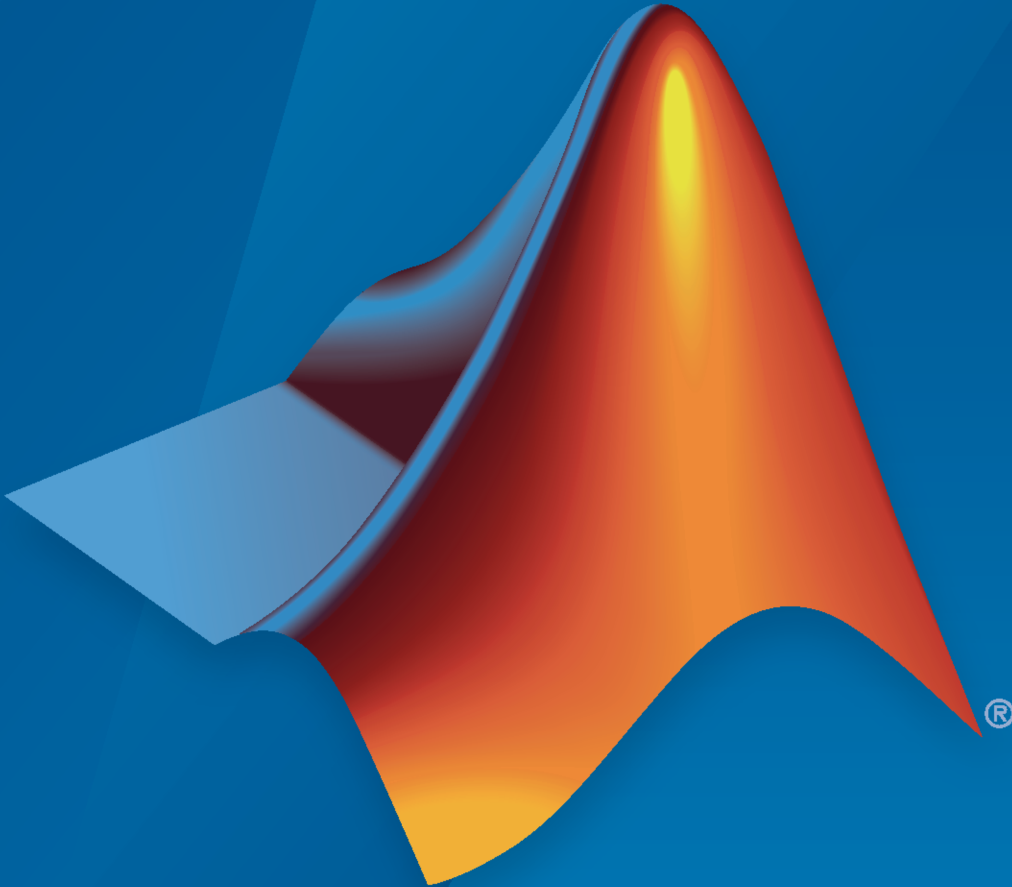
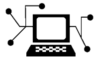
HDL Coder™ Release Notes



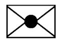
MATLAB® & SIMULINK®

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™ Release Notes

© COPYRIGHT 2012–2021 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.

Contents

R2021b

Model and Architecture Design **1-2**

RAM style attributes for Intel/Altera and Microchip **1-2**
HDL code check for trigonometric blocks **1-2**
Timestamp macro in custom file header comments **1-2**
Enhanced multiple enumeration in Verilog **1-2**
HDL Industry Coding Standard check for the presence of assignments to the same variable in multiple cascaded conditional regions **1-3**
Layout choices for model generation **1-3**

Block Enhancements **1-5**

Newton-Raphson algorithm for Math Reciprocal block **1-5**
Magnitude square function in Math Function block **1-5**
Half-precision data types for MATLAB Function block **1-5**
Double-Precision data types for Logarithmic block **1-5**
For-Generate loops for Reshape and Concat blocks **1-5**
Fixed-point output types for Divide block and Reciprocal block **1-6**
Enhanced HDL math library **1-6**
4-D and 5-D lookup table support **1-7**
Improved denormal optimizations for half-precision data types **1-7**
Improved multiplier partitioning DSP QoR **1-7**
Reset minimization in Native Floating-Point (NFP) for ASIC **1-7**
Set-Reset (SR) flip-flops **1-7**
HDL Code Generation for Discrete State-Space block **1-8**
Trigger and event modes for subsystems, MATLAB Function blocks, and Stateflow blocks **1-8**
Wireless HDL Toolbox Reference Applications: Implement 5G NR SIB1 recovery, WLAN receiver, and DVB-S2 PL header recovery **1-8**
Wireless HDL Toolbox Blocks: Model WLAN LDPC decoder, CCSDS RS decoder, DVBS2 symbol demodulator, and APP decoder **1-8**
Multipixel-Multicomponent Video Streaming: Implement color space conversion and demosaic interpolation algorithms for high-frame-rate color video **1-9**
Reflection Padding: Pad image frames by reflecting around the edge pixel **1-9**

Code Generation and Verification **1-11**

Code View: View your generated HDL code directly in Simulink model window **1-11**
Stateflow multicycle path enhancements **1-11**
Register-to-register path info option not recommended in HDL Coder . . . **1-12**
Execute chart at initialization option for Stateflow charts **1-12**

HDL code generation performance improvement for matrix multiplication 1-12

Speed and Area Optimizations 1-14

Enhanced sharing and streaming optimizations for matrix-types 1-14
User control for tunable parameter processing and improve code generation time 1-14
Improved zero-protection in Simulink-to-HDL 1-14
Minimize intermediate initialization of variables in generated HDL code 1-15
Improved optimizations for conditional subsystems 1-16
Delay-balancing behavior standardization in BalanceDelays=off network 1-17
Lookup Table blocks mapping to RAM and adaptive pipelining 1-17

IP Core Generation and Hardware Deployment 1-18

Microsemi Libero System On A Chip (SoC) support for IP core generation workflow 1-18
MATLAB Prototyping API Enhancements: Support complex data in AXI4 Stream Interface and input register readback in AXI4 Interface 1-18
Upgrade to Intel Quartus Pro 20.2 1-18
Inserted JTAG AXI Master at fixed frequency to avoid timing issue 1-18
Unsupported tool version in HDL workflow advisor 1-19
Multicycle path constraint packaging for IP core 1-19
HDL Coder Workflow Advisor: Option to expose DUT clock enable port and clock enable output port 1-19
Devicetree generation for IP cores 1-19
Updates to addAXI4StreamInterface function for fpga hardware connection object 1-20
Reset AXI4-Stream TLAST counter 1-20
HDL Coder Workflow Advisor: Improved code generation times 1-20
HDL Coder Workflow Advisor: Resource and timing report enhancement 1-20
Data type for Speedgoat PCIe Interface: Map bus data types to Speedgoat PCIe Interface 1-21
HDL Coder Support Package for Xilinx RFSoC Devices: Generate IP core and deploy reference designs on Xilinx RFSoC devices 1-21

Simscape Hardware-in-the-Loop Workflow 1-22

Support multiple solver times in Simscape models 1-22
Enable FPGA parameters in the protected model 1-22
RAM mapping for partition solver 1-22

R2021a

Model and Architecture Design 2-2

Half precision floating-point example for Field-Oriented Control algorithm 2-2

Comments tab in Global Settings pane and option to disable comments . . 2-2
HDL Code Advisor check for file extension based on target language 2-2
Hard Floating Point Support using Intel Quartus Pro 2-2

Block Enhancements **2-4**

Enhancement to parameterized HDL code generation for 1-D and 2-D mask values 2-4
HDL code generation for For Each Subsystem block with 1-D and 2-D partitioning of mask parameters 2-4
HDL code generation for For Each Subsystem block with matrix ports . . . 2-4
HDL code generation for Interval blocks and additional Detect blocks . . . 2-4
ShiftAdd architecture for Product block to avoid DSP consumption 2-4
HDL Coder library for fixed-point mathematical function blocks with latency 2-5
Count hit port for HDL Counter block to indicate when count value resets 2-5
3-D lookup table support 2-5
HDL Code Generation for Data Type Conversion block supports enumerated data types 2-5
Enhancement to HDL code generation for Sqrt block 2-6
New HDL-optimized Simulink blocks for reciprocal, divide, and modulo . . 2-6
Reduced HDL resource utilization in fixed-point matrix library blocks 2-6
Wireless HDL Toolbox Reference Applications: Implement 5G NR MIB recovery for FR2, OFDM interleaver and deinterleaver, and WLAN time and frequency synchronization 2-6
Wireless HDL Toolbox Blocks: Model OFDM Equalizer, NR CRC Encoder, and NR CRC Decoder 2-7
External Memory Modeling Examples: Model and deploy streaming video algorithms that require random access to memory (requires SoC Blockset product) 2-7
Multipixel-Multicomponent Video Streaming: Implement Pixel Stream Aligner, Pixel FIFO, and ROI Selector blocks for high-frame-rate color video 2-7
Functionality being removed or changed 2-8

Code Generation and Verification **2-9**

Improvement to HDL code generated for Stateflow Moore Chart blocks 2-9
Stateflow Chart property Initialize Outputs Every Time Chart Wakes Up cleared for HDL code generation 2-9
HDL block property GenericList for Subsystem blocks with BlackBox architecture 2-9
Single file for identical Simulink systems (Atomic and Virtual) 2-9

Speed and Area Optimizations **2-11**

Improved delay balancing support for multiple instances of atomic subsystems 2-11
Improved streaming in presence of scalar expanded constants 2-11
Enhancement to optimization that removes redundant logic for atomic subsystems and model references 2-11
Enhancement to sharing optimization for matrix data types 2-11
Adaptive pipelining optimization disabled on model by default 2-11

Generation of target-specific timing databases for critical path estimation . . . 2-12

IP Core Generation and Hardware Deployment . . . 2-13

Updates to supported software . . . 2-13
Data Type Support for AXI4 Slave: Map bus data types to AXI4 slave interfaces in IP Core generation . . . 2-13
HDL Workflow Advisor Enhancements . . . 2-13
FPGA Data Capture in HDL Workflow Advisor supports sequential trigger . . . 2-14
FPGA Data Capture integration with IP Core Generation workflow for generic Xilinx and generic Intel targets . . . 2-14
Multirate IP Core Generation: Support AXI4-Stream interface on slower-rate DUT ports . . . 2-14
Complex data type on AXI4-Stream data port . . . 2-15
High Bandwidth AXI Stream: Generate IP cores that have bit-widths greater than 128 bits on AXI4-Stream data ports . . . 2-15
Generation of HDL IP cores that have greater than 128 bits on external IO interfaces and external ports . . . 2-15
Interface option to customize initial value of AXI4 Master and AXI4 Stream registers . . . 2-15

Simscape Hardware-in-the-Loop Workflow . . . 2-17

Partitioning solver: Use partitioning solver to generate HDL code from nonlinear models . . . 2-17
Optimal value of oversampling factor automatically set on HDL implementation model . . . 2-17

R2020b

Model and Architecture Design . . . 3-2

Half-Precision Native Floating Point: Generate target-independent synthesizable RTL code from half-precision floating-point models . . . 3-2
HDL code generation for lookup tables that have floating-point types . . . 3-2
HDL Code Advisor check for blocks that introduce latency with fixed-point types . . . 3-2
Automatically package protected models with their dependencies . . . 3-2

