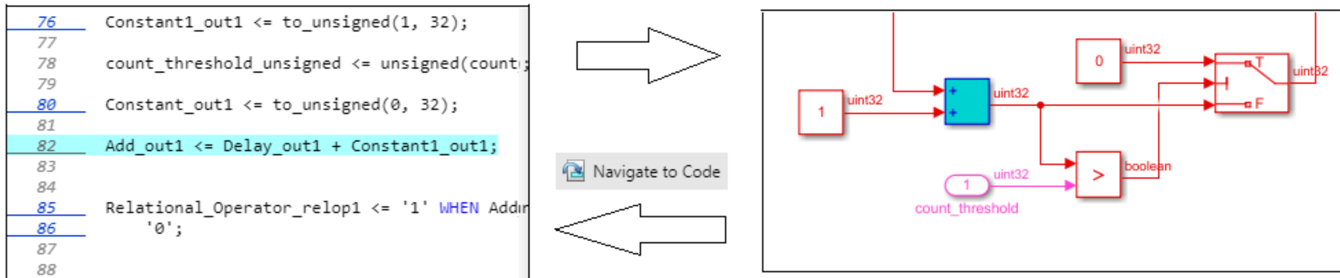


Use the Code Generation Report:

1 In the Code Generation Report, navigate to the **Traceability Report** section, and then click the links in the **Code Location** section.

2 Select the hyperlink to a line of code to highlight the corresponding block in your model.

To navigate from a block in your model to the HDL code, select that block, and then click the **Navigate to Code** button in the **Review Results** section of the **HDL Code** tab.



See “Navigate Between Simulink Model and HDL Code by Using Traceability” and “Create and Use Code Generation Reports”.

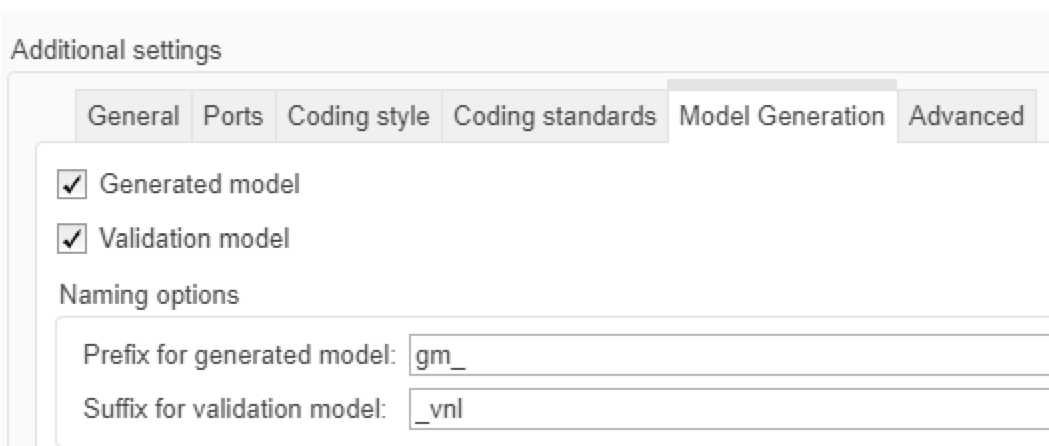
In the **Generated Source Files** section, if you click the HDL file `HDL_DUT`, you see the signals `clk`, `reset`, and `clk_enable`. These signals are the clock, reset, and clock enables signals that control the flip-flops on the target hardware. HDL Coder generates these signals in the code depending on sequential elements such as Delay blocks that you use in your model. See “Generation of Clock Bundle Signals in HDL Coder” on page 3-26.

Validate HDL Behavior Using Validation Model

To validate the behavioral model of the HDL code with your original model, generate a validation model. The validation model contains both the original model and generated model. It compares the outputs of both models by using the test vectors that you provided in the original model.

To generate the validation model:

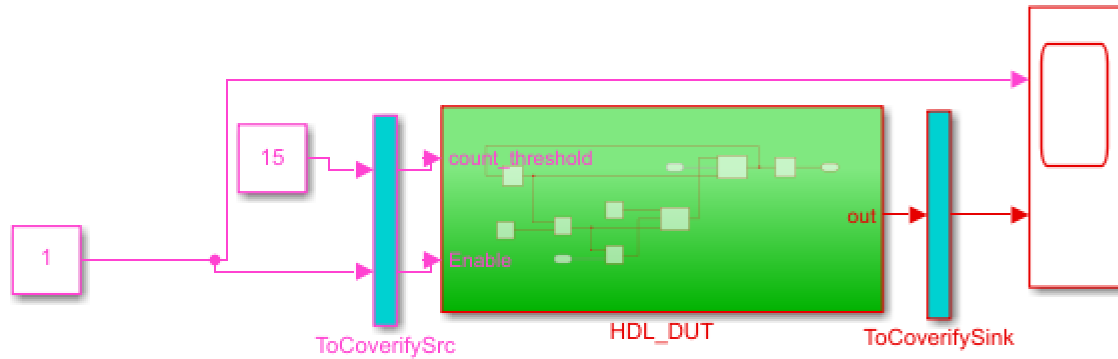
1. In the **HDL Code** tab, click **Settings**.
2. In the **HDL Code Generation** > **Global Settings** > **Model Generation** tab, select **Validation model**, and then generate HDL code for the `HDL_DUT` subsystem.

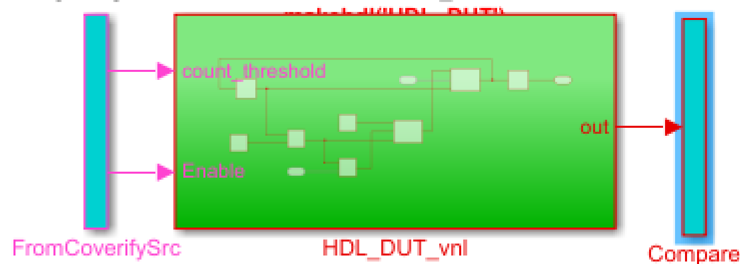


In the code generation logs, you see a link to the validation model. The validation model has the same prefix as the generated model and also has the suffix `_vnl`. For the counter model, the validation

model has the name `gm_hdlcoder_simple_up_counter_vnl.slx`. You can find this model in the same folder as the generated model. To open this model, enter:

```
gm_hdlcoder_simple_up_counter_vnl
```





After you simulate the model, double-click the `Compare` subsystem, and then navigate inside the `Assert_Out` subsystem. If you open the Scope block, you see that the `err` signal has a value of zero, which means that the generated model output matches the original model.

See “Generated Model and Validation Model”.

Verify Generated HDL Code

Before you deploy your design on the target hardware, verify the generated HDL code. From the `hdlsrc` folder, navigate to the current working folder. See “Verify Generated HDL Code from Simulink Model” on page 3-16.

See Also

`makehdl` | `hdlset_param` | `hdlsetup`

More About

- “Create HDL-Compatible Simulink Model” on page 3-3
- “HDL Code Generation and FPGA Synthesis from Simulink Model” on page 3-21

Verify Generated HDL Code from Simulink Model

In this section...

"What is an HDL Test Bench?" on page 3-16
"Simple Counter Model" on page 3-16
"Verification Methods" on page 3-17
"Generate HDL Test Bench" on page 3-17
"View HDL Test Bench Files" on page 3-18
"Run Simulation and Verify Generated HDL Code" on page 3-18
"Deploy Generated HDL Code on Target Device" on page 3-19

This example shows how to generate an HDL test bench and verify the generated code for a simple counter model. To generate HDL code for this model, see "Generate HDL Code from Simulink Model" on page 3-9. If you have not generated HDL code for this model, HDL Coder runs code generation before generating the testbench.

What is an HDL Test Bench?

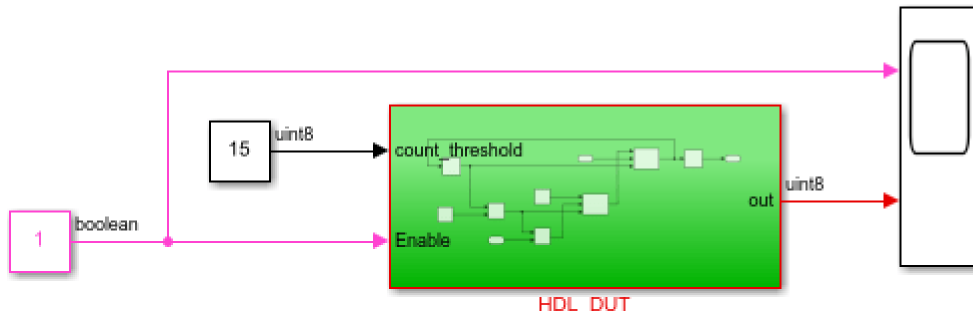
To verify the functionality of the HDL code for the DUT, generate a HDL test bench. A test bench includes:

- Stimulus data generated by signal sources connected to the entity under test.
- Output data generated by the entity under test. During a test bench run, this data is compared to the outputs of the VHDL model for verification.
- Clock, reset, and clock enable inputs to drive the entity under test.
- A component instantiation of the entity under test.
- Code to drive the entity under test and compare its outputs to the expected data.

You can simulate the generated test bench and script files by using the Mentor Graphics ModelSim simulator.

Simple Counter Model

Open this model to see a simple counter. The model counts up from zero to a threshold value and then wraps back to zero. The threshold value is set to `15`. To change the threshold value, change the value of the input to the `count_threshold` port. The Enable signal specifies whether the counter counts upward or holds the previous value. A value of `1` indicates that the counter counts upward continuously.



Verification Methods

If you have HDL Verifier installed, you can also verify the generated HDL code by using these methods.

Verification Method	For More Information
HDL Cosimulation	"Cosimulation"
SystemVerilog DPI Test Bench	"SystemVerilog DPI Test Bench"
FPGA-in-the-Loop	"FPGA-in-the-Loop"

Generate HDL Test Bench

Generate VHDL or Verilog test bench code as applicable. By default, the HDL code and the test bench code are written to the same target folder `hdlsrc` relative to the current folder.

For the counter model, the `HDL_DUT` subsystem is the DUT. To generate the testbench, select this subsystem.