Block Enhancements . . . 3-4

Optimized Square Root: Generate high-frequency fixed-point HDL implementation of square root operations . . . 3-4
Custom latency for math and trigonometric blocks with fixed-point types . . . 3-4
Modulo option for HDL Counter block . . . 3-4
HDL code generation for Scoped tag visibility for Goto block . . . 3-5
Product block enhancements for HDL code generation . . . 3-5
5G NR HDL MIB Recovery Reference Application: Implement 5G NR MIB recovery subsystem on FPGA or ASIC . . . 3-5

OFDM Transmitter and Receiver Reference Applications: Implement custom OFDM wireless communication system on FPGA or ASIC **3-5**
HDL-optimized FIR Decimation block and System object: Downsample signals using a FIR decimation filter with a hardware-friendly interface and architecture **3-5**
Gigasample-per-second (GSPS) CIC Decimation HDL-Optimized Block: Increase throughput of CIC decimation by using frame-based input . . . **3-6**
Corner Detector Block and System Object: Detect features using Harris algorithm **3-6**
Region of Interest (ROI) Resource Sharing: Share hardware resources and streaming control signals between vertically aligned regions **3-6**
Blob Analysis Example: Detect and label connected components in streaming video **3-6**
HDL Minimum Resource FFT and HDL Streaming FFT blocks have been removed **3-6**

Code Generation and Verification **3-8**

Option to scalarize vector ports only at DUT level in VHDL code **3-8**
HDL code generation for models that have comment through blocks **3-8**
HDL code generation for models that have Subsystem Reference blocks **3-8**
Enhancement to HDL code generation for nontop DUT **3-8**
HDL code generation for nonboolean inputs at control ports **3-9**
HDL code generation for absolute time temporal logic in Stateflow **3-9**
Default HDL simulation command vsim -novopt has changed to vsim -voptargs=+acc **3-9**
UseMatrixTypesInHDL property not recommended **3-9**

Speed and Area Optimizations **3-10**

Option to control removal of unused ports in generated HDL code **3-10**
Hierarchy flattening report **3-10**
Optimization enhancements for Sum of Elements and MinMax blocks . . . **3-10**

IP Core Generation and Hardware Deployment **3-11**

Rapid prototyping of HDL IP core by using software interface script **3-11**
Interface option to customize initial value of AXI4 slave registers **3-11**
Generation of HDL IP cores that have greater than 128 bits on internal IO interface **3-12**
IP core generation workflow for scalarization of vector ports only at DUT level in VHDL code **3-12**
Intel Quartus Pro SoC Targeting: Generate generic HDL IP core or integrate IP core into Intel reference designs **3-12**
Arria 10 SoC AXI4 Slave reference design **3-13**
Speedgoat I/O Modules IO331 and IO333 being removed **3-13**
Audio filter reference application for Intel SoC device **3-13**
Updates to supported software **3-13**

Simscape Hardware-in-the-Loop Workflow **3-14**

Automatic replacement of Simscape subsystem with state-space implementation **3-14**
Automatic setting of number of solver iterations in Simscape HDL Workflow Advisor **3-14**

Mapping of state-space parameters to RAM in HDL implementation model . . . 3-14
Duplicate configurations removed in generated HDL implementation model . . . 3-14

R2020a

Model and Architecture Design . . . **4-2**

Additional HDL modeling guidelines added to documentation . . . **4-2**
Functionality being removed or changed . . . **4-2**

Block Enhancements . . . **4-3**

Inverse of streaming matrix input using Gauss-Jordan elimination method . . . **4-3**
Improvement to readability of bus element port names in HDL code . . . **4-3**
New Fixed-Point Designer Simulink block library . . . **4-4**
LTE HDL Toolbox name change to Wireless HDL Toolbox . . . **4-4**
5G NR Signal Synchronization Reference Application: Use primary and secondary synchronization signals (PSS and SSS) to detect connection to valid cell . . . **4-5**
5G NR Polar Encoder and Decoder, 5G NR LDPC Encoder and Decoder blocks . . . **4-5**
OFDM Modulator, OFDM Channel Estimator, and RS Decoder blocks . . . **4-5**
Variable CIC Decimation Factor: Specify decimation factor as an input to the CIC Decimation HDL Optimized block . . . **4-6**
Gigasample-per-second (GSPS) NCO: Generate frame-based output from HDL-optimized NCO for high speed applications (requires HDL Coder for code generation) . . . **4-6**
Corner Detector Block and System Object: Detect features using FAST algorithm . . . **4-6**
Line Buffer with No Padding: Specify option to not add padding for blocks that use line buffer memory . . . **4-6**

Code Generation and Verification . . . **4-8**

Obfuscated HDL Output: Generate plain-text HDL code with randomized identifier names . . . **4-8**
Improvements to HDL code generated for Stateflow charts . . . **4-8**

Speed and Area Optimizations . . . **4-11**

Upsampling signals without latency using Rate Transition blocks . . . **4-11**

IP Core Generation and Hardware Deployment . . . **4-12**

AXI4-Stream for MIMO: Generate IP cores with multiple input and output channels . . . **4-12**
High-Bandwidth AXI Master: Generate IP cores with up to 512 bits on AXI4 Master data ports . . . **4-12**
Performance improvement to AXI4 Master write operations . . . **4-12**

Dynamic customization of reference design based on reference design parameters . . . 4-13
Option to insert JTAG MATLAB AXI Master in standalone FPGA reference designs (requires HDL Verifier) . . . 4-14
socExportReferenceDesign Function: Automatically create reference design (requires SoC Blockset) . . . 4-14
Intel Quartus Pro Targeting: Synthesize and implement generated HDL code on Intel FPGAs by using HDL Workflow Advisor . . . 4-14
Speedgoat IO Modules IO331 and IO331-6 being removed . . . 4-15
Updates to supported software . . . 4-15

Simscape Hardware-in-the-Loop Workflow . . . 4-17

Simscape Hardware-in-the-Loop: Generate HDL implementation model from multiple Simscape networks . . . 4-17
Reduction in latency of HDL implementation model generated from Simscape algorithm . . . 4-17
Improvement to single-rate resource sharing in HDL implementation model . . . 4-17

R2019b

Model and Architecture Design . . . 5-2

HDL code generation for MATLAB Function block in native floating-point mode . . . 5-2
HDL Coder contextual tab on Simulink Toolstrip . . . 5-2
Documentation revision for HDL code generation support for blocks . . . 5-2

Block Enhancements . . . 5-4

Discrete FIR Filter HDL Optimized block supports complex coefficient values . . . 5-4
Process high-frame-rate or high-resolution video with multipixel streaming interface . . . 5-4
OFDM Demodulator, Convolutional Encoder, and Puncturer blocks for custom wireless communication protocols . . . 5-4
Symbol Demodulator and 1536-point FFT for LTE and NR (5G) designs . . . 5-4
HDL-optimized CIC Decimation block and System Object . . . 5-5
Enhancements to fixed-point Division and Reciprocal operators . . . 5-5
FWFT mode for HDL FIFO block . . . 5-5
HDL code generation enhancements to matrix support . . . 5-5
Block-level option to control HDL code generated for Multiport Switch block . . . 5-6
HDL code generation for partitioning of mask parameters in For Each Subsystem . . . 5-6
HDL code generation for Fcn block . . . 5-6

Code Generation and Verification . . . 5-7

UltraRAM mapping in Xilinx devices . . . 5-7

Speed and Area Optimizations 5-8
Enhanced optimization support for MATLAB Function block 5-8
HDL optimizations across MATLAB Function blocks and other Simulink blocks 5-8
Flattening of subsystems in presence of optimizations 5-8

IP Core Generation and Hardware Deployment 5-9
Optimization of AXI4 slave readback logic 5-9
Customization of AXI4 Slave ID width in Generic IP Core Generation workflow 5-9
Option to insert JTAG MATLAB AXI Master in SoC reference designs (requires HDL Verifier) 5-9
Performance improvement to AXI Master interfaces in HDL DUT IP core 5-10
Updates to supported software 5-10

Simscape Hardware-in-the-Loop Workflow 5-11
Enhanced HDL implementation model for Simscape and Simulink plant in feedback loop 5-11
Number display of differential and algebraic variables in Simscape HDL Workflow Advisor 5-11
Separation of Get state-space parameters task for extracting and discretizing equations 5-11
Generation of implementation model with coefficients as single type and computation of results in double type 5-12

R2019a

Model and Architecture Design 6-2
Protected Model Code Generation: Share protected Simulink models with the option to allow HDL code generation 6-2
Enhancements to single-precision native floating-point operators support 6-2
Additional block support with double-precision native floating-point code generation 6-3
Additional Verilog constructs supported with HDL import 6-3
HDL Coder contextual tab in Simulink Toolstrip 6-3
HDL Coder Modeling Guidelines in Documentation 6-4

Block Enhancements 6-5
Streaming Matrix Multiply and Streaming Matrix Inverse Reference Applications 6-5
Partition Offset parameter support in For Each Subsystem block 6-5
Enhancements to Assignment and Selector blocks 6-5
Enhancements to Discrete FIR Filter HDL Optimized block and frame-based Discrete FIR Filter block 6-6

LTE Reference Applications: Transmitter example and TDD support for SIB recovery 6-6
OFDM Modulator block and LTE and 5G Symbol Modulator blocks 6-6
Increased kernel size limits for Image Filter block 6-7

Code Generation and Verification 6-8

Customization of constant name in VHDL code generated for Lookup Table data 6-8
Optimized counters in generated HDL code for Stateflow temporal logic 6-8
HDL Coder Workflow: Enhanced options for model generation 6-9
HDL Code Generation: Diagnostics tab renamed to Advanced 6-9

Speed and Area Optimizations 6-12

Improvements to element-wise matrix transformation 6-12
Optimization of unconnected port for removing redundant logic in design 6-12

IP Core Generation and Hardware Deployment 6-13

DUT AXI4 slave interface connection to multiple AXI Master interfaces in reference designs 6-13
Default system with External DDR4 Memory Access reference design . . . 6-13
Generation of HDL IP core without AXI4 slave interfaces 6-13
Improved synchronization of global reset signal to IP core clock domain 6-14
Minimization of clock enable signals in IP Core Generation workflow . . . 6-14
Updates to supported software 6-14

Simscape Hardware-in-the-Loop Workflow 6-15

Double-precision floating-point support for HDL code generation from Simscape models 6-15
Validation logic verification for functional equivalence of HDL implementation model with Simscape model 6-15
Simscape to HDL Workflow Reference Applications 6-15

R2018b

Model and Architecture Design 7-2

Verilog Import: Import synthesizable Verilog code and generate Simulink model 7-2
Double-Precision Native Floating Point: Generate target-independent synthesizable RTL from double-precision floating-point models 7-2
Custom latency specification for native floating-point operators 7-2
Enhancements to supported blocks and complex data types with single-precision native floating-point 7-3
Enhancements to output delay absorption for complex multipliers with single-precision native floating-point 7-3

Block Enhancements **7-5**

Enhancements to matrix support for HDL code generation **7-5**
HDL code generation support for Probe block and blocks that detect change in input signal value **7-5**
HDL code generation support for Foreach Subsystem with Minimize global resets setting **7-5**
HDL Coder support for virtual bus containing nonvirtual subbus **7-5**
Viterbi Decoder and Depuncturer Block: Decode bitstreams by using the Viterbi algorithm with puncturing, terminated, and truncated modes (requires LTE HDL Toolbox) **7-6**
HDL code generation support for complex input signals or complex coefficients of frame-based Discrete FIR Filter and FIR Decimation blocks (requires DSP System Toolbox) **7-6**
Discrete FIR Filter HDL Optimized: Select transposed architecture, optimize symmetric and antisymmetric coefficients, and enable reset port (requires DSP System Toolbox) **7-6**

Code Generation and Verification **7-8**

Test Point Integration with FPGA Data Capture: Use FPGA data capture to specify signals to be captured during FPGA testing by using Test Points in Simulink **7-8**
User-Interface Improvements to HDL Workflow Advisor and HDL Code Generation Pane in Configuration Parameters Dialog Box **7-8**

Speed and Area Optimizations **7-11**

Enhancements to optimization that removes redundant logic in design . . **7-11**
Streaming operation modes of Multiply-Accumulate block **7-11**
Different output latencies for designs with clock-rate pipelining enabled at output ports **7-11**

IP Core Generation and Hardware Deployment **7-14**

Xilinx Zynq UltraScale+ MPSoC Targeting: Select from predefined targets and reference designs to generate code for MPSoC devices **7-14**
Multirate IP Core Generation: Target AXI4-Stream and AXI4 Master interfaces for designs with multiple sample rates **7-14**
PCIe MATLAB as AXI Master with External DDR4 Memory Access reference design for Intel Arria10 GX FPGA Development kit **7-14**
Timing failure check in Build FPGA Bistream step of IP Core Generation workflow **7-15**
Support for read back of AXI4 write registers in IP Core Generation workflow **7-15**
Microsemi Libero SoC Targeting: Synthesize and implement generated code on Microsemi FPGAs by using HDL Workflow Advisor **7-16**
Speedgoat IO Modules IO321 and IO321-5 being replaced **7-17**
Updates to supported software **7-17**

Simscape Hardware-in-the-Loop Workflow **7-18**

Hardware Acceleration of Plant Models: Generate HDL code from Simscape Electrical switched linear models **7-18**

Model and Architecture Design **8-2**

HDL Model Checker integrated with Model Advisor **8-2**
Updates to model checks in HDL Coder **8-2**
Enhanced Radix-4 algorithm for Divide and Reciprocal blocks in Native Floating Point mode **8-3**
Improved shift-and-add algorithm for exponential and hyperbolic functions in Native Floating Point mode **8-3**
HDL code generation support for all rounding modes of Data Type Conversion block in Native Floating Point mode **8-3**
Floating-point control for Multiport Switch and Selector blocks **8-4**

Block Enhancements **8-5**

Matrix Support: Generate HDL code directly from two-dimensional matrix data types and operations **8-5**
Additional blocks and block modes supported for HDL code generation . . **8-5**
Bit-Natural FFT Output: Directly access the bit-natural output from the frame-based FFT/IFFT (Requires DSP System Toolbox) **8-5**
LTE OFDM demodulation and Gold sequence generation blocks (Requires LTE HDL Toolbox) **8-6**
Additional pipelining of HDL-optimized Complex to Magnitude-Angle (Requires DSP System Toolbox) **8-6**
5G filtered-OFDM modulation reference application (Requires LTE HDL Toolbox) **8-6**

Code Generation and Verification **8-7**

Line-Level Traceability: Navigate directly between Simulink blocks and corresponding lines of generated HDL code **8-7**
Microsemi FPGA Support: Specify Microsemi Libero SoC as Synthesis Tool and generate HDL code **8-7**
Concise summary of synthesis results displayed in HDL Workflow Advisor **8-7**
New Code Generation Report: View more information and navigate through code generation results more easily **8-8**

Speed and Area Optimizations **8-11**

Critical Path Estimation with Native Floating Point: Report critical path for designs with single-precision floating-point operations **8-11**
Simplification of constant operations and other optimizations for fixed-point and floating-point arithmetic operations **8-11**
Improvement to reduction of matching delays in clock-rate pipelining regions across hierarchical boundaries **8-11**
MaxOversampling and MaxComputationLatency parameters being removed **8-12**

IP Core Generation and Hardware Deployment **8-13**

AXI4-Stream for Intel FPGA: Generate IP cores with the AXI4-Stream interface targeting Intel FPGAs **8-13**

Intel SoC Reference Design: Target the Intel Arria 10 SoC Development Kit with DDR4 external memory access **8-13**
Simulink test point port mapping in IP Core Generation and Simulink Real-Time FPGA I/O workflows **8-13**
Audio Reference Design Example on ZYBO Board: Create custom reference design to run audio algorithm on ZYBO board **8-14**
IP Core Generation of I2C Master Controller Example: Generate IP core for Stateflow-Based I2C Master Controller to configure Audio Codec chip **8-14**
Ethernet programming method being removed **8-14**
Updates to supported software **8-14**

R2017b

Model and Architecture Design **9-2**

Model Advisor Checks: Check and update your Simulink model for HDL code generation compatibility **9-2**
Simulink Test Points in HDL: Debug internal signals by automatically routing the signals to top-level HDL ports **9-2**
Floating-point Support for Simulink Real-Time FPGA I/O: Generate single-precision floating point HDL for communication over the Simulink Real-Time PCIe Interface **9-2**
Additional single-precision floating-point operators and block support . . . **9-3**
Improvements to native floating-point operators and algorithms **9-3**
Input Range Reduction setting for Trigonometric Function blocks in native floating-point mode **9-3**
Block-level latency customization for Discrete Transfer Function and Discrete Time Integrator blocks with native floating-point **9-4**

Block Enhancements **9-5**

Minimum Resource FFT/IFFT: Reduce resource usage with the Burst Radix 2 architecture of the HDL-Optimized FFT (requires DSP System Toolbox) **9-5**
Support for scalar addressing mode with vector data input to hdl.RAM System Object **9-5**
New HDL RAMs Block Library and hdl.RAM System Object based blocks **9-5**
Synchronous versions of Unit Delay blocks with reset and enable ports in Discrete block library **9-6**
Bilateral filter, bird's-eye-view transform, and line buffer for vision applications **9-7**
HDL code generation support for Bus Element port blocks **9-7**
One-hot and two-hot encoding schemes for enumeration types **9-7**
Custom header and footer comments in generated HDL code **9-8**

Code Generation and Verification **9-9**

Changes to HDL Code Generation Panel in Configuration Parameters Dialog Box **9-9**

Speed and Area Optimizations **9-10**

Vector Input Multiply-Accumulate (MAC) Block: Map arithmetic operations efficiently to FPGA DSP slices **9-10**
Hierarchical Clock Rate Pipelining: Apply clock rate pipelining across hierarchical boundaries **9-10**
Support for enable-based multicycle path constraints **9-10**
Clock-rate pipelining enhancements **9-11**

IP Core Generation and Hardware Deployment **9-12**

AXI4 Master Interface: Facilitate communication between your design and external memory by using the AXI4 Master protocol for more flexible data access **9-12**
IP Core Generation Support for Xilinx System Generator: Generate an HDL IP core for DUT containing System Generator blocks **9-12**
INOUT port type support for External Port interface in IP Core Generation workflow **9-12**
Faster Simulink Real-Time FPGA I/O model build time with version register in generated IP core **9-12**
Default system with External DDR3 Memory Access reference design . . . **9-13**
Updates to supported software **9-13**
HDL Coder support packages renamed **9-13**

R2017a

Model and Architecture Design **10-2**

HDL Floating Point Operations Library: Easily find additional and existing single-precision floating-point blocks supported for HDL code generation **10-2**
Floating-Point Latency Customization at Block-Level **10-2**
Additional Block and System Object Support with Native Floating Point **10-3**
Custom reference model prefix specification **10-3**
GenerateWebview parameter name changed to HDLGenerateWebview **10-4**
Comments in HDL code for Simulink blocks with text annotations **10-4**

Block Enhancements **10-7**

For Each Subsystems: Reduce block replication and improve code reuse in HDL-targeted designs **10-7**
HDL Optimized Filters: Model and generate optimized hardware implementations for FIR filters (requires DSP System Toolbox) **10-7**
HDL Channelizer Block and System Object: Isolate narrowband channels from a wideband signal and generate HDL with efficient multiplier usage (requires DSP System Toolbox) **10-7**
Gigasample per Second (GSPS) Signal Processing: Increase throughput of FIR decimation algorithms by using frame input **10-7**
Enhancements to MATLAB Function block support in synchronous subsystems **10-8**

HDL Coder support for blocks that support bus signal treated as vector **10-8**
HDL code generation support for Bus Assignment block with nonvirtual bus **10-9**
Additional HDL Coder bus support **10-9**
HDL code generation support for System Objects with enumeration types **10-9**

Code Generation and Verification **10-10**

Native Floating-Point Testbench: Generate SystemVerilog DPI, cosimulation, and FPGA-in-the-loop test benches with single-precision data types (requires HDL Verifier) **10-10**
More fixed-size variable information in Fixed-Point Conversion step of HDL Coder App **10-10**
Comments in generated HDL code for MATLAB System blocks **10-10**
Global reset signals minimization in generated HDL Code **10-10**
HDL code generation support for DUT subsystem with custom HDL properties **10-11**
Changes in HDL Code Generation Panel in Configuration Parameters Dialog Box **10-11**
Syntax Highlighting of Generated HDL Code in HTML Report **10-11**

Speed and Area Optimizations **10-12**

Improvements to delay balancing in multirate regions **10-12**
Functionality Being Removed or Changed **10-12**

IP Core Generation and Hardware Deployment **10-13**

Data Type Support for AXI4 Slave: Map floating-point signals and vector signals to AXI4 slave interfaces in IP core generation **10-13**
Incremental Vivado Synthesis: Enable IP caching for faster synthesis of Xilinx Vivado reference designs **10-13**
IP core generation support for Altera Megafunction **10-14**
Custom IP repository specification **10-14**
Xilinx Virtex-2 FPGA board support being removed **10-14**
Updates to supported software **10-14**

R2016b

Model and Architecture Design **11-2**

Native Floating Point: Generate target-independent synthesizable RTL from single-precision floating-point models **11-2**
HDL Coder support for tunable parameters in data dictionary **11-2**
Generic ports for DUT mask parameters **11-2**
Simulink diagnostic suppressor option **11-2**

Block Enhancements **11-5**

Gigasample Per Second (GSPS) Signal Processing: Increase throughput of HDL code generated from Discrete FIR Filter and Integer Delay blocks by using frame input **11-5**
Bit-reversed input order for HDL-optimized FFT **11-5**
High-throughput polyphase filter bank for HDL example **11-5**
HDL support for reset port on Discrete FIR Filter **11-5**
HDL Coder support for array of buses **11-5**
Synchronous behavior for Resettable Subsystem with State Control block **11-6**
HDL optimized Sine and Cosine blocks **11-6**
Simpler method to call System objects **11-6**

Code Generation and Verification **11-7**

Logic Analyzer: Visualize, measure, and analyze transitions and states over time for Simulink signals **11-7**
HDL Coder support for creating and attaching configuration sets **11-7**
VHDL Architecture Name available in Configuration Parameters dialog box **11-7**
RAM with generic ports enhancement **11-7**
Stateflow comments generated as comments in HDL **11-7**
Tolerance check for floating-point libraries **11-8**
Code Generation Report enhancements **11-9**
Comprehensive documentation for HDL coding standard rules **11-9**
More discoverable logs and reports for fixed-point conversion in HDL Coder app **11-9**
Enhancements in generated model for Lookup Tables **11-10**
Target and Optimizations pane in HDL Coder Configuration Parameters **11-10**
Link to Code Generation Report after HDL code generation **11-11**

Speed and Area Optimizations **11-12**

Adaptive Pipelining: Specify synthesis tool and target clock frequency for automatic pipeline insertion and balancing **11-12**
Clock-rate pipelining enhancements **11-12**
Resource sharing enhancements **11-12**
Delay balancing failures reported as errors **11-13**
Optimization of Delay blocks with nonzero initial condition **11-13**
Initialization script specification for Delay blocks without reset **11-13**