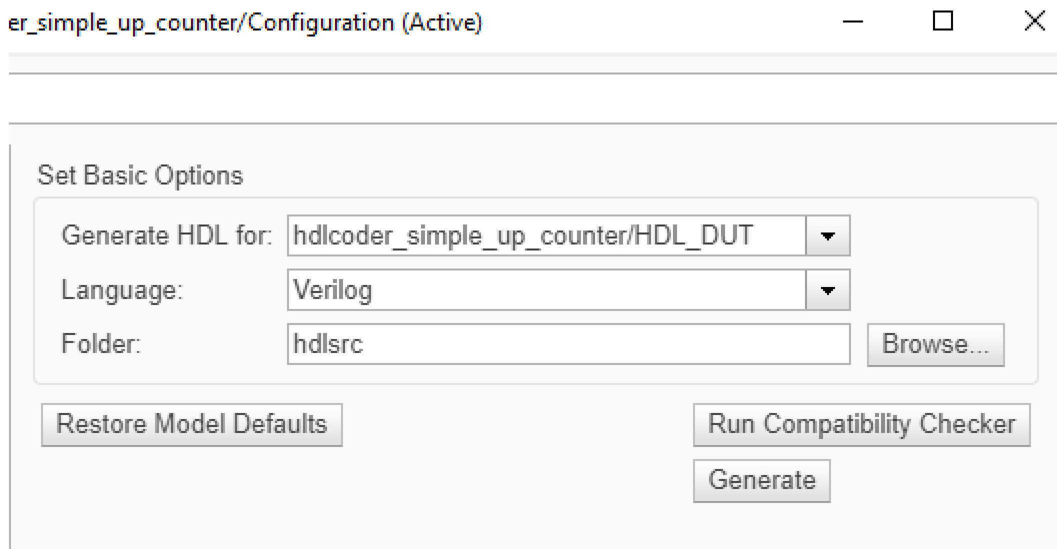
1. In the **Apps** tab, select **HDL Coder**.
2. Select the DUT subsystem, `HDL_DUT`, and make sure this name appears in the **Code for** option on the **HDL Code** tab. To remember the selection, pin this option. Click **Generate Testbench**.

Generate Verilog Test Bench Code

To generate Verilog testbench code for the counter model:

1. In the **HDL Code** tab, click **Settings**.
2. In the **HDL Code Generation** pane, for **Language**, select `Verilog`.

3 In the **HDL Code Generation** > **Test Bench** pane, click **Generate Test Bench**.



HDL Coder compiles the model and generates the test bench.

Test bench generation is completed and displays this message. The generated files appear in the `hdlsrc` folder.

```
### HDL TestBench Generation Complete.
```

View HDL Test Bench Files

For the counter model, the `hdlsrc` folder contains these test bench files.

- `HDL_DUT_tb.vhd`: VHDL test bench code containing generated test and output data. If you generated Verilog test bench code, the generated file is `HDL_DUT_tb.v`.
- `HDL_DUT_tb_pkg.vhd`: Package file for VHDL test bench code. This file is not generated if you specified Verilog as the target language.
- `HDL_DUT_tb_compile.do`: Mentor Graphics ModelSim compilation script (`vcom` commands). This script compiles and loads the entity to be tested (`HDL_DUT.vhd`) and the test bench code (`HDL_DUT_tb.vhd`).
- `HDL_DUT_tb_sim.do`: Mentor Graphics ModelSim script to initialize the simulator, set up **wave** window signal displays, and run a simulation.

To view the generated test bench code in the MATLAB Editor, double-click the `HDL_DUT_tb.vhd` or `HDL_DUT_tb.v` file in the current folder.

Run Simulation and Verify Generated HDL Code

To verify the simulation results, you can use the Mentor Graphics ModelSim simulator. You must have already installed Mentor Graphics ModelSim.

To open the simulator, use the `vsim` (HDL Verifier) function. This command shows how to open the simulator by specifying the path to the executable:

```
vsim('vsimdir','C:\Program Files\ModelSim\questasim\10.6b\win64\vsim.exe')
```

To compile and run a simulation of the generated model and test bench code, use the HDL Coder generated scripts. For the counter model, run these commands to compile and simulate the generated test bench for the `HDL_DUT` Subsystem.

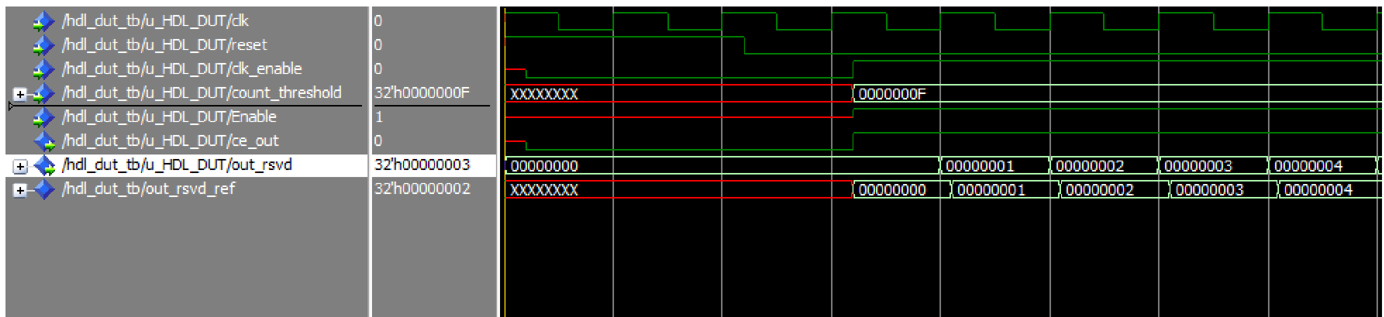
1. Open the Mentor Graphics ModelSim software and navigate to the folder that has the generated code files and the scripts.
2. Use the generated compilation script to compile and load the generated model and text bench code. For the `HDL_DUT` subsystem, run this command to compile the generated code.

   ```
   QuestaSim>do HDL_DUT_tb_compile.do
   ```

3. Use the generated simulation script to execute the simulation. The following listing displays the command. You can ignore warning messages. For the `HDL_DUT` Subsystem, run this command to simulate the generated code.

   ```
   QuestaSim>do HDL_DUT_tb_sim.do
   ```

   The simulator optimizes your design and displays the results in a **wave** window. if you don't see the simulation results, open the **wave** window. The simulation script displays inputs and outputs in the model including the clock, reset, and clock enable signals in the **wave** window.



You can now view the signals and verify that the simulation results match the functionality of your original design. After verifying, close the Mentor Graphics ModelSim simulator, and then close the open files in the MATLAB Editor.

Deploy Generated HDL Code on Target Device

To deploy the generated code on a target FPGA device, use the Simulink HDL Workflow Advisor. See “HDL Code Generation and FPGA Synthesis from Simulink Model” on page 3-21.

See Also

`makehdl` | `makehdltb`

More About

- “Test Bench Generation Output Parameters”
- “HDL Test Bench”

- “HDL Code Generation and FPGA Synthesis from Simulink Model” on page 3-21

HDL Code Generation and FPGA Synthesis from Simulink Model

In this section...

"Simulink HDL Workflow Advisor" on page 3-21
"Simple Counter Model" on page 3-21
"Set Up Tool Path" on page 3-22
"Open the HDL Workflow Advisor" on page 3-22
"Generate HDL Code" on page 3-23
"Perform FPGA Synthesis and Analysis" on page 3-24
"Run Workflow at Command Line with a Script" on page 3-25

This example shows how you can generate HDL code for a simple counter model and synthesize the generated code on a Xilinx FPGA by using the Simulink HDL Workflow Advisor. To create this model, see "Create HDL-Compatible Simulink Model" on page 3-3.

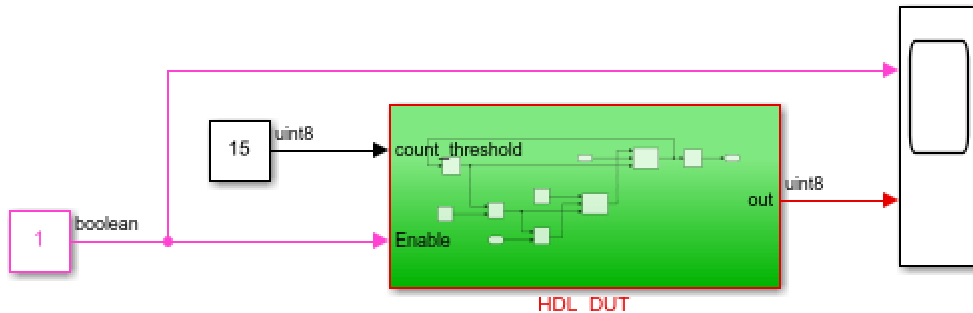
Simulink HDL Workflow Advisor

The HDL Workflow Advisor guides you through generating HDL code for a Simulink subsystem and the FPGA design process, such as:

- Checking the model for HDL code generation compatibility and automatically fixing incompatible settings.
- Generation of HDL code, a test bench, and scripts to build and run the code and test bench.
- Generation of cosimulation or SystemVerilog DPI test benches and code coverage (requires HDL Verifier).
- Synthesis and timing analysis through integration with third-party synthesis tools.
- Back-annotation of the model with critical path information and other information obtained during synthesis.
- Complete automated workflows for selected FPGA development target devices and the Simulink Real-Time FPGA I/O workflow, including FPGA-in-the-loop simulation.

Simple Counter Model

Open this model to see a simple counter. The model counts up from zero to a threshold value and then wraps back to zero. The threshold value is set to `15`. To change the threshold value, change the value of the input to the `count_threshold` port. The Enable signal specifies whether the counter counts upward or holds the previous value. A value of `1` indicates that the counter counts upward continuously.



Set Up Tool Path

To synthesize your design on a target platform, before you open the HDL Workflow Advisor and run the workflow, set up the path to your synthesis tool. This example uses Xilinx Vivado, so you must have already installed Xilinx Vivado. To set the tool path, use the `hdlsetuptoolpath` function to point to an installed Xilinx Vivado 2019.2 executable.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath',...
 'C:\Xilinx\Vivado\2019.2\bin\vivado.bat');
```

If you use a different synthesis tool, set the path to that synthesis tool by using `hdlsetuptoolpath`. To learn about the latest supported tools, see “HDL Language Support and Supported Third-Party Tools and Hardware” on page 1-3.

If you want to generate HDL code but not synthesize your design, you do not have to set the tool path.

Open the HDL Workflow Advisor

To start the HDL Workflow Advisor from a Simulink model,

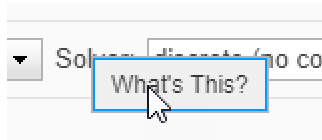
1. In the **Apps** tab, select **HDL Coder**.
2. Select the DUT Subsystem in your model, `HDL_DUT`, and make sure this name appears in the **Code for** option on the **HDL Code** tab. To remember the selection, pin this option. Click **Workflow Advisor**.

When you open the HDL Workflow Advisor, the code generator might warn that the project folder is incompatible. To open the Advisor, select **Remove and continue**.

The HDL Workflow Advisor displays a group of tasks in the left pane grouped by folders. Expanding the folders shows available tasks in each folder. Selecting a task or folder displays information about

that task or folder in the right pane. The right pane has simple controls for running the task to several parameters and options for code or test bench generation, and contains a display area for status messages and other task results.

To learn more about each individual task, right-click that task, and select **What's This?**.



See “Getting Started with the HDL Workflow Advisor”.

Generate HDL Code

1. In the **Set Target** > **Set Target Device and Synthesis Tool** step, for **Synthesis tool**, select `Xilinx Vivado` and select **Run This Task**. See also “Workflows in HDL Workflow Advisor”. To generate HDL code but not synthesize the code, leave the **Synthesis tool** setting to `No Synthesis Tool Specified`.

   1.1. Set Target Device and Synthesis Tool
   Analysis (^Triggers Update Diagram)
   Set Target Device and Synthesis Tool for HDL code generation
   Input Parameters
   Target workflow: Generic ASIC/FPGA
   Target platform: Launch Board Manager
   Synthesis tool: Xilinx Vivado Tool version: 2018.2 Refresh
   Family: Virtex7 Device: xc7vx485t
   Package: ffg1761 Speed: -2
   Project folder: hdl_prj Browse...
   Run This Task
   Result: Passed
   Passed Set Target Device and Synthesis Tool.

2. In **Set Target Frequency** task, specify a target frequency that you want the design to achieve by using the “Target Frequency Parameter”. For this example, set **Target Frequency (MHz)** to `200`.

3. To check you model for code generation compatibility, run the tasks in the **Prepare Model For HDL Code Generation** folder. Right-click the **Check Sample Times** task and select **Run to Selected Task**. If running a task generates a warning, select **Modify All**, and rerun the task.

4 To modify code generation , use the tasks in **Set Code Generation Options**. For example, to customize the target HDL language and the target code generation folder, use the **Set Basic Options** task. After you make changes, click **Apply**.
5 To generate code, right-click the **Generate RTL Code and Testbench** task and select **Run to Selected Task**.

Perform FPGA Synthesis and Analysis

1 In the **FPGA Synthesis and Analysis** > **Perform Synthesis and P/R** > **Run Implementation** task, clear **Skip this task** and click **Apply**. Then, right-click this task and select **Run to Selected Task**.

   The task displays the amount of resources consumed by the design and the data path delay. The slack is the difference between the required time and the arrival time for a combinational path. In this case, the slack is a positive value, which means that data arrived much earlier than the required time.

   Parsed resource report file: HDL_DUT_utilization_placed.rpt.

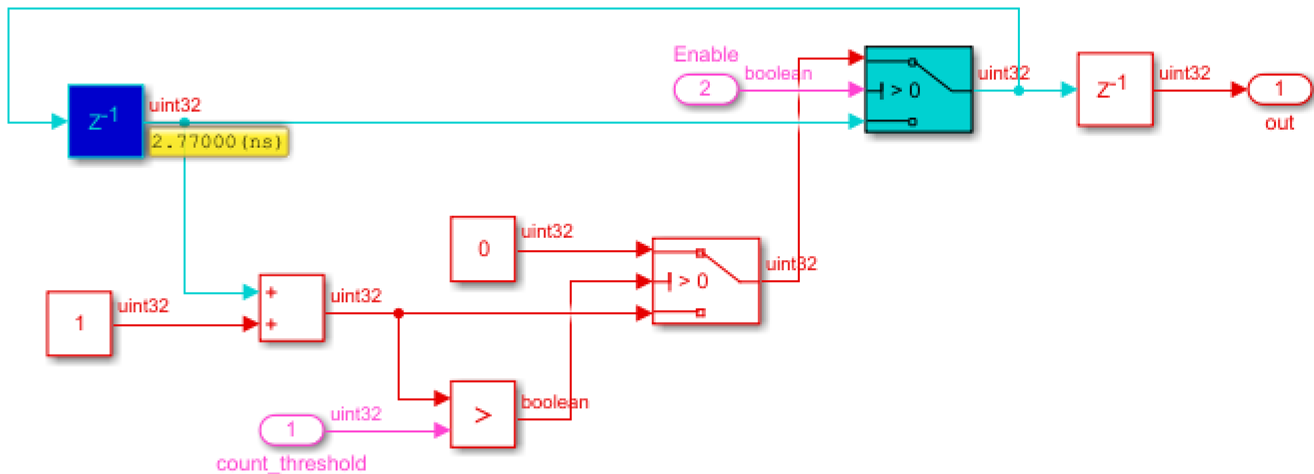
Resource summary	
Resource	**Usage**
Slice LUTs	47
Slice Registers	32
DSPs	0
Block RAM Tile	0
URAM	0

   Parsed timing report file: timing_post_route.rpt.

Timing summary	
	Value (ns)
Requirement	5
Data Path Delay	2.858
Slack	2.113

2 Right-click **Annotate Model with Synthesis Result** and select **Run to Selected Task**. If you chose `Intel Quartus Pro` or `Microsemi Libero SoC` as the **Synthesis tool**, the **Annotate Model with Synthesis Result** task is not available. To see the critical path, run the workflow to synthesis and then open the timing reports.

View the annotated critical path in the model.



Critical path is a combinational path between the input and output that has the maximum delay. The critical path delay for the counter model is `2.77ns`. The data path delay reported in **Run Implementation** task is more than the critical path because it accounts for routing delays on the target FPGA. To save resources, optimize the critical path, and improve timing of your design on the target FPGA, use speed and area optimizations in HDL Coder. To learn more, see "Speed and Area Optimizations in HDL Coder".

Run Workflow at Command Line with a Script

To run the HDL workflow at the MATLAB command prompt, export the Workflow Advisor settings to a script. To export to script, in the HDL Workflow Advisor window, select **File** > **Export to Script**. In the Export Workflow Configuration dialog box, enter a file name and save the script. See "Run HDL Workflow with a Script".

See Also

`hdladvisor` | `hdlsetuptoolpath` | `makehdl`

More About

- "Tool Setup" on page 2-2
- "Create HDL-Compatible Simulink Model" on page 3-3
- "Generate HDL Code from Simulink Model" on page 3-9

Generation of Clock Bundle Signals in HDL Coder

The clock bundle signals consist of clock, reset, and clock enable signals. During code generation, HDL Coder creates the clock bundle signals based on sequential elements such as persistent variables or Delay blocks that you use in your design. By default, a single primary clock and a single primary reset drives all sequential elements in your design.

MATLAB Code and Clock Relationship

If you use persistent variables in MATLAB, HDL Coder generates the clock bundle signals. A persistent variable is a local variable in a MATLAB function that retains its value in memory between calls to the function. For code generation, functions must initialize a persistent variable if it is empty. For more information, see `persistent`.

Consider this MATLAB code that uses a persistent variable `n`.

```
function y = persist_fcn(u)
    persistent n

    if isempty(n)
        n = 1;
    end

    y = n;
    n = n + u;
end
```

When you generate code, HDL Coder creates the clock, reset, and clock enable signals. These signals are named as `clk`, `reset`, and `clk_enable` in the HDL code. To learn how to generate HDL code, see “Basic HDL Code Generation and FPGA Synthesis from MATLAB” on page 3-35.

This code shows the generated Verilog code for the model. To match the MATLAB persistent variable behavior, the HDL code uses an always block. At the positive edge of the clock signal, when reset is low and the enable signal is high, the value `tmp` is assigned to the variable `n` after a delay of `1 ns`.

```
`timescale 1 ns / 1 ns

module persist_fcn_fixpt
          (clk, reset, clk_enable,
           u, ce_out, y);

  input   clk, reset, clk_enable;
  input   u;  // ufix1
  output  ce_out;
  output  y;  // ufix1
..

assign enb = clk_enable;

assign p4tmp_1 = {1'b0, u};
assign tmp = n + p4tmp_1;
```

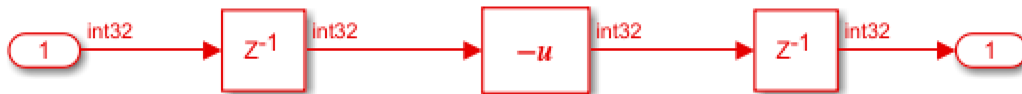
```
always @(posedge clk or posedge reset)
    begin : n_reg_process
      if (reset == 1'b1) begin
        n <= 2'b01;
      end
      else begin
        if (enb) begin
          n <= tmp;
        end
      end
    end

    assign y = n[0];
    assign ce_out = clk_enable;

endmodule  // persist_fcn_fixpt
```

See also “Persistent Variables and Persistent Array Variables”.

Simulink Model and Clock Relationship

To model sequential elements in Simulink and generate the clock bundle signals, you can use various kinds of Delay blocks, Stateflow charts, or persistent variables in MATLAB Function blocks or MATLAB System blocks. The code generator maps the sample time that you specify on your model to clock cycles in the HDL design. By default, a model is single rate which means that one sample time unit in Simulink maps to one clock cycle in the HDL code.

For example, consider this model that outputs unary minus of an input after two units of sample time. The input has `int32` as the output data type.



When you generate code, HDL Coder creates the clock, reset, and clock enable signals. These signals are named as `clk`, `reset`, and `clk_enable` in the HDL code. To learn how to generate code, see “Generate HDL Code from Simulink Model” on page 3-9.

This code shows the generated Verilog code for the model. To match the Simulink Delay block behavior, the HDL code uses an always block for each Delay block. At the positive edge of the clock signal, when reset is low and the enable signal is high, the input is passed to the output after a unit delay. One always block delays the input by `1 ns` before computing the unary minus. The other always block computes the unary minus after `1 ns`.

```
`timescale 1 ns / 1 ns

module unary_minus
          (clk, reset, clk_enable,
           In1, ce_out, Out1);
```

```
  input   clk, reset, clk_enable;
  input   signed [31:0] In1;  // int32
  output  ce_out;
  output  signed [31:0] Out1;  // int32
 ...

  assign enb = clk_enable;

  always @(posedge clk or posedge reset)
    begin : Delay_process
      if (reset == 1'b1) begin
        Delay_out1 <= 32'sb0;
      end
      else begin
        if (enb) begin
          Delay_out1 <= In1;
        end
      end
    end
 ...

  always @(posedge clk or posedge reset)
    begin : Delay2_process
       if (reset == 1'b1) begin
        Delay2_out1 <= 32'sb0;
      end
      else begin
        if (enb) begin
          Delay2_out1 <= Unary_Minus_out1;
        end
      end
    end
 ...

endmodule  // unary_minus
```

If you use different sample times in your model or enable speed and area optimizations, the model becomes multirate. To learn about clock bundle generation from multirate models, see "Code Generation from Multirate Models".

See Also

`makehdl` | `hdlsetup`

More About

- "Initialize Persistent Variables in MATLAB Functions"
- "Guidelines for Clock and Reset Signals"
- "Timing Controller for Multirate Models"
- "Multirate Model Requirements for HDL Code Generation"

Get Started with MATLAB to HDL Workflow

This example shows how to create an HDL Coder™ project and generate code from your MATLAB® design. In this example, you:

1. Create a MATLAB HDL Coder project.
2. Add the design and test bench files to the project.
3. Start the HDL Workflow Advisor for the MATLAB design.
4. Run fixed-point conversion and HDL code generation.

FIR Filter MATLAB Design

The MATLAB design `mlhdlc_sfir` is a simple symmetric FIR filter.