IP Core Generation and Hardware Deployment **11-15**

AXI4-Stream Video Interface: Generate HDL code with the AXI4-Stream Video interface by using the IP core generation workflow **11-15**
Customizable FPGA floating-point target configuration **11-15**
Additional block support for FPGA floating-point target library mapping **11-15**
Default video system reference design **11-15**
Custom reference design enhancements **11-16**
IP Core Generation workflow for Xilinx and Altera FPGA devices **11-16**
Additional FPGA board support for IP Core Generation workflow **11-17**
Target clock frequency specification **11-17**
Simulink Real-Time FPGA I/O workflow support for Xilinx Vivado **11-17**
Speedgoat IO333–325K target hardware support **11-17**
Updates to supported software **11-17**

R2016a

Model and Architecture Design **12-2**

Gigasample per Second (GSPS) Signal Processing: Increase throughput of HDL-optimized FFT and IFFT algorithms using frame input **12-2**
Tunable and nontunable parameter enhancements **12-2**
Reusable HDL code enhancements for subsystems with tunable mask parameters **12-2**
HDL Coder support for nondirect feedthrough setting in MATLAB Function blocks **12-3**

Block Enhancements **12-4**

Synchronous Subsystem Toggle: Specify enable and reset behavior for cleaner HDL code by using State Control block **12-4**
Region-of-interest selection and grayscale morphology **12-5**
Nested bus support enhancements **12-5**
Block support enhancements **12-5**

Code Generation and Verification **12-6**

Faster Test Bench Generation and HDL Simulation: Generate SystemVerilog DPI test benches for large data sets with HDL Verifier **12-6**
Code Generation Report enhancements **12-6**
Changes to Fixed-Point Conversion Code Coverage **12-6**
Progress indicator for HDL test bench generation **12-7**
Test bench generation with updated model stop time **12-7**
Performance improvement for MATLAB to HDL test bench generation . . **12-8**
Coding standard check for length of control flow statements in a process block **12-8**
Warnings for non-ASCII characters in generated HDL code **12-8**
Japanese translation for resource report **12-8**

Speed and Area Optimizations **12-9**

Resource Sharing Enhancements: Share multipliers and gain operations that have different data types **12-9**
Biquad Filter block participates in subsystem HDL optimizations **12-9**
More functions for Multiply-Add block to map to DSP **12-9**
Generation of Multiply-Add blocks for complex multiply operations **12-9**
RAM mapping for pipeline and floating-point delays **12-9**
Initialization script generated for Delay blocks without reset for ModelSim simulation **12-10**

IP Core Generation and Hardware Deployment **12-11**

Hard Floating-Point IP Targeting: Generate HDL to map to Altera Arria 10 floating-point units at user-specified target frequency **12-11**
End-to-end scripting for Simulink Real-Time FPGA I/O workflow **12-11**
SoC device programmed by using Ethernet connection **12-11**
Custom programming method for IP Core Generation workflow **12-11**
Interface connection name and type for custom reference designs **12-11**
Updates to supported software **12-12**

Automatic generation of FPGA top-level wrapper based on workflow . . 12-12

R2015aSP1

Bug Fixes

R2015b

Model and Architecture Design **14-2**

Model Arguments: Parameterize instances of model reference blocks . . . **14-2**
Integration with Xilinx Vivado System Generator for DSP blocks **14-2**
struct input and output for top-level MATLAB design function **14-2**
Tunable parameters in MATLAB Function block **14-2**
Output initialization requirement for Stateflow Moore Charts **14-2**
Enforce ASCII character requirement for model property values **14-3**

Block Enhancements **14-4**

Expanded Bus Support: Generate HDL for enabled or triggered subsystems with bus inputs and for black boxes with bus I/O **14-4**
Library Browser view shows blocks supported for HDL code generation **14-4**
Trigonometric Function block with sin or cos function can have vector inputs **14-4**
Discrete FIR Filter supports HDL optimizations **14-4**
HDL-optimized FIR Rate Conversion block and System object **14-5**

Code Generation and Verification **14-6**

HDL Coder Configuration Parameters in list view **14-6**
Support for configuration parameter Default parameter behavior **14-6**
Test bench performance improvements with file I/O **14-6**
Image processing examples **14-7**

Speed and Area Optimizations **14-8**

Quality of Results Improvement: Stream and share resources more broadly and efficiently **14-8**
Multiply-Add block **14-8**
Hierarchy flattening for masked subsystems and user library blocks **14-8**
Loop optimization improvement **14-8**
Complex Gain speed optimization **14-8**
Redesigned serializer for streaming and resource sharing **14-9**
Tapped Delay optimization **14-9**

IP Core Generation and Hardware Deployment **14-10**

Tunable Parameters: Map to AXI4 interfaces to enable hardware run-time tuning by the embedded software on the ARM processor **14-10**
End-to-end scripting from design through IP core generation, FPGA Turnkey, and generic ASIC/FPGA workflows **14-10**
Synthesis objective for synthesis tool target optimization **14-10**
AXI4-Stream vector interface **14-10**
Connect IP core with other IP blocks in custom reference designs **14-11**
Kintex UltraScale and Virtex UltraScale device family support in generic ASIC/FPGA and IP core generation workflows **14-11**

R2015a

Model and Architecture Design **15-2**

Localized control using pragmas for pipelining, loop streaming, and loop unrolling in MATLAB code **15-2**
Model templates for HDL code generation **15-2**
Tunable parameter data type and model reference support enhancements **15-3**
Include custom or legacy code using DocBlock **15-4**
Single library for VHDL code generated from model references **15-4**
Timing controller architecture and postfix options in Configuration Parameters dialog box and HDL Workflow Advisor **15-4**
Functionality Being Removed or Changed **15-4**

Block Enhancements **15-6**

Enumeration support at DUT ports **15-6**
Map to multiple RAM banks **15-6**
Code generation for bus output from Bus Selector and Constant blocks **15-6**
Initial condition for Deserializer1D **15-6**
Block support enhancements **15-6**
Code generation for predefined System objects in MATLAB System block **15-6**
Specify filter coefficients using a System object **15-7**
Libraries for HDL-supported DSP System Toolbox and Communications Toolbox blocks **15-7**
Support for image processing, video, and computer vision designs in new Vision HDL Toolbox product **15-7**
Support for ‘inherit via internal rule’ data type setting on FIR Decimation and Interpolation blocks **15-7**

Code Generation and Verification **15-8**

Coding standard check for X and Z constants **15-8**
Coding style improvements **15-8**
Example HDL implementation of LTE OFDM modulator and detector with LTE Toolbox **15-8**

Speed and Area Optimizations **15-9**

Critical path estimation without running synthesis 15-9
Clock-rate pipelining enhancements 15-9
Partitioning for large multipliers to improve clock frequency and DSP reuse on the FPGA 15-9
Highlighting for blocks in the model that prevent retiming 15-9
Resource sharing for adders and more control over shareable resources 15-10
Speed and area optimizations for designs that use Unit Delay Enabled, Unit Delay Resettable, and Unit Delay Enabled Resettable 15-10
Resource sharing for multipliers and adders with input data types in different order 15-10
Vector streaming for MATLAB code 15-10

IP Core Generation and Hardware Deployment **15-12**

Mac OS X platform support 15-12
AXI4-Stream interface generation for Xilinx Zynq IP core 15-12
Custom reference design and custom SoC board support 15-12
Automatic iterative optimization for IP core generation and FPGA Turnkey workflows 15-12
Speedgoat IO331-6 digital I/O interface target 15-12
IP core settings saved with model 15-12
Updates to supported software 15-13

R2014b

Model and Architecture Design **16-2**

Custom or legacy HDL code integration in the MATLAB to HDL workflow 16-2
Model reference as DUT for code generation 16-2
Tunable parameter support for Gain and Constant blocks 16-2
Code generation for Stateflow active state output 16-2
Clock enable minimization for code generated from MATLAB designs 16-2
HDL Block Properties dialog box shows only valid architectures 16-2
2-D matrix types in HDL generated for MATLAB matrices 16-2

Block Enhancements **16-4**

Code generation for HDL optimized FFT/IFFT System object and HDL optimized Complex to Magnitude-Angle System object and block 16-4
Added features to HDL optimized FFT/IFFT blocks, including reduced latency 16-4
HDL Reciprocal block with Newton-Raphson Implementation 16-4
Serializer1D and Deserializer1D blocks 16-5
Additional blocks supported for code generation 16-5
Composite user-defined System object support 16-5
System object output and update method support 16-5
hdlram renamed to hdl.RAM 16-5
Functionality Being Removed or Changed 16-6

Code Generation and Verification **16-7**

Coding standards customization **16-7**
HDL Designer script generation **16-7**
Traceable names for RAM blocks and port signals **16-7**
for-generate statements in generated VHDL code **16-7**
Validation model generation regardless of delay balancing results **16-7**

Speed and Area Optimizations **16-8**

Clock-rate pipelining to optimize timing in multi-cycle paths **16-8**
RAM mapping for user-defined System object private properties **16-8**
Highlighting for feedback loops that inhibit optimizations **16-8**
Optimizations available for conditional-execution subsystems **16-8**
Variable pipelining in conditional MATLAB code **16-9**
Optimizations available with UseMatrixTypesInHDL for MATLAB Function block **16-9**

IP Core Generation and Hardware Deployment **16-10**

Support for Xilinx Vivado **16-10**
IP core generation for Altera SoC platform **16-10**
Custom HDL code for IP core generation from MATLAB **16-10**
Target platform interface mapping information saved with model **16-10**
Documentation installation with hardware support package **16-11**

R2014a

Model and Architecture Design **17-2**

HDL block library in Simulink **17-2**
Persistent keyword not needed in HDL code generation **17-2**
Negative edge clocking **17-2**
Bidirectional port specification **17-3**
Port names in generated code match signal names **17-3**
ModelReference default architecture for Model block **17-3**
Reset for timing controller **17-3**
Reset port optimization **17-4**
Functionality Being Removed or Changed **17-4**

Block Enhancements **17-6**

Code generation for enumeration data types **17-6**
Code generation for FFT HDL Optimized and IFFT HDL Optimized blocks **17-6**
Bus support improvements **17-6**
Variant Subsystem support for configurable models **17-6**
Trigger signal can clock triggered subsystems **17-6**
2-D matrix types in code generated for MATLAB Function block **17-6**
64-bit data support **17-7**
HDL code generation from MATLAB System block **17-7**
System object methods in conditional code **17-7**
Dual Rate Dual Port RAM block **17-7**

Additional blocks and block implementations supported for code generation . . . 17-8

Code Generation and Verification . . . 17-9

Errors instead of warnings for blocks not supported for code generation . . . 17-9
Ascent Lint script generation . . . 17-9
Incremental code generation and synthesis . . . 17-9
Automatic C compiler setup . . . 17-9

Speed and Area Optimizations . . . 17-10

RAM mapping scheduler improvements . . . 17-10
Performance-prioritized retiming . . . 17-10
Retiming without moving user-created design delays . . . 17-10
Resource sharing factor can be greater than number of shareable resources . . . 17-10
Reduced area with multirate delay balancing . . . 17-11
Serializer-deserializer and multiplexer-demultiplexer optimization . . . 17-11

IP Core Generation and Hardware Deployment . . . 17-12

ZC706 target for IP core generation and integration into Xilinx EDK project . . . 17-12
Automatic iterative clock frequency optimization . . . 17-12
Synthesis attributes for multipliers . . . 17-12
Custom HDL code for IP core generation . . . 17-12
Synthesis and simulation tool addition and detection after opening HDL Workflow Advisor . . . 17-12
xPC Target is Simulink Real-Time . . . 17-13
Updates to supported software . . . 17-13

R2013b

Model and Architecture Design . . . 18-2

Model reference support and incremental code generation . . . 18-2
Code generation for subsystems containing Altera DSP Builder blocks . . . 18-2
Module or entity generation for local functions in MATLAB Function block . . . 18-2
Reset port optimization . . . 18-2
Load constants from MAT-files . . . 18-2

Block Enhancements . . . 18-4

Code generation for user-defined System objects . . . 18-4
Bus signal inputs and outputs for MATLAB Function block and Stateflow charts . . . 18-4
HDL Counter has specifiable start value . . . 18-4
Maximum 32-bit address for RAM . . . 18-4

Removing HDL Support for NCO Block 18-4

Code Generation and Verification **18-5**

Coding style improvements according to industry standard guidelines . . 18-5
Coding standard report target language enhancement and text file format 18-5
UI for SpyGlass, Leda, and custom lint tool script generation 18-5
File I/O to read test bench data in VHDL and Verilog 18-5
Floating point for FIL and HDL cosimulation test bench generation 18-6
Fixed-point file name change 18-6

Speed and Area Optimizations **18-7**

RAM inference in conditional MATLAB code 18-7
Coding style for improved ROM mapping 18-7
Pipeline registers between adder or multiplier and rounding or saturation logic 18-7
Distributed pipelining improvements with loop unrolling in MATLAB Function block 18-7

IP Core Generation and Hardware Deployment **18-8**

IP core integration into Xilinx EDK project for ZC702 and ZedBoard 18-8
FPGA Turnkey and IP Core generation in MATLAB to HDL workflow 18-8
Synthesis tool addition and detection after MATLAB-to-HDL project creation 18-8
Synthesis script generation for Microsemi Libero and other synthesis tools 18-8
Floating-point library mapping for mixed floating-point and fixed-point designs 18-9
xPC Target FPGA I/O workflow separate from FPGA Turnkey workflow 18-9
AXM-A75 AD/DA module for Speedgoat IO331 FPGA board 18-9
Speedgoat IO321 and IO321-5 target hardware support 18-9
Support package for Xilinx Zynq-7000 platform 18-9
Support package for Altera FPGA boards 18-10
Support package for Xilinx FPGA boards 18-10
Additional FPGA board support for FIL verification, including Xilinx KC705 and Altera DSP Development Kit, Stratix V edition 18-11

R2013a

Model and Architecture Design **19-2**

Code generation for System objects in a MATLAB Function block 19-2
Output folder structure includes model name 19-2
Prefix for module or entity name 19-2
Functionality being removed 19-2

Block Enhancements **19-3**

Single rate Newton-Raphson architecture for Sqrt, Reciprocal Sqrt **19-3**
Additional System objects supported for code generation **19-3**
Additional blocks supported for code generation **19-3**

Code Generation and Verification **19-4**

Static range analysis for floating-point to fixed-point conversion **19-4**
Cosimulation and FPGA-in-the-Loop for MATLAB HDL code generation **19-4**
HDL coding standard report and lint tool script generation **19-4**
File I/O to read test bench data in Verilog **19-5**

Speed and Area Optimizations **19-6**

User-specified pipeline insertion for MATLAB variables **19-6**
Resource sharing and streaming without over clocking **19-6**
Resource sharing for the MATLAB Function block **19-6**
Finer control for delay balancing **19-6**
Complex multiplication optimizations in the Product block **19-6**

IP Core Generation and Hardware Deployment **19-7**

Generation of custom IP core with AXI4 interface **19-7**
Coprocessor synchronization in FPGA Turnkey and IP Core Generation workflows **19-7**
Speedgoat IO331 Spartan-6 FPGA board for FPGA Turnkey workflow ... **19-7**

R2012b

Input parameter constants and structures in floating-point to fixed-point conversion **20-2**

RAM, biquad filter, and demodulator System objects **20-2**
HDL RAM System object **20-2**
HDL System object support for biquad filters **20-2**
HDL support with demodulator System objects **20-2**

Generation of MATLAB Function block in the MATLAB to HDL workflow **20-2**

HDL code generation for Reed Solomon encoder and decoder, CRC detector, and multichannel Discrete FIR filter **20-2**
HDL code generation **20-2**
Multichannel Discrete FIR filters **20-3**

Targeting of custom FPGA boards **20-3**

Optimizations for MATLAB Function blocks and black boxes **20-3**

Generate Xilinx System Generator Black Box block from MATLAB **20-3**

Save and restore HDL-related model parameters **20-3**

Command-line interface for MATLAB-to-HDL code generation **20-3**

User-specifiable clock enable toggle rate in test bench **20-4**

RAM mapping for dsp.Delay System object **20-4**

Code generation for Repeat block with multiple clocks **20-4**

Automatic verification with cosimulation using HDL Coder **20-4**

ML605 Board Added To Turnkey Workflow **20-4**

R2012a

Product Name Change and Extended Capability **21-2**

Code Generation from MATLAB **21-2**

Code Generation from Any Level of Subsystem Hierarchy **21-3**

Automated Subsystem Hierarchy Flattening **21-3**

Support for Discrete Transfer Fcn Block **21-3**

User Option to Constrain Registers on Output Ports **21-3**

Distributed Pipelining for Sum of Elements, Product of Elements, and MinMax Blocks **21-3**

MATLAB Function Block Enhancements **21-3**
Multiple Accesses to RAMs Mapped from Persistent Variables **21-3**
Streaming for MATLAB Loops and Vector Operations **21-4**
Loop Unrolling for MATLAB Loops and Vector Operations **21-4**

Automated Code Generation from Xilinx System Generator for DSP Blocks **21-4**

Altera Quartus II 11.0 Support in HDL Workflow Advisor **21-4**

Automated Mapping to Xilinx and Altera Floating Point Libraries **21-4**

Vector Data Type for PCI Interface Data Transfers Between xPC Target and FPGA **21-4**

New Global Property to Select RAM Architecture **21-5**

Turnkey Workflow for Altera Boards **21-5**

HDL Support For Bus Creator and Bus Selector Blocks **21-5**

HDL Support For HDL CRC Generator Block **21-5**

HDL Support for Programmable Filter Coefficients **21-5**
Notes **21-6**

Synchronous Multiclock Code Generation for CIC Decimators and Interpolators **21-6**

Filter Block Resource Report Participation **21-6**

HDL Block Properties Interface Allows Choice of Filter Architecture **21-7**

HDL Support for FIR Filters With Serial Architectures and Complex Inputs **21-8**

HDL Support for External Reset Added for Proportional-Integral-Derivative (PID) and Discrete Time Integrator (DTI) Blocks **21-8**

R2021b

Version: 3.19

New Features

Bug Fixes

Compatibility Considerations

Model and Architecture Design

RAM style attributes for Intel/Altera and Microchip

You can now map large memory blocks such as `ultra` from the Xilinx® family and `M144k` from the Quartus® family on FPGAs during synthesis. To map these memory blocks, specify different attribute values for the synthesis attribute RAM style. To specify these attributes in the Simulink® HDL workflow, configure appropriate attribute values for **RAMDirective** in HDL Block properties. For more information, see "RAMDirective".

To specify these attributes in the MATLAB® HDL workflow, use the `RAMDirective` parameter value pair for `hdl.RAM` instantiation or use `hdlset_param`. You can set the `RAMDirective` by using either of these commands:

```
hRam = hdl.RAM('RAMType', 'Dual port', 'RAMDirective', 'ultra');
```

or

```
hdlset_param(<ram_block_name>, 'RAMDirective', <attribute_value>);
```

HDL code check for trigonometric blocks

A Model Advisor check is added for trigonometric blocks that use the LUT-based approximation method. This check runs on trigonometric blocks such as `cos`, `sin`, `cos+jsin`, and `sincos`. Set the approximation method to `Lookup`. This Model Advisor check determines that code generation is not supported for these trigonometric blocks with the LUT-based approximation. You can change the functionality of the blocks for successful code generation.

Timestamp macro in custom file header comments

You can now include the arguments such as date, time, and timestamp in your generated HDL code. In the custom file header and footer comments, these macros are supported for the arguments listed in this table.

Argument	Macro Syntax	Output Format
Date	__DATE__	DD-MM-YYYY
Time	__TIME__	hh:mm:ss
Timestamp	__TIMESTAMP__	DD-MM-YYYY hh:mm:ss

For more information, see "Custom File Header Comment".

Enhanced multiple enumeration in Verilog

In R2021b, HDL Coder supports Verilog® code generation for multiple enumerations that have the same element names but different element values. For example, for these enumeration definitions:

```
classdef(Enumeration) Colors < Simulink.IntEnumType
    enumeration
        Red(1),
        Blue(2),
        Green(4),
```

```
        Yellow(8),
        Purple(16),
        Orange(32),
        Black(64)

    end
end

classdef(Enumeration) ColorsAlternate < Simulink.IntEnumType
    enumeration
        Red(10)
    end
end
```

This is the generated code:

```
parameter ColorsAlternate_red = 4'd10;
parameter Colors_Red = 7'd1, Colors_Blue = 7'd2, Colors_Green = 7'd4, Colors_Yellow = 7'd8, Colors_Purple = 7'd16, Colors_Orange = 7'd32
```

HDL Coder can now generate Verilog code from a design that contains two enumeration classes, `Color` and `ColorsAlternate`, that both contain the same element name `Red` that have different element values, `1` and `10` respectively.

HDL Industry Coding Standard check for the presence of assignments to the same variable in multiple cascaded conditional regions

In R2021b, HDL Coder provides a coding standard to check for assignments to the same variable in multiple cascaded control regions within the same process block. HDL Coder points to the blocks that can generate such a coding style. If the style is not recommended for use in your production workflow, consider an alternative coding style or replace the block pointed to by the rule. For more information, see “Cascaded Conditional Region Variable Assignments”.

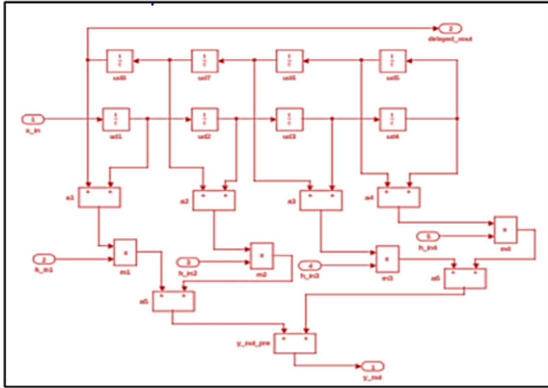
Layout choices for model generation

Starting in R2021b, the **Layout style** configuration parameter has been added to the **Model generation** pane in HDL Coder properties. You can now select the layout style for your generated model. The layout style has the options listed in this table:

Layout Style	Description
`None`	The generated model has no layout
`Default`	The model is generated using the default HDL Coder layout
`AutoArrange`	The model is generated using the Simulink layout

For better layout visualization of your generated model, choose the appropriate layout style option based on the design complexity. You can identify these layout style effects in the generated model when you configure any optimization on your input model. For more information, see “Layout Style”.

For example, see the generated layout of the input model that uses different layout style options.