```
design_name = 'mlhdlc_sfir';
testbench_name = 'mlhdlc_sfir_tb';
```

Review the MATLAB design.

```
open(design_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB design: Symmetric FIR Filter
%
% Introduction:
%
% We can reduce the complexity of the FIR filter by leveraging its symmetry.
% Symmetry for an n-tap filter implies, coefficient h0 = coefficient hn-1,
% coefficient, h1 = coefficient hn-2, etc. In this case, the number of
% multipliers can be approximately halved. The key is to add the
% two data values that need to be multiplied with the same coefficient
% prior to performing the multiplication.
%
% Key Design pattern covered in this example:
% (1) Filter states represented using the persistent variables
% (2) Filter coefficients passed in as parameters

%   Copyright 2011-2019 The MathWorks, Inc.

%#codegen
function [y_out, delayed_xout] = mlhdlc_sfir(x_in,h_in1,h_in2,h_in3,h_in4)
% Symmetric FIR Filter

% declare and initialize the delay registers
persistent ud1 ud2 ud3 ud4 ud5 ud6 ud7 ud8;
if isempty(ud1)
    ud1 = 0; ud2 = 0; ud3 = 0; ud4 = 0; ud5 = 0; ud6 = 0; ud7 = 0; ud8 = 0;
end

% access the previous value of states/registers
a1 = ud1 + ud8; a2 = ud2 + ud7;
a3 = ud3 + ud6; a4 = ud4 + ud5;

% multiplier chain
m1 = h_in1 * a1; m2 = h_in2 * a2;
```

```
m3 = h_in3 * a3; m4 = h_in4 * a4;

% adder chain
a5 = m1 + m2; a6 = m3 + m4;

% filtered output
y_out = a5 + a6;

% delayout input signal
delayed_xout = ud8;

% update the delay line
ud8 = ud7;
ud7 = ud6;
ud6 = ud5;
ud5 = ud4;
ud4 = ud3;
ud3 = ud2;
ud2 = ud1;
ud1 = x_in;
end
```

FIR Filter MATLAB Test Bench

A MATLAB testbench `mlhdlc_sfir_tb` exercises the filter design.

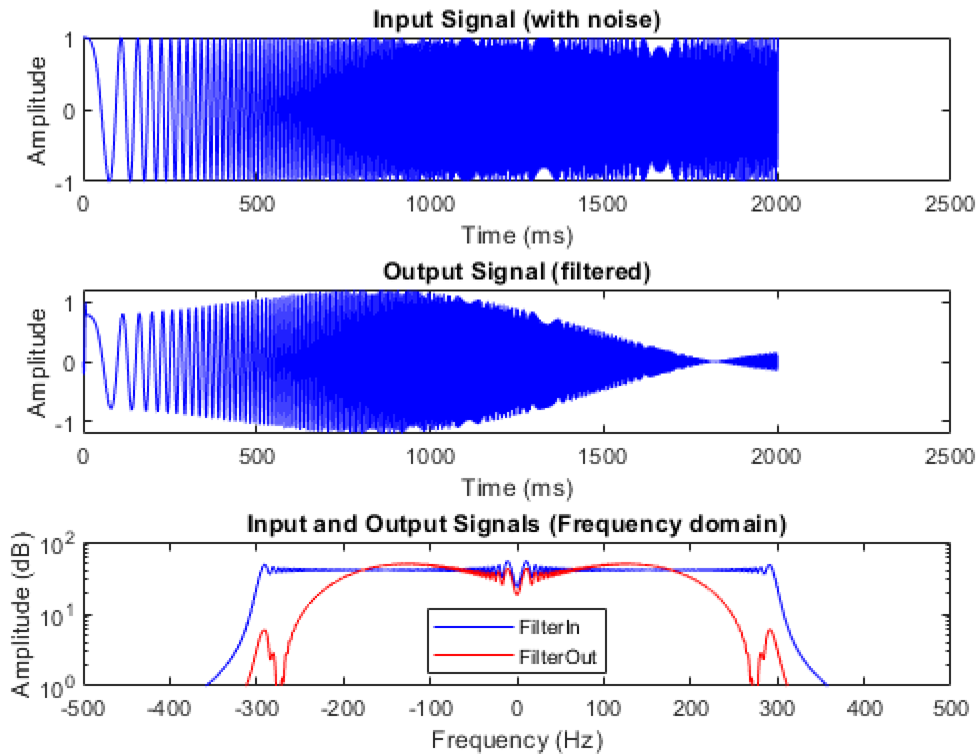
```
open(testbench_name);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB test bench for the FIR filter
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%   Copyright 2011-2019 The MathWorks, Inc.
clear mlhdlc_sfir;
T = 2;
dt = 0.001;
N = T/dt+1;
sample_time = 0:dt:T;

df = 1/dt;
sample_freq = linspace(-1/2,1/2,N).*df;

% input signal with noise
x_in = cos(2.*pi.*(sample_time).*(1+(sample_time).*75)).';

% filter coefficients
h1 = -0.1339; h2 = -0.0838; h3 = 0.2026; h4 = 0.4064;

len = length(x_in);
y_out = zeros(1,len);
x_out = zeros(1,len);

for ii=1:len
    data = x_in(ii);
    % call to the design 'mlhdlc_sfir' that is targeted for hardware
    [y_out(ii), x_out(ii)] = mlhdlc_sfir(data, h1, h2, h3, h4);
```

```
end

figure('Name', [mfilename, '_plot']);
subplot(3,1,1);
plot(1:len,x_in,'-b');
xlabel('Time (ms)')

ylabel('Amplitude')
title('Input Signal (with noise)')
subplot(3,1,2); plot(1:len,y_out,'-b');
xlabel('Time (ms)')
ylabel('Amplitude')
title('Output Signal (filtered)')

freq_fft = @(x) abs(fftshift(fft(x)));

subplot(3,1,3); semilogy(sample_freq,freq_fft(x_in),'-b');
hold on
semilogy(sample_freq,freq_fft(y_out),'-r')
hold off
xlabel('Frequency (Hz)')
ylabel('Amplitude (dB)')
title('Input and Output Signals (Frequency domain)')
legend({'FilterIn', 'FilterOut'}, 'Location','South')
axis([-500 500 1 100])
```

Test the MATLAB Algorithm

To avoid run-time errors, simulate the design by using the test bench.

```
mlhdlc_sfir_tb
```



Create a Folder and Copy Relevant Files

To copy the example files into a temporary folder, run these commands:

```
design_name = 'mlhdlc_sfir';
testbench_name = 'mlhdlc_sfir_tb';

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

Create a temporary folder and copy the MATLAB files.

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

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

Create an HDL Coder Project

To create an HDL Coder project:

1. In the MATLAB Editor, on the **Apps** tab, select **HDL Coder**. Enter `sfir_project` as **Name** of the project.

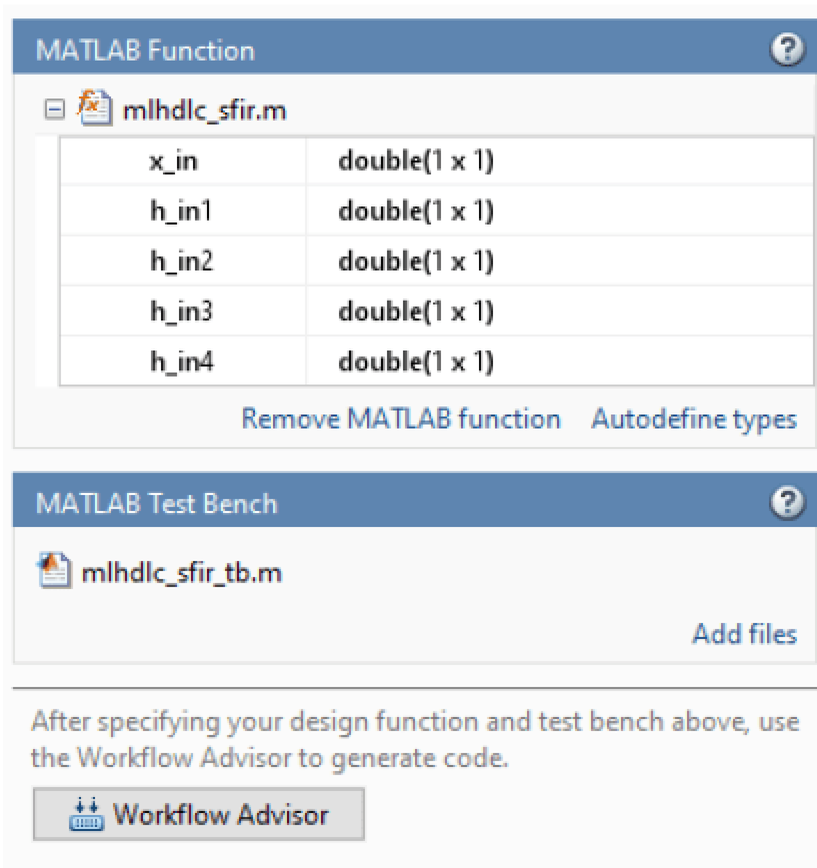
To create a project at the MATLAB command prompt, run this command:

```
coder -hdlcoder -new sfir_project
```

A `sfir_project.prj` file is created in the current folder.

2. For **MATLAB Function**, click the **Add MATLAB function** link and select the FIR filter MATLAB design `mlhdlc_sfir`. Under the **MATLAB Test Bench** section, click **Add files** and add the MATLAB test bench `mlhdlc_sfir_tb.m`.

3. Click **Autodefine types** and use the recommended types for the MATLAB design. The code generator infers the input types from the MATLAB test bench.



Run Fixed-Point Conversion and HDL Code Generation

1. Click the **Workflow Advisor** button to start the HDL Workflow Advisor.
2. Right-click the **HDL Code Generation** task and select **Run to selected task**.

The code generator runs the Workflow Advisor tasks to generate HDL code for the filter design.