Layout options

Layout style: Default

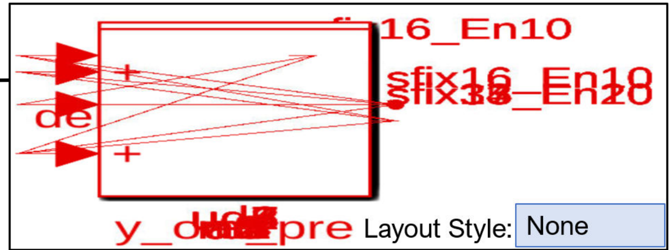
Default

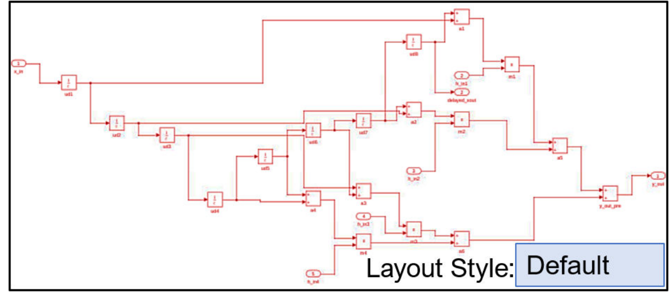
AutoArrange

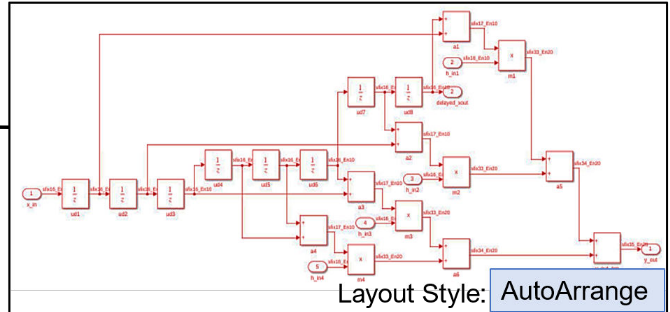
None

Auto signa

Inter-block ho







Block Enhancements

Newton-Raphson algorithm for Math Reciprocal block

HDL Reciprocal block functionality is now added to the Math Function block with `reciprocal` function. The Math Function block now supports the `Newton-Raphson` algorithm method. You can use following algorithm method and related HDL architecture for the Math Function block.

Algorithm Method (Block Parameters)	**Architecture (HDL Block Properties)**
`Exact`	`Math`
	`Reciprocal`
	`ReciprocalRsqrtBasedNewton`
	`ReciprocalRsqrtBasedNewtonSingleRate`
`Newton-Raphson`	`ReciprocalNewton`
	`ReciprocalNewtonSingleRate`

Starting in R2021b, it is recommended to use the Math Function block with the Newton-Raphson algorithm method instead of the HDL Reciprocal block. For details, see HDL Code Generation in Math Function.

Magnitude square function in Math Function block

HDL code generation now supports the `magnitude square` function in the Math Function block. Magnitude Square block is used in signal processing applications and supports fixed-point data types. It also supports `single` and `double` floating-point data types. You can use complex data types for the Magnitude Square block.

Half-precision data types for MATLAB Function block

HDL Coder now supports half-precision data types for the MATLAB Function block. The half-precision support is added for the basic arithmetic and logical functions used in the MATLAB Function block.

Double-Precision data types for Logarithmic block

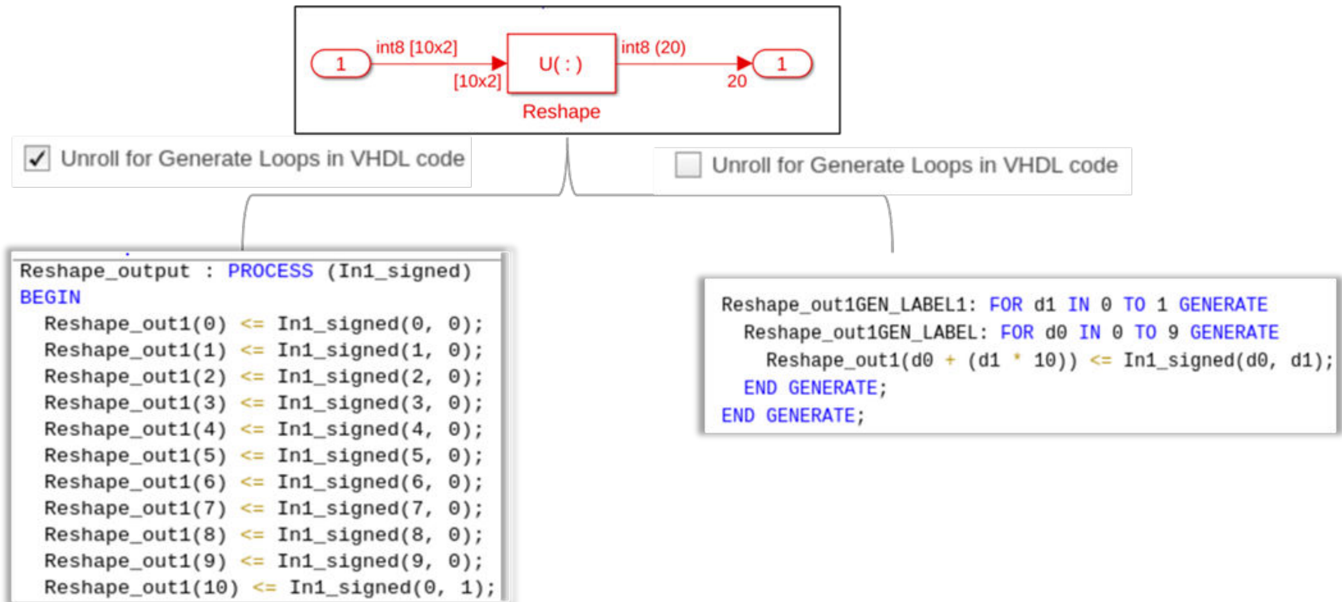
HDL code generation now supports double-precision data types for the `Log` function in the Math Function block and the MATLAB Function block.

For-Generate loops for Reshape and Concat blocks

In R2021b, the coding style for Reshape and Concat blocks has been improved. The generated HDL code has these coding style enhancements:

- For Reshape and Concat blocks, the HDL code is generated with For-Generate loops when you select the target language as `VHDL` or `System Verilog`.
- By default, the loop unrolled code is generated for the Reshape block the when target language is `Verilog`.

The generated code has better code readability, reduced lines of code and reduced code generation time. For example, this image shows code generated for the Reshape block with `For-Generate` loops and without `For-Generate` loops.



For more information, see “Unroll For-Generate Loops in VHDL code”.

Fixed-point output types for Divide block and Reciprocal block

In R2021b, HDL Coder extends the support for the Divide and Reciprocal blocks. Prior to R2021b, these blocks supported only output data types as `Inherit: Inherit via internal rule`. You can now use these output data types for the blocks:

- `Inherit: Keep MSB`
- `Inherit: Match scaling`
- `Inherit: Inherit via back propagation`
- `Inherit: Same as first input`
- `Integer types (uint8,int8,uint16,int16,uint32,int32,uint64,int64)`
- `Fixed point types`

To use these output data types for Divide and Reciprocal blocks, select **Architecture** to `ShiftAdd`. For more information, see Divide.

Limitation: Floating-point output data types are not supported for these blocks.

Enhanced HDL math library

HDLMathLib blocks now have enhanced output handling, where outputs are held to the previous value and do not change when the validOut is low.

4-D and 5-D lookup table support

In R2021b, HDL Coder enhances the support for the direct lookup table and the n-D lookup table. The code generation now supports 4-D and 5-D lookup tables. 4-D and 5-D lookup tables supports `half`, `single`, and `double` data types. You can also use flat or linear interpolation for 4-D and 5-D lookup tables. Linear interpolation is not supported for half data types. You can also use fully parallel and serial implementation for 4-D and 5-D lookup tables. For more information, see n-D Lookup Table.

Improved denormal optimizations for half-precision data types

Half-precision types have a lower dynamic range, so it is beneficial to always have denormal logic generated for the operation. When your design uses half-precision data types and **Handle Denormals** is `off`, not having denormal logic sometimes can cause numeric issues for very small values. To avoid such a scenario, make sure that denormals logic is `on` for the half-precision data types.

In R2021b, HDL coder enhances the functionality of denormals logic for the floating-point library. A new parameter value `Auto` is added for denormals logic. The default value for **Handle Denormals** is set to `Auto`. This option maps the denormal to `on` when your design uses half-precision data types. It maps denormal to `off` for `single` and `double` data types. The cost of adding denormal logic in hardware for half-precision types is very low. You can control the option by explicitly turning the denormal logic `on` or `off`. For more information, see “Handle Denormals”.

Improved multiplier partitioning DSP QoR

In R2021b, HDL Coder enhances the multiplier partitioning optimization for signed inputs. If you apply **Multiplier partitioning threshold** on your model that has signed inputs, the multiply operations are reduced by one when the inputs are signed to a partitioned multiplier. This enhancement reduces the digital signal processor (DSP) consumption and improves frequency.

This enhancement is not applicable for a model that contains a subsystem with **Distributed pipelining** turned `on` and **DP priority** set to `NumericalIntegrity`. In such cases, optimization is performed by using an additional 2-bit multiply operation.

Reset minimization in Native Floating-Point (NFP) for ASIC

In R2021a, when you set `MinimizeGlobalResets` to `on` for a subsystem that had NFP IP blocks and `NoResetInitializationMode` was set to `None`. The generated code did not have reset port. No registers were initialized. When both settings were specified, the output propagated `'X'` logic for the simulation instead of a few initial clock cycles for NFP IP blocks. The number of initial clock cycles for which output showed `‘X’` logic depended on the latency of the NFP IP blocks.

In R2021b, HDL Coder enhances the algorithm for `MinimizeGlobalResets` for Native Floating-Point IP blocks. Now, when you specify the preceding settings, `'X'` logic is propagated until only the latency of the block.

Set-Reset (SR) flip-flops

In R2021b, HDL Coder supports code generation for an S-R flip-flop block. See S-R Flip-Flop.

HDL Code Generation for Discrete State-Space block

In R2021b, HDL Coder supports code generation for the fixed-point Discrete State-Space block. See Discrete State-Space.

Trigger and event modes for subsystems, MATLAB Function blocks, and Stateflow blocks

Previously, HDL Coder supported only the rising and falling edge modes for blocks that support input triggers and events. In R2021b, HDL Coder supports either mode for blocks that support input triggers and events, such as:

- Trigger ports of subsystems. For more information, see Triggered Subsystem.
- Input events for Stateflow® charts. For more information, see “Activate a Stateflow Chart by Sending Input Events” (Stateflow).
- Input events for MATLAB Function blocks. For more information, see “Update method”.

Wireless HDL Toolbox Reference Applications: Implement 5G NR SIB1 recovery, WLAN receiver, and DVB-S2 PL header recovery

The “NR HDL SIB1 Recovery” (Wireless HDL Toolbox) reference application builds on the “NR HDL MIB Recovery” (Wireless HDL Toolbox) design and shows how to implement system information block type 1 (SIB1) decoding. The “NR HDL Downlink Receiver MATLAB Reference” (Wireless HDL Toolbox) example (renamed from "NR HDL Cell Search and MIB Recovery MATLAB Reference") now includes cell search, MIB recovery, and SIB1 recovery.

The “HDL Implementation of WLAN Receiver” (Wireless HDL Toolbox) example is an extension to the WLAN HDL Time and Frequency Synchronization example that was introduced in R2021a. The HDL Implementation of WLAN Receiver example detects frame format and decodes signal and data fields according to wireless local area network (WLAN) standards. This example supports a single-input single-output (SISO) 20 MHz bandwidth option for non-high-throughput (non-HT), high-throughput mixed mode (HT-MM), and very-high-throughput (VHT) frame formats.

The “DVB-S2 HDL PL Header Recovery” (Wireless HDL Toolbox) example implements Digital Video Broadcasting Satellite Second Generation (DVB-S2) receiver synchronization and a physical layer (PL) header recovery system that can handle radio frequency (RF) impairments to support DVB-S2 waveforms. The example shows how to perform frame synchronization and time, frequency, and phase offset estimation and correction. It also shows how to decode the PL header information.

These examples support HDL code generation and are ready for deployment to hardware.

Wireless HDL Toolbox Blocks: Model WLAN LDPC decoder, CCSDS RS decoder, DVBS2 symbol demodulator, and APP decoder

The WLAN LDPC Decoder block implements layered belief propagation with min-sum approximation and normalized min-sum approximation algorithms for decoding low-density parity-check (LDPC) codes according to these WLAN standards: 802.11n, 802.11ac, 802.11ax, and 802.11ad.

The CCSDS RS Decoder block decodes and recovers messages from a Reed-Solomon (RS) codeword according to the Consultative Committee for Space Data Systems (CCSDS) standard. The block

supports RS codewords (255, *k*), where *k* is a message length of 239 or 223. The block also supports shortened message lengths and interleaving depth values 1, 2, 3, 4, 5, and 8.

The DVBS2 Symbol Demodulator block demodulates complex constellation symbols to a set of log-likelihood ratio (LLR) values. The block supports π/2-BPSK, QPSK, 8-PSK, 16-APSK, and 32-APSK modulation types. It also supports multiple code rates according to the DVB-S2 standard. You can configure the modulation type and code rate during runtime while using this block.

The APP Decoder block decodes coded LLR values using the maximum a-posteriori probability (MAP) decoding algorithm and serves as a basic building block to implement a turbo decoder. The block accepts soft inputs and provides soft outputs, which can be useful for iterative decoding. The block supports terminated and truncated modes and these decoding rates: 1/2, 1/3, 1/4, 1/5, 1/6, and 1/7.

These blocks provide an interface and architecture for HDL code generation with HDL Coder.

Multipixel-Multicomponent Video Streaming: Implement color space conversion and demosaic interpolation algorithms for high-frame-rate color video

The Color Space Converter and Demosaic Interpolator Vision HDL Toolbox™ blocks now support multipixel-multicomponent streams.

The HDL implementation replicates the algorithm for each pixel in parallel.

The Color Space Converter block supports input matrices of *NumPixels*-by-3 values and output matrices of *NumPixels*-by-*NumComponents* values, where *NumComponents* is 3 or 1. The Demosaic Interpolator block accepts an input vector of *NumPixels*-by-1 values and returns an output matrix of *NumPixels*-by-3 values. The **ctrl** ports remain scalar, and the control signals in the `pixelcontrol` bus apply to all pixels in the matrix.

You can simulate System objects with a multipixel streaming interface, but System objects that use multipixel streams are not supported for HDL code generation. Use the equivalent blocks to generate HDL code for multipixel algorithms.

Reflection Padding: Pad image frames by reflecting around the edge pixel

Pad the edge of a frame by reflecting around the edge-pixel value. This padding method helps reduce edge contrast effects and can improve results for machine learning while maintaining the original frame size.

330	300	270	300	330
210	180	150	180	210
90	60	30	60	90
210	180	150	180	210
330	300	270	300	330

To use this feature, set the **Padding method** parameter to `Reflection` on any of these blocks.

- Line Buffer
- Image Filter
- Bilateral Filter
- Median Filter
- Corner Detector

For more information on padding methods, see “Edge Padding” (Vision HDL Toolbox).

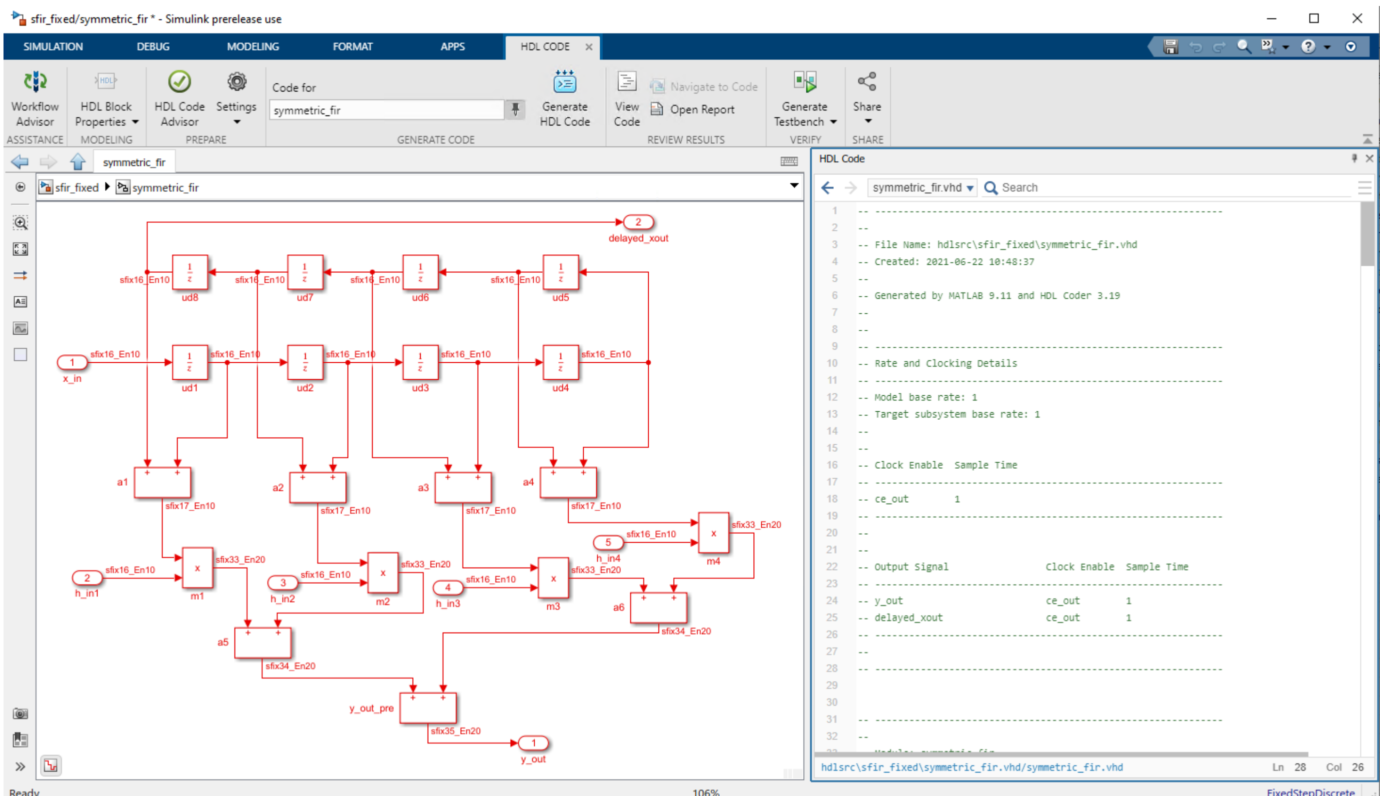
Code Generation and Verification

Code View: View your generated HDL code directly in Simulink model window

You can now view your generated HDL code alongside your model by using the Code view. Use this integration between code and model to:

- Quickly navigate from model elements to their generated code. When you click on a block in the model, the Code view highlights the code for the block and scrolls to the highlighted code lines.
- Trace lines of code to the model elements from which they were generated. In the Code view, click the line number hyperlink or code comment link to highlight the block that the code line traces to.
- Customize your generated code and verify that the results are correct by viewing the code and the model at the same time.

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, on the Simulink toolstrip, click the **View Code** button. At the top of the Code view, select the file that you want to display. You can dock or undock the Code view from the editor and minimize or expand the Code view. You can also use rich text capabilities such as code folding and hiding comments.



Stateflow multicycle path enhancements

In R2021a, when you sourced enable-based multicycle-path (MCP) constraints in the FPGA synthesis tool to relax the timing constraints on multicycle data paths, you saw that the MCP constraints were

not applied in the generated HDL code. Because HDL Coder generates MCP with timing controller logic, the synthesis tool optimizes this timing controller logic to convert it to the single clock domain design. The MCP constraints are not applied in the generated HDL code.

HDL Coder has improved the enable-based constraints algorithm to generate an MCP constraints file. Starting in R2021b, when you source enable-based MCP constraints, the attribute `direct_enable` is added to all the enable signals in your generated timing controller HDL code. Due to the `direct_enable` attribute, the synthesis tool preserves the timing control logic and the MCP constraints are applied in your design. This attribute is added to timing controller logic in both the Verilog and VHDL® designs. The attribute is supported in synthesis tools such as Xilinx Vivado®, Intel® Quartus, and Intel Quartus Pro.

Register-to-register path info option not recommended in HDL Coder

In R2021b, it is not recommended to use **Register-to-register path info** for generating multicycle path information. Use **Enable-based constraints** to meet the timing requirement of the multicycle paths in your model. For more information, see “Multicycle Path Constraints Parameters”.

Compatibility Considerations

The **Register-to-register path info** option is removed from the **Optimization** pane in HDL Coder Properties. You can still enable this option in the MATLAB command-line interface by using hdlset_param or makehdl commands. HDL Coder will remove this functionality in future releases. It is recommended to use Enable-based constraints for generating multicycle path information.

Execute chart at initialization option for Stateflow charts

In R2021b, HDL Coder supports disabling the Stateflow chart option `Execute (enter) chart at initialization`. This option is useful for when you want your chart to begin executing from a known configuration. This Stateflow option is available for Mealy and Classic charts, but not for Moore charts. For more information, see “Execute (enter) chart at initialization” (Stateflow).

HDL code generation performance improvement for matrix multiplication

HDL code generation for Simulink models that perform matrix multiplication operations by using the `DotProductStrategy` property set to `Fully Parallel (default)` now has significant performance improvement when you generate HDL code. For example, HDL code generation time for a 16-by-16 fixed-point matrix multiplication now takes 17 seconds, whereas it previously took 328 seconds.

In R2021b, HDL Coder generates concise code by vectorizing matrix multiplication inner products to create for-generate constructs and reduces the size of the generated model. For example, when you have two matrices of size m-by-n and n-by-p, the generated model has one product block with `m*n*p` vector size and n-by-1 adder blocks in a linear form that have `m*p` vector size. Prior to R2021b, the generated model had `m*n*p` individual product blocks and `(n-1)*m*p` individual adder blocks. To minimize multiplier and adder DSP consumption for `Fully Parallel (default)` matrix multiplication, set a streaming factor for the parent subsystem. A streaming factor of `m*n*p` results in a full serialization of the multiplication operators and near full serialization of the addition operations. Adding a sharing factor of n-1 can fully share the data-dependent addition operations.

Compatibility Considerations

In R2021b, when you set the DotProductStrategy property to Fully Parallel, you cannot use a sharing factor for successful resource sharing. Use a streaming factor instead. The Serial Multiply-Accumulate and Parallel Multiply-Accumulate settings for DotProductStrategy are not affected by the new optimization.

To replicate the old Fully Parallel behavior, use the new Fully Parallel Scalarized option. This new option for the legacy behavior is recommended for only smaller matrix sizes.