- Translate your floating-point MATLAB design to a fixed-point design. To examine the generated fixed-point code from the floating-point design, click the **Fixed-Point Conversion** task. The generated fixed-point MATLAB code opens in the MATLAB editor. For details, see “Floating-Point to Fixed-Point Conversion”.
- Generate HDL code from the fixed-point MATLAB design. By default, HDL Coder generates VHDL® code. To examine the generated HDL code, click the **HDL Code Generation** task, and

then click the hyperlink to `mlhdlc_sfir_fixpt.vhd` in the Code Generation Log window. To generate Verilog® code, in the **HDL Code Generation** task, select the **Advanced** tab, and set **Language** to `Verilog`. For more information and to learn how to specify code generation options, see "MATLAB to HDL Code and Synthesis".

Clean Up Generated Files

To clean up the temporary project folder, run these commands:

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

Basic HDL Code Generation and FPGA Synthesis from MATLAB

This example shows how to create a HDL Coder™ project, generate code for your MATLAB® design, and synthesize the HDL code. In this example, you:

1. Create a MATLAB HDL Coder project.
2. Add the design and test bench files to the project.
3. Start the HDL Workflow Advisor for the MATLAB design.
4. Run fixed-point conversion and HDL code generation.
5. Generate a HDL test bench from the MATLAB test bench.
6. Verify the generated HDL code by using a HDL simulator. This example uses ModelSim® as the tool.
7. Synthesize the generated HDL code by using a synthesis tool. This example uses Xilinx® Vivado® as the tool.

FIR Filter MATLAB Design

The MATLAB design `mlhdlc_sfir` is a simple symmetric FIR filter.

```
design_name = 'mlhdlc_sfir';
testbench_name = 'mlhdlc_sfir_tb';
```

Review the MATLAB design.

```
open(design_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB design: Symmetric FIR Filter
%
% Introduction:
%
% We can reduce the complexity of the FIR filter by leveraging its symmetry.
% Symmetry for an n-tap filter implies, coefficient h0 = coefficient hn-1,
% coefficient, h1 = coefficient hn-2, etc. In this case, the number of
% multipliers can be approximately halved. The key is to add the
% two data values that need to be multiplied with the same coefficient
% prior to performing the multiplication.
%
% Key Design pattern covered in this example:
% (1) Filter states represented using the persistent variables
% (2) Filter coefficients passed in as parameters

%   Copyright 2011-2019 The MathWorks, Inc.

%#codegen
function [y_out, delayed_xout] = mlhdlc_sfir(x_in,h_in1,h_in2,h_in3,h_in4)
% Symmetric FIR Filter

% declare and initialize the delay registers
persistent ud1 ud2 ud3 ud4 ud5 ud6 ud7 ud8;
if isempty(ud1)
    ud1 = 0; ud2 = 0; ud3 = 0; ud4 = 0; ud5 = 0; ud6 = 0; ud7 = 0; ud8 = 0;
end
```

```
% access the previous value of states/registers
a1 = ud1 + ud8; a2 = ud2 + ud7;
a3 = ud3 + ud6; a4 = ud4 + ud5;

% multiplier chain
m1 = h_in1 * a1; m2 = h_in2 * a2;
m3 = h_in3 * a3; m4 = h_in4 * a4;

% adder chain
a5 = m1 + m2; a6 = m3 + m4;

% filtered output
y_out = a5 + a6;

% delayout input signal
delayed_xout = ud8;

% update the delay line
ud8 = ud7;
ud7 = ud6;
ud6 = ud5;
ud5 = ud4;
ud4 = ud3;
ud3 = ud2;
ud2 = ud1;
ud1 = x_in;
end
```

FIR Filter MATLAB Test Bench

A MATLAB testbench `mlhdlc_sfir_tb` exercises the filter design by using a representative input range. Review the MATLAB test bench `mlhdlc_sfir_tb`.

```
open(testbench_name);
```

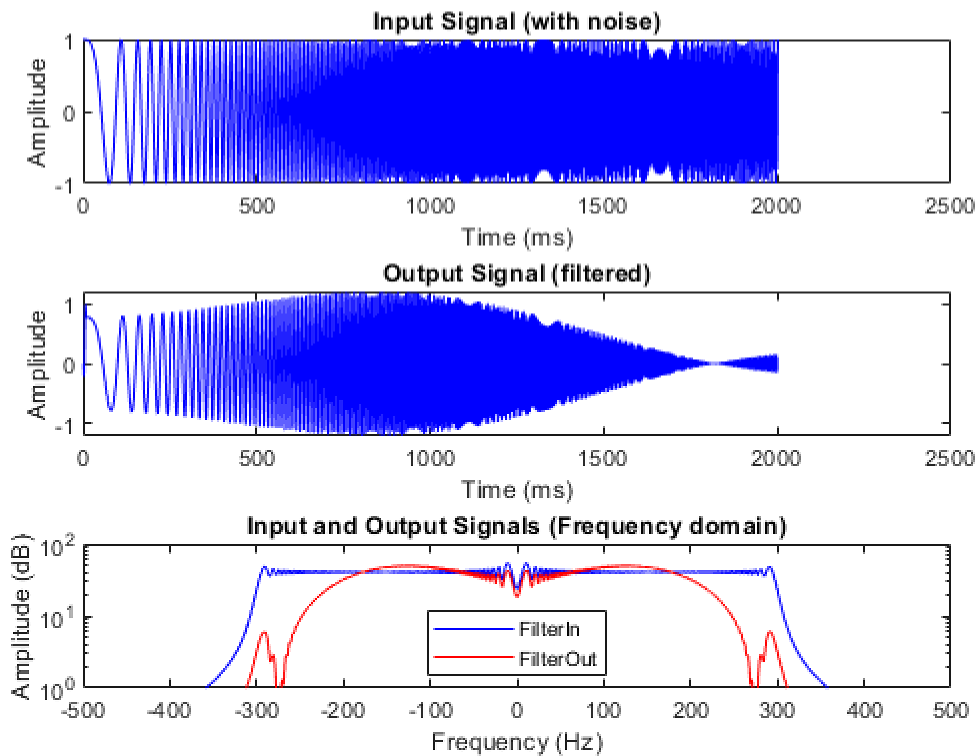
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB test bench for the FIR filter
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%   Copyright 2011-2019 The MathWorks, Inc.
clear mlhdlc_sfir;
T = 2;
dt = 0.001;
N = T/dt+1;
sample_time = 0:dt:T;

df = 1/dt;
sample_freq = linspace(-1/2,1/2,N).*df;

% input signal with noise
x_in = cos(2.*pi.*(sample_time).*(1+(sample_time).*75)).';

% filter coefficients
h1 = -0.1339; h2 = -0.0838; h3 = 0.2026; h4 = 0.4064;
```

```
len = length(x_in);
y_out = zeros(1,len);
x_out = zeros(1,len);

for ii=1:len
    data = x_in(ii);
    % call to the design 'mlhdlc_sfir' that is targeted for hardware
    [y_out(ii), x_out(ii)] = mlhdlc_sfir(data, h1, h2, h3, h4);
end

figure('Name', [mfilename, '_plot']);
subplot(3,1,1);
plot(1:len,x_in,'-b');
xlabel('Time (ms)')

ylabel('Amplitude')
title('Input Signal (with noise)')
subplot(3,1,2); plot(1:len,y_out,'-b');
xlabel('Time (ms)')
ylabel('Amplitude')
title('Output Signal (filtered)')

freq_fft = @(x) abs(fftshift(fft(x)));

subplot(3,1,3); semilogy(sample_freq,freq_fft(x_in),'-b');
hold on
semilogy(sample_freq,freq_fft(y_out),'-r')
hold off
xlabel('Frequency (Hz)')
ylabel('Amplitude (dB)')
title('Input and Output Signals (Frequency domain)')
legend({'FilterIn', 'FilterOut'}, 'Location','South')
axis([-500 500 1 100])
```

Test the Original MATLAB Algorithm

To avoid run-time errors, simulate the design by using the test bench.

```
mlhdlc_sfir_tb
```



Create a Folder and Copy Relevant Files

To copy the example files into a temporary folder, run these commands:

```
design_name = 'mlhdlc_sfir';
testbench_name = 'mlhdlc_sfir_tb';
```

Create a temporary folder

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderder
mlhdlc_temp_dir = [tempdir 'mlhdlc_sfir'];
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);
```

Copy the MATLAB files.

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

Set Up HDL Simulator and Synthesis Tool Path

If you want to synthesize the generated HDL code, before you use HDL Coder to generate code, set up your synthesis tool path. To set up the path to your synthesis tool, use the hdlsetuptoolpath function. For example, if your synthesis tool is Xilinx Vivado:

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath',...
               'C:\Xilinx\Vivado\2018.3\bin\vivado.bat');
```

You must have already installed Xilinx Vivado. To check your Xilinx Vivado synthesis tool setup, launch the tool by running this command:

```
!vivado
```

If you want to simulate the generated HDL code by using a HDL test bench, you can use an HDL simulator such as ModelSim®. You must have already installed the HDL simulator.

Create an HDL Coder Project

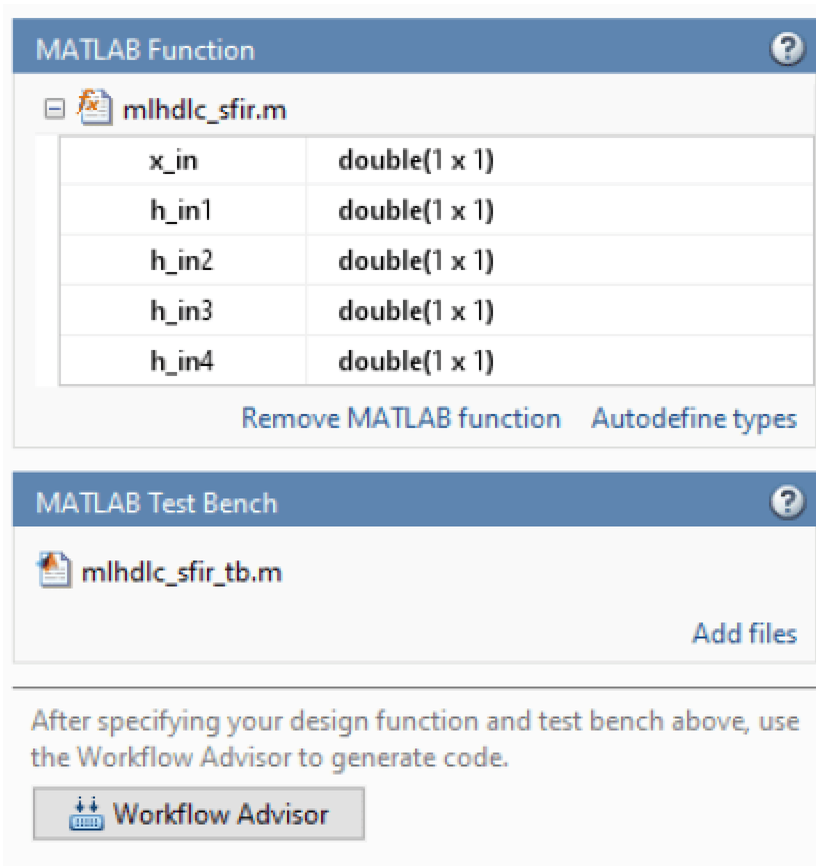
To create an HDL Coder project:

1. Create a project by running this command:

```
coder -hdlcoder -new sfir_project
```

2. For **MATLAB Function**, add the MATLAB design `mlhdlc_sfir`. Add `mlhdlc_sfir_tb.m` as the MATLAB test bench.

3. Click **Autodefine types** and use the recommended types for the MATLAB design. The code generator infers data types by running the test bench.



Create Fixed-Point Versions of Algorithm and Test Bench

1. Click the **Workflow Advisor** button to open the Workflow Advisor. You see that the **Define Input Types** task has passed.
2. Run the **Fixed-Point Conversion** task. The **Fixed-Point Conversion** tool opens in the right pane.

When you run fixed-point conversion, to propose fraction lengths for floating-point data types, HDL Coder uses the **Default word length**. In this tutorial, the **Default word length** is `14`. The advisor provides a default **Safety Margin for Simulation Min/Max** of `0%`. The advisor adjusts the range of the data by this safety factor. For example, a value of 4 specifies that you want a range of at least 4 percent larger. See also “Floating-Point to Fixed-Point Conversion”.

Select Code Generation Options and Generate HDL Code

Before you generate HDL code, if you want to deploy the code onto a target platform, specify the synthesis tool. In the **Code Generation Target** task, leave **Workflow** to `Generic ASIC/FPGA` and specify `Xilinx Vivado` as the **Synthesis Tool**. If you don't see the synthesis tool, click **Refresh list**. Run this task.

In the **HDL Code Generation** task, by using the tabs on the right side of this task, you can specify additional code generation options.

1. By default, HDL Coder generates VHDL® code. To generate Verilog code, in the **Target** tab, choose `Verilog` as the **Language**.
2. To generate a code generation report with comments and traceability links, in the **Coding style** tab, select **Include MATLAB source code as comments** and 'Generate report*.
3. To optimize your design, you can use the distributed pipelining optimization. In the **Optimizations** tab, specify `1` for **Input pipelining** and **Output pipelining** and then select **Distribute pipeline registers**. To learn more, see “Distributed Pipelining”.
4. Click **Run** to generate Verilog code.

Examine the log window and click the links to explore the generated code and the reports.

Generate HDL Test Bench and Simulate the Generated Code

HDL Coder generates a HDL test bench, runs the HDL test bench by using a HDL simulator, and verifies whether the HDL simulation matches the numerics and latency of the fixed-point MATLAB simulation.

To generate a HDL test bench and simulate the generated code, in the **HDL Verification > Verify with HDL Test Bench** task:

1. In the **Output Settings** tab, select **Generate HDL test bench**.
2. To simulate the generated test bench, set the **Simulation Tool** to `ModelSim`. You must have already installed ModelSim.
3. To specify generation of HDL test bench code and test bench data in separate files, in the **Test Bench Options** tab, select **Multi-file test bench**.
4. Click the **Run** button.

The task generates an HDL test bench, then simulates the fixed-point design by using the selected simulation tool, and generates a compilation report and a simulation report.

Synthesize Generated HDL Code

HDL Coder synthesizes the HDL code on the target platform and generates area and timing reports for your design based on the target device that you specify.

To synthesize the generated HDL code:

1. Run the **Create project** task.

This task creates a Xilinx Vivado synthesis project for the HDL code. HDL Coder uses this project in the next task to synthesize the design.

2. Select and run the **Run Synthesis** task.

This task launches the synthesis tool in the background, opens the synthesis project, compiles the HDL code, synthesizes the design, and generates netlists and area and timing reports.

3. Select and run the **Run Implementation** task.

This task launches the synthesis tool in the background, runs place and route on the design, and generates pre- and post-route timing information for use in critical path analysis and back annotation of your source model.

Clean Up Generated Files

To clean up the temporary project folder, run these commands:

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

Generate HDL Code from MATLAB Code Using the Command Line Interface

This example shows how to use the HDL Coder™ command line interface to generate HDL code from MATLAB® code, including floating-point to fixed-point conversion and FPGA programming file generation.

Overview

HDL code generation with the command-line interface has the following basic steps:

1. Create a `fixpt` coder config object. (Optional)
2. Create an `hdl` coder config object.
3. Set config object parameters. (Optional)
4. Run the codegen command to generate code.

The HDL Coder command-line interface can use two coder config objects with the codegen command. The optional `fixpt` coder config object configures the floating-point to fixed-point conversion of your MATLAB code. The `hdl` coder config object configures HDL code generation and FPGA programming options.

In this example, we explore different ways you can configure your floating-point to fixed-point conversion and code generation.

The example code implements a discrete-time integrator and its test bench.

Copy the Design and Test Bench Files Into a Temporary Folder

Execute the following code to copy the design and test bench files into a temporary folder:

```
close all;
design_name = 'mlhdlc_dti';
testbench_name = 'mlhdlc_dti_tb';

mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdem
mlhdlc_temp_dir = [tempdir 'mlhdlc_dti'];

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

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

Basic Code Generation With Floating-Point to Fixed-Point Conversion

You can generate HDL code and convert the design from floating-point to fixed-point using the default settings.

You need only your design name, `mlhdlc_dti`, and test bench name, `mlhdlc_dti_tb`:

```
close all;
```

```
% Create a 'fixpt' config with default settings
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'mlhdlc_dti_tb';

% Create an 'hdl' config with default settings
hdlcfg = coder.config('hdl'); %#ok<NASGU>
```

After setting up `fixpt` and `hdl` config objects, run the following codegen command to perform floating-point to fixed-point conversion, and generate HDL code.

```
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_dti
```

If your design already uses fixed-point types and functions, you can skip fixed-point conversion:

```
hdlcfg = coder.config('hdl'); % Create an 'hdl' config with default settings
hdlcfg.TestBenchName = 'mlhdlc_dti_tb';
codegen -config hdlcfg mlhdlc_dti
```

The rest of this example describes how to configure code generation using the `hdl` and `fixpt` objects.

Create a Floating-Point to Fixed-Point Conversion Config Object

To perform floating-point to fixed-point conversion, you need a `fixpt` config object.

Create a `fixpt` config object and specify your test bench name:

```
close all;
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'mlhdlc_dti_tb';
```

Set Fixed-Point Conversion Type Proposal Options

The code generator can propose fixed-point types based on your choice of either word length or fraction length. These two options are mutually exclusive.

Base the proposed types on a word length of `24`:

```
fixptcfg.DefaultWordLength = 24;
fixptcfg.ProposeFractionLengthsForDefaultWordLength = true;
```

Alternatively, you can base the proposed fixed-point types on fraction length. The following code configures the coder to propose types based on a fraction length of `10`:

```
fixptcfg.DefaultFractionLength = 10;
fixptcfg.ProposeWordLengthsForDefaultFractionLength = true;
```

Set the Safety Margin

The code generator increases the simulation data range on which it bases its fixed-point type proposal by the safety margin percentage. For example, the default safety margin is `4`, which increases the simulation data range used for fixed-point type proposal by `4%`.

Set the SafetyMargin to `10%`:

```
fixptcfg.SafetyMargin = 10;
```

Enable Data Logging

The code generator runs the test bench with the design before and after floating-point to fixed-point conversion. You can enable simulation data logging to plot the quantization effects of the new fixed-point data types.

Enable data logging in the `fixpt` config object:

```
fixptcfg.LogIOForComparisonPlotting = true;
```

View the Numeric Type Proposal Report

Configure the code generator to launch the type proposal report once the fixed-point types have been proposed:

```
fixptcfg.LaunchNumericTypesReport = true;
```

Create an HDL Code Generation Config Object

To generate code, you must create an `hdl` config object and set your test bench name:

```
hdlcfg = coder.config('hdl');
hdlcfg.TestBenchName = 'mlhdlc_dti_tb';
```

Set the Target Language

You can generate either VHDL or Verilog code. HDL Coder generates VHDL code by default. To generate Verilog code:

```
hdlcfg.TargetLanguage = 'Verilog';
```

Generate HDL Test Bench Code

Generate an HDL test bench from your MATLAB® test bench:

```
hdlcfg.GenerateHDLTestBench = true;
```

Simulate the Generated HDL Code Using an HDL Simulator

If you want to simulate your generated HDL code using an HDL simulator, you must also generate the HDL test bench.

Enable HDL simulation and use the ModelSim simulator:

```
hdlcfg.SimulateGeneratedCode = true;
hdlcfg.SimulationTool = 'ModelSim'; %  or 'ISIM'
```

Generate an FPGA Programming File

You can generate an FPGA programming file if you have a synthesis tool set up. Enable synthesis, specify a synthesis tool, and specify an FPGA:

```
% Enable Synthesis.
hdlcfg.SynthesizeGeneratedCode = true;

% Configure Synthesis tool.
hdlcfg.SynthesisTool = 'Xilinx ISE'; %  or 'Altera Quartus II';
hdlcfg.SynthesisToolChipFamily = 'Virtex7';
```

```
hdlcfg.SynthesisToolDeviceName = 'xc7vh580t';
hdlcfg.SynthesisToolPackageName = 'hcg1155';
hdlcfg.SynthesisToolSpeedValue = '-2G';
```

Run Code Generation

Now that you have your `fixpt` and `hdl` config objects set up, run the codegen command to perform floating-point to fixed-point conversion, generate HDL code, and generate an FPGA programming file:

```
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_dti
```

Generating Modular HDL Code for Functions

This example shows how to generate modular HDL code from MATLAB® code that contains functions.

By default, HDL Coder™ inlines the body of all MATLAB functions that are called inside the body of the top-level design function. This inlining results in the generation of a single file that contains the HDL code for the functions. To generate modular HDL code, use the **Generate instantiable code for functions** setting. When you enable this setting, HDL Coder generates a single VHDL® entity or Verilog® module for each function.

LMS Filter MATLAB Design

The MATLAB design used in the example is an implementation of an LMS (Least Mean Squares) filter. The LMS filter is a class of adaptive filter that identifies an FIR filter signal that is embedded in the noise. The LMS filter design implementation in MATLAB consists of a top-level function `mlhdlc_lms_fcn` that calculates the optimal filter coefficients to reduce the difference between the output signal and the desired signal.

```
design_name = 'mlhdlc_lms_fcn';
testbench_name = 'mlhdlc_lms_fir_id_tb';
```

Review the MATLAB design:

```
open(design_name);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB Design: Adaptive Noise Canceler algorithm using Least Mean Square
% (LMS) filter implemented in MATLAB
%
% Key Design pattern covered in this example:
% (1) Use of function calls
% (2) Function inlining vs instantiation knobs available in the coder
% (3) Use of system objects in the testbench to stream test vectors into the design
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%#codegen
function [filtered_signal, y, fc] = mlhdlc_lms_fcn(input, ...
                                        desired, step_size, reset_weights)
% 'input'  : The signal from Exterior Mic which records the ambient noise.
% 'desired': The signal from Pilot's Mic which includes
%            original music signal and the noise signal
% 'err_sig': The difference between the 'desired' and the filtered 'input'
%           It represents the estimated music signal (output of this block)
%
% The LMS filter is trying to retrieve the original music signal('err_sig')
% from Pilot's Mic by filtering the Exterior Mic's signal and using it to
% cancel the noise in Pilot's Mic. The coefficients/weights of the filter
% are updated(adapted) in real-time based on 'input' and 'err_sig'.

% register filter coefficients
persistent filter_coeff;
if isempty(filter_coeff)
    filter_coeff = zeros(1, 40);
end
```

```
% Variable Filter: Call 'tapped_delay_fcn' function on path to create
% 40-step tapped delay
delayed_signal = mtapped_delay_fcn(input);

% Apply filter coefficients
weight_applied = delayed_signal .* filter_coeff;

% Call treesum function on matlab path to sum up the results
filtered_signal = mtreesum_fcn(weight_applied);

% Output estimated Original Signal
td = desired;
tf = filtered_signal;
esig = td - tf;
y = esig;

% Update Weights: Call 'update_weight_fcn' function on MATLAB path to
% calculate the new weights
updated_weight = update_weight_fcn(step_size, esig, delayed_signal, ...
                                   filter_coeff, reset_weights);

% update filter coefficients register
filter_coeff = updated_weight;
fc = filter_coeff;

function y = mtreesum_fcn(u)
%Implement the 'sum' function without a for-loop
%  y = sum(u);

%  The loop based implementation of 'sum' function is not ideal for
%  HDL generation and results in a longer critical path.
%  A tree is more efficient as it results in
%  delay of log2(N) instead of a delay of N delay

%  This implementation shows how to explicitly implement the vector sum in
%  a tree shape to enable hardware optimizations.

%  The ideal way to code this generically for any length of 'u' is to use
%  recursion but it is not currently supported by MATLAB Coder


% NOTE: To instruct MATLAB Coder to compile an external function,
% add the following compilation directive or pragma to the function code
%#codegen

% This implementation is hardwired for a 40tap filter.

level1 = vsum(u);
level2 = vsum(level1);
level3 = vsum(level2);
level4 = vsum(level3);
level5 = vsum(level4);
level6 = vsum(level5);
y = level6;

function output = vsum(input)

coder.inline('always');
```

```
vt = input(1:2:end);

for i = int32(1:numel(input)/2)
    k = int32(i*2);
    vt(i) = vt(i) + input(k);
end

output = vt;

function tap_delay = mtapped_delay_fcn(input)
% The Tapped Delay function delays its input by the specified number
% of sample periods, and outputs all the delayed versions in a vector
% form. The output includes current input

% NOTE: To instruct MATLAB Coder to compile an external function,
% add the following compilation directive or pragma to the function code
%#codegen

persistent u_d;
if isempty(u_d)
    u_d = zeros(1,40);
end


u_d = [u_d(2:40), input];

tap_delay = u_d;

function weights = update_weight_fcn(step_size, err_sig, ...
            delayed_signal, filter_coeff, reset_weights)
% This function updates the adaptive filter weights based on LMS algorithm

%   Copyright 2007-2015 The MathWorks, Inc.

% NOTE: To instruct MATLAB Coder to compile an external function,
% add the following compilation directive or pragma to the function code
%#codegen

step_sig = step_size .* err_sig;
correction_factor = delayed_signal .* step_sig;
updated_weight = correction_factor + filter_coeff;

if reset_weights
    weights = zeros(1,40);
else
    weights = updated_weight;
end
```

The MATLAB function is modular and uses functions:

- `mtapped_delay_fcn` to calculate delayed versions of the input signal in vector form.
- `mtreesum_fcn` to calculate the sum of the applied weights in a tree structure. The individual sum is calculated by using a `vsum` function.
- `update_weight_fcn` to calculate the updated filter weights based on the least mean square algorithm.

LMS Filter MATLAB Test Bench

Review the MATLAB test bench:

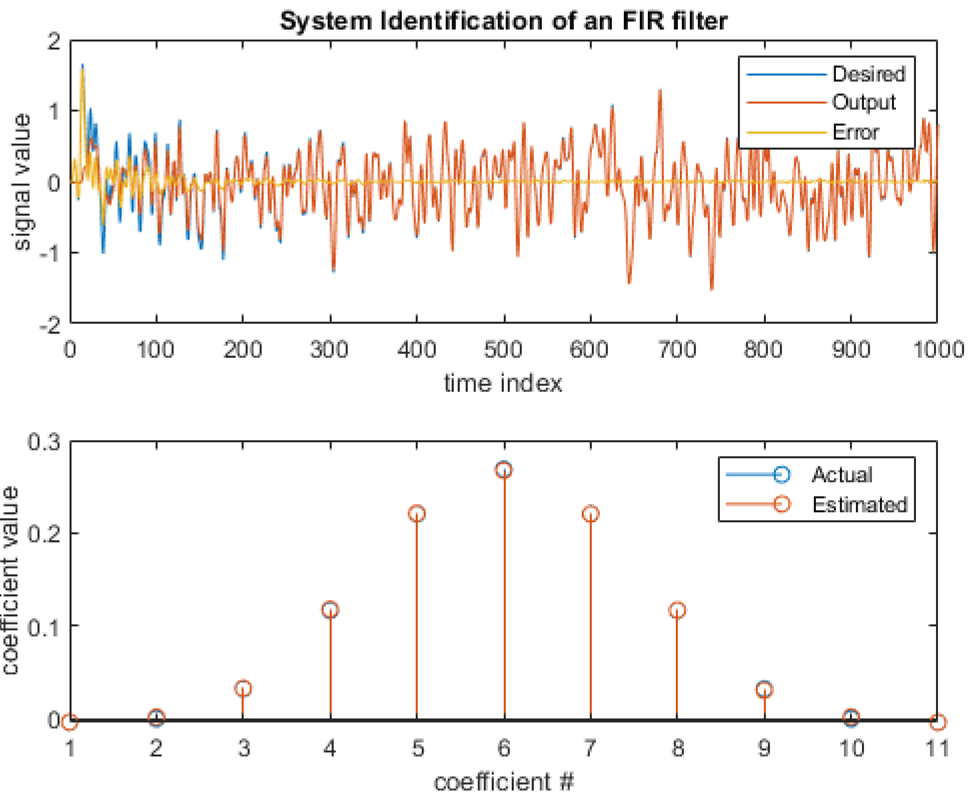
```
open(testbench_name);
```

```
clear ('mlhdlc_lms_fcn');
% returns an adaptive FIR filter System object, HLMS, that computes the
% filtered output, filter error, and the filter weights for a given input
% and desired signal using the Least MeanSquares (LMS) algorithm.

% Copyright 2011-2019 The MathWorks, Inc.

stepSize = 0.01;
reset_weights =false;

hfilt = dsp.FIRFilter;                    % System to be identified
hfilt.Numerator = fir1(10, .25);

rng('default');                           % always default to known state
x = randn(1000,1);                        % input signal
d = step(hfilt, x) + 0.01*randn(1000,1);  % desired signal

hSrc = dsp.SignalSource(x);
hDesiredSrc = dsp.SignalSource(d);

hOut = dsp.SignalSink;
hErr = dsp.SignalSink;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Call to the design
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
while (~isDone(hSrc))
    [y, e, w] = mlhdlc_lms_fcn(step(hSrc), step(hDesiredSrc), ...
                stepSize, reset_weights);
    step(hOut, y);
    step(hErr, e);
end

figure('Name', [mfilename, '_plot']);
subplot(2,1,1), plot(1:1000, [d,hOut.Buffer,hErr.Buffer]);
title('System Identification of an FIR filter');
legend('Desired', 'Output', 'Error');
xlabel('time index'); ylabel('signal value');
subplot(2,1,2); stem([hfilt.Numerator.', w(end-10:end).']);
legend('Actual','Estimated');
xlabel('coefficient #'); ylabel('coefficient value');
```

Test the MATLAB Algorithm

To avoid run-time errors, simulate the design with the test bench.

```
mlhdlc_lms_fir_id_tb
```



Create a Folder and Copy Relevant Files

Before you generate HDL code for the MATLAB design, copy the design and test bench files to a writeable folder. These commands copy the files to a temporary folder.

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

Create a temporary folder and copy the MATLAB files.

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

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

Create an HDL Coder Project

To generate HDL code from a MATLAB design:

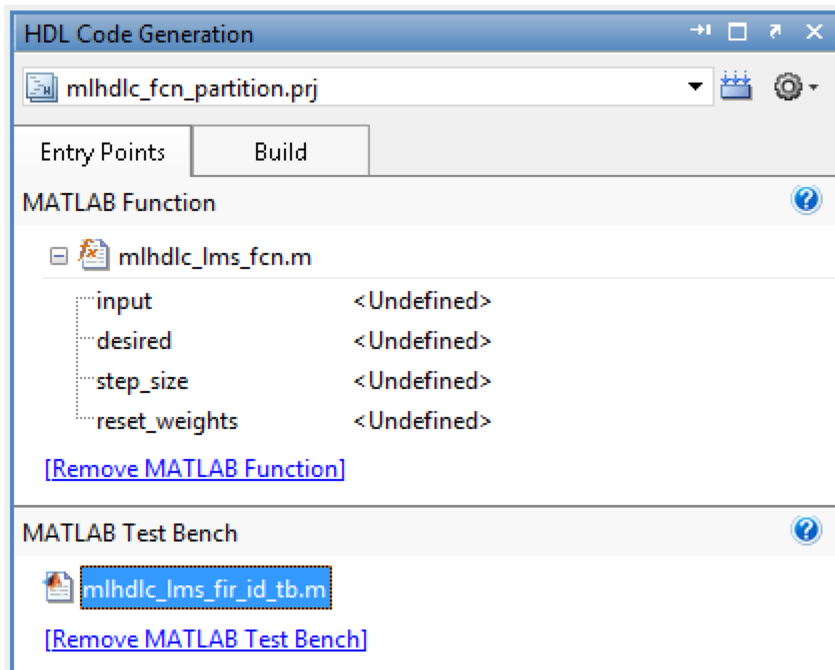
1. Create a HDL Coder project:

```
coder -hdlcoder -new mlhdlc_fcn_partition
```

2. Add the file `mlhdlc_lms_fcn.m` to the project as the **MATLAB Function** and `mlhdlc_lms_fir_id_tb.m` as the **MATLAB Test Bench**.

3. Click **Autodefine types** to use the recommended types for the inputs and outputs of the MATLAB function `mlhdlc_lms_fcn`.

Refer to “Get Started with MATLAB to HDL Workflow” on page 3-29 for a more complete tutorial on creating and populating MATLAB HDL Coder projects.



Run Fixed-Point Conversion and HDL Code Generation

1. Click the **Workflow Advisor** button to start the Workflow Advisor.
2. Right click the **HDL Code Generation** task and select **Run to selected task**.

A single HDL file `mlhdlc_lms_fcn_FixPt.vhd` is generated for the MATLAB design. The VHDL code for all functions in the MATLAB design is inlined into this file.

Generate Instantiable HDL Code

1. In the **Advanced** tab, select the **Generate instantiable code for functions** check box.
2. Click the **Run** button to rerun the **HDL Code Generation** task.

You see multiple HDL files that contain the generated code for the top-level function and the functions that are called inside the top-level function. See also “Generate Instantiable Code for Functions”.

Control Inlining For Each Function

In some cases, you may want to inline the HDL code for helper functions and utilities and then instantiate them. To locally control inlining of such functions, use the `coder.inline` pragma in the MATLAB code.

To inline a function in the generated code, place this directive inside that function:

```
coder.inline('always')
```

To prevent inlining of a function in the generated code, place this directive inside that function:

```
coder.inline('never')
```

To let the code generator determine whether to inline a function in the generated code, place this directive inside that function:

```
coder.inline('default')
```

To learn how to use `coder.inline` pragma, enter:

```
help coder.inline
```

Limitations for Instantiating HDL Code from Functions

- Function calls inside conditional expressions and for loops are inlined and are not instantiated.
- Functions with states are inlined.

Clean Up Generated Files

To clean up the temporary project folder, run these commands:

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

System Design with HDL Code Generation from MATLAB and Simulink

This example shows how to generate a MATLAB Function block from a MATLAB® design for system simulation, code generation, and FPGA programming in Simulink®.

Introduction

HDL Coder can generate HDL code from both MATLAB® and Simulink®. The coder can also generate a Simulink® component, the MATLAB Function block, from your MATLAB code.

This capability enables you to:

1. Design an algorithm in MATLAB;
2. Generate a MATLAB Function block from your MATLAB design;
3. Use the MATLAB component in a Simulink model of the system;
4. Simulate and optimize the system model;
5. Generate HDL code; and
6. Program an FPGA with the entire system design.

In this example, you will generate a MATLAB Function block from MATLAB code that implements a FIR filter.

MATLAB Design

The MATLAB code used in the example is a simple FIR filter. The example also shows a MATLAB testbench that exercises the filter.

```
design_name = 'mlhdlc_fir';
testbench_name = 'mlhdlc_fir_tb';
```

1. Design: mlhdlc_fir
2. Test Bench: mlhdlc_fir_tb

Create a New Folder and Copy Relevant Files

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

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

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

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

Simulate the Design

To simulate the design with the test bench prior to code generation to make sure there are no runtime errors, enter the following command:

mlhdlc_fir_tb

Create a New Project

To create a new HDL Coder project, enter the following command:

```
coder -hdlcoder -new fir_project
```

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

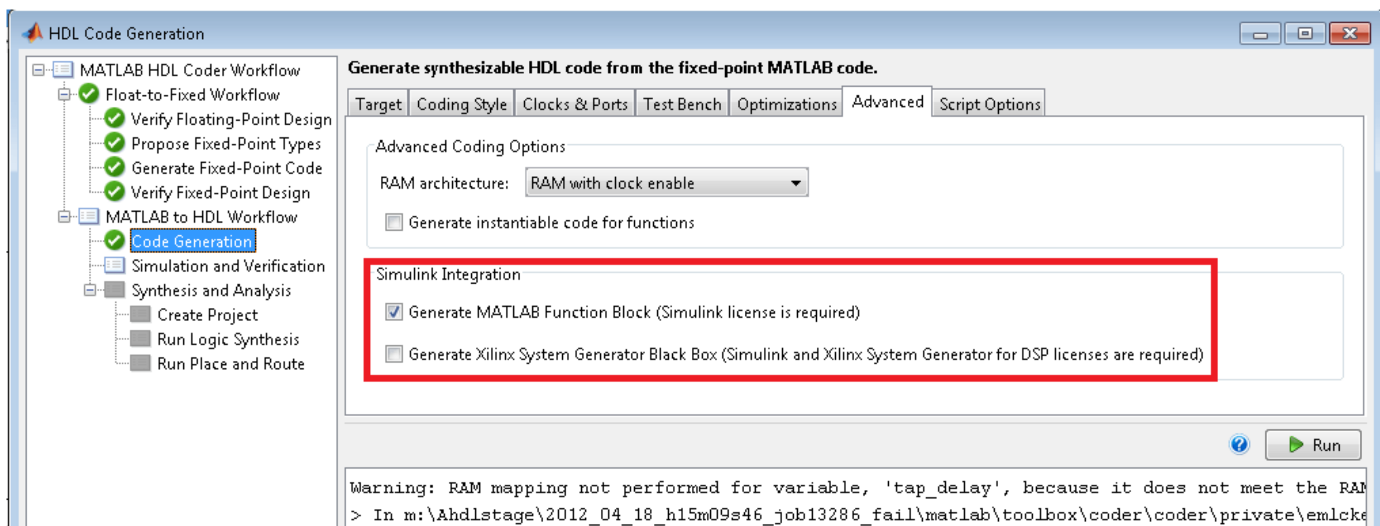
Click the Workflow Advisor button to launch the HDL Workflow Advisor.

Enable the MATLAB Function Block Option

To generate a MATLAB Function block from a MATLAB HDL design, you must have a Simulink license. If the following command returns '1', Simulink is available:

```
license('test', 'Simulink')
```

In the HDL Workflow Advisor Advanced tab, enable the Generate MATLAB Function Block option.



Run Floating-Point to Fixed-Point Conversion and Generate Code

To generate a MATLAB Function block, you must also convert your design from floating-point to fixed-point.

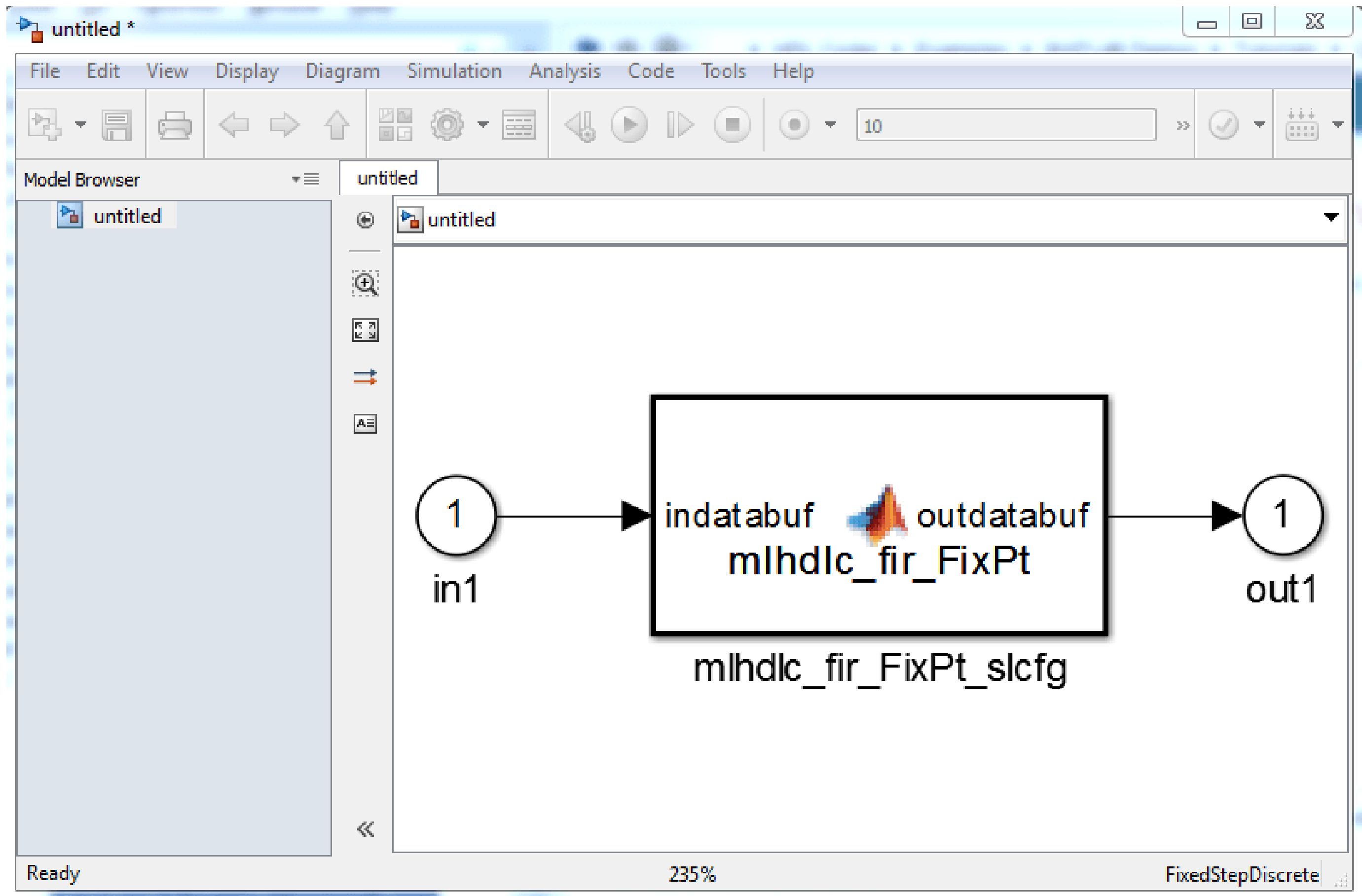
Right-click the 'Code Generation' step and choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

Examine the Generated MATLAB Function Block

An untitled model opens after HDL code generation. It has a MATLAB Function block containing the fixed-point MATLAB code from your MATLAB HDL design. HDL Coder automatically applies settings to the model and MATLAB Function block so that they can simulate in Simulink and generate HDL code.

To generate HDL code from the MATLAB Function block, enter the following command:

```
makehdl('untitled');
```



You can rename and save the new block to use in a larger Simulink design.

Clean Up the Generated Files

You can run the following commands to clean up the temporary project folder.

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