Speed and Area Optimizations

Enhanced sharing and streaming optimizations for matrix-types

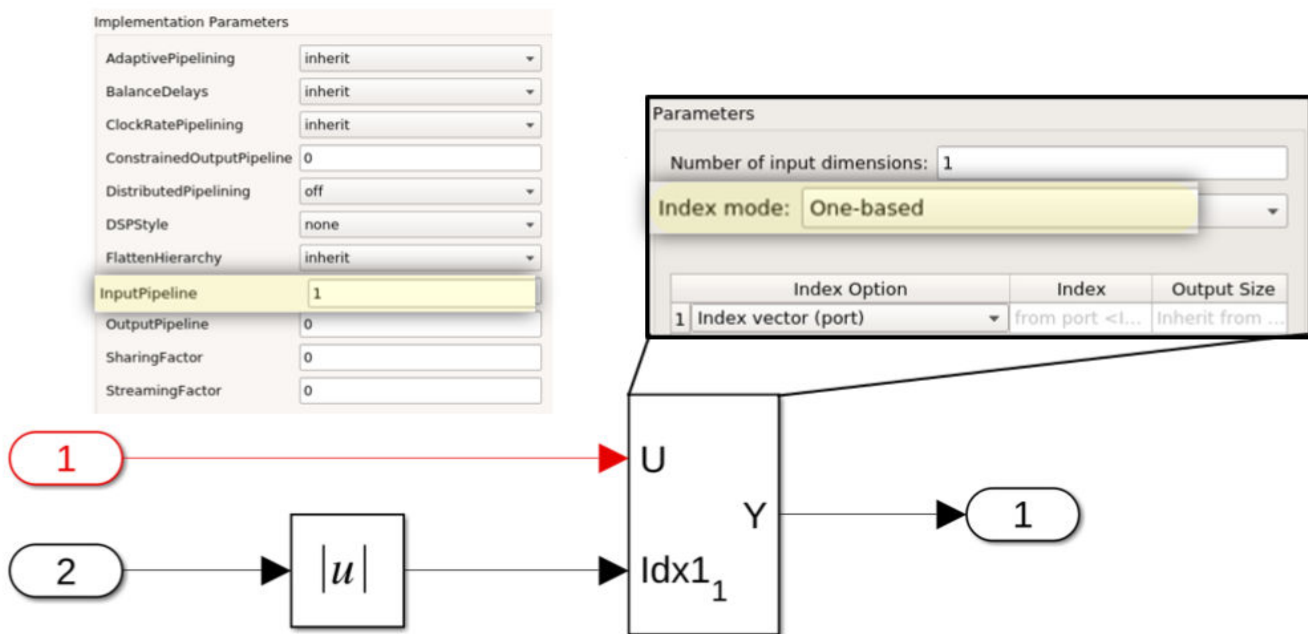
You can now configure sharing and streaming optimizations at the same time on a model that has matrix type inputs. To enable these optimizations on your input model, specify sharing and streaming factors in HDL block properties.

User control for tunable parameter processing and improve code generation time

You can now disable the design under test (DUT) port generation for tunable parameters by disabling `Enable HDL DUT Port Generation for Tunable Parameters`. Optimize your code generation time by disabling the DUT inport generation for tunable parameters and DUT output port generation for test points.

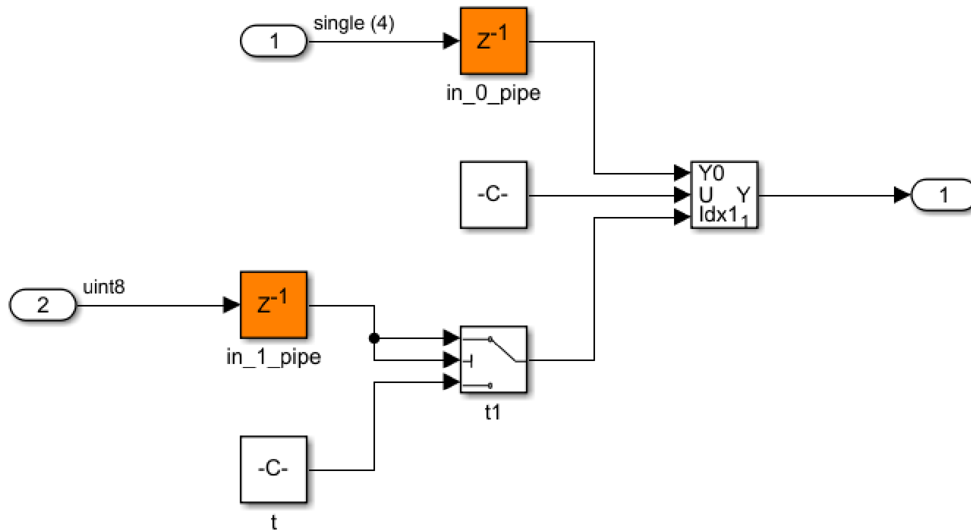
Improved zero-protection in Simulink-to-HDL

In R2021b, HDL Coder inserts zero-protection switches that prevent input zeros from pipeline initial values reaching blocks that do not work well with zero values. For example, consider the following DUT and block-specific properties:



The DUT has one input pipeline register specified in the DUT's HDL Block parameters. The selector blocks use one-based indexing, so a zero at the index port is an invalid argument. Prior to R2021b, the generated model failed during simulation. In R2021b, for the preceding model, you see a zero-protection switch is correctly applied in the generated model and in the generated code. The

simulation of the generated model is successfully completed.



To avoid protection logic, minimize zeros feeding into blocks that do not expect zero run-time values.

Minimize intermediate initialization of variables in generated HDL code

In R2021b, HDL Coder performs optimizations during code generation to reduce area usage while maintaining functionality and performance. For example, this figure shows the difference in the

generated code prior to R2021b and code generated in R2021b.

```
1   function y = foo2(u, v)
2   %codegen
3   t = u + v;
4   y = zeros(1,10,'like',u);
5
6   for i = 1:10
7       y(i) = 2*t + i;
8       for j = 1:5
9           y(i) = y(i) + j*4;
10      end
11  end
12  end
```

Code Generated before 21b

```
76      add_cast_0 := resize(v_unsigned, 10);
77      t := add_cast + add_cast_0;
78
79      FOR i IN 0 TO 9 LOOP
80        y1(i) := to_unsigned(16#00#, 7);
81        add_cast_1(i) := resize(t & '0', 12);
82        add_cast_2(i) := resize(add_cast_1(i), 13);
83        add_temp(i) := to_signed(i + 1, 32);
84        add_cast_3(i) := unsigned(add_temp(i)(3 DOWNTO 0));
85        add_cast_4(i) := resize(add_cast_3(i) & '0' & '0' & '0' & '0', 13);
86        add_temp_0(i) := add_cast_2(i) + add_cast_4(i);
87        y1(i) := add_temp_0(i)(8 DOWNTO 2);
88
89        FOR j IN 0 TO 4 LOOP
90          add_cast_5(j) := resize(y1(i), 8);
91          mul_temp(j) := to_signed((j + 1) * 4, 64);
92          add_cast_6(j) := mul_temp(j)(31 DOWNTO 0);
93          add_cast_7(j) := unsigned(add_cast_6(j)(4 DOWNTO 0));
94          add_cast_8(j) := resize(add_cast_7(j) & '0' & '0', 8);
95          add_temp_1(j) := add_cast_5(j) + add_cast_8(j);
96          y1(i) := add_temp_1(j)(6 DOWNTO 0);
97        END LOOP;
98
99        y_tmp(i) <= y1(i);
100     END LOOP;
```

Code Generated in 21b

line 80 is removed

```
76      add_cast := resize(u_unsigned & '0' & '0', 10);
77      add_cast_0 := resize(v_unsigned, 10);
78      t := add_cast + add_cast_0;
79
80      FOR i IN 0 TO 9 LOOP
81        add_cast_1(i) := resize(t & '0', 12);
82        add_cast_2(i) := resize(add_cast_1(i), 13);
83        add_temp(i) := to_signed(i + 1, 32);
84        add_cast_3(i) := unsigned(add_temp(i)(3 DOWNTO 0));
85        add_cast_4(i) := resize(add_cast_3(i) & '0' & '0' & '0' & '0', 13);
86        add_temp_0(i) := add_cast_2(i) + add_cast_4(i);
87        slice_temp := add_temp_0(i)(8 DOWNTO 2);
88
89        FOR j IN 0 TO 4 LOOP
90          add_cast_5(j) := resize(slice_temp, 8);
91          mul_temp(j) := to_signed((j + 1) * 4, 64);
92          add_cast_6(j) := mul_temp(j)(31 DOWNTO 0);
93          add_cast_7(j) := unsigned(add_cast_6(j)(4 DOWNTO 0));
94          add_cast_8(j) := resize(add_cast_7(j) & '0' & '0', 8);
95          add_temp_1(j) := add_cast_5(j) + add_cast_8(j);
96          slice_temp := add_temp_1(j)(6 DOWNTO 0);
97        END LOOP;
98
99        y_tmp(i) <= slice_temp;
100     END LOOP;
```

```
1     function Cv = matadd(Av, Bv)
2     %#codegen
3 -       A = reshape(Av, 3, 3);
4 -       B = reshape(Bv, 3, 3);
5
6 -       C = A + B;
7
8 -       Cv = reshape(C, 9, 1);
9     end
```

```
48      VARIABLE A : matrix_of_unsigned32(0 TO 2, 0 TO 2);
49      VARIABLE B : matrix_of_unsigned32(0 TO 2, 0 TO 2);
50      VARIABLE C : matrix_of_unsigned32(0 TO 2, 0 TO 2);
51      VARIABLE add_temp : vector_of_unsigned33(0 TO 8);
52      VARIABLE cast_temp : unsigned(7 DOWNTO 0);
53      VARIABLE cast_temp_0 : unsigned(15 DOWNTO 0);
54      VARIABLE mul_temp : unsigned(15 DOWNTO 0);
55      VARIABLE oldIdx : INTEGER;
56    BEGIN
57      cast_temp := to_unsigned(16#00#, 8);
58      cast_temp_0 := to_unsigned(16#0000#, 16);
59      mul_temp := to_unsigned(16#0000#, 16);
60      oldIdx := 0;
61
62      FOR i IN 0 TO 2 LOOP
63        FOR i2 IN 0 TO 2 LOOP
64          cast_temp := unsigned(resize(to_signed(i, 32), 8));
65          cast_temp_0 := unsigned(resize(to_signed(i2, 32), 16));
66          mul_temp := cast_temp * to_unsigned(16#03#, 8);
67          oldIdx := to_integer(mul_temp + cast_temp_0);
68          A(i2, i) := Av_unsigned(oldIdx);
69          B(i2, i) := Bv_unsigned(oldIdx);
70          add_temp(oldIdx) := resize(A(i2, i), 33) + resize(B(i2, i), 33);
71          IF add_temp(oldIdx)(32) /= '0' THEN
72            C(i2, i) := X"FFFFFFFF";
73          ELSE
74            C(i2, i) := add_temp(oldIdx)(31 DOWNTO 0);
75          END IF;
76          Cv_tmp(oldIdx) <= C(i2, i);
77        END LOOP;
78      END LOOP;
79
80    END PROCESS MATLAB_Function_1_output;
81
82
83    outputgen: FOR k IN 0 TO 8 GENERATE
84      Cv(k) <= std_logic_vector(Cv_tmp(k));
```

```
33      -- Signals
34      SIGNAL Av_unsigned          : vector_of_unsigned32(0 TO 8);  -- uint32 [9]
35      SIGNAL Bv_unsigned          : vector_of_unsigned32(0 TO 8);  -- uint32 [9]
36      SIGNAL Cv_tmp               : vector_of_unsigned32(0 TO 8);  -- uint32 [9]
37      SIGNAL add_temp             : vector_of_unsigned33(0 TO 8);  -- ufix33 [9]
38
39    BEGIN
40      outputgen2: FOR k IN 0 TO 8 GENERATE
41        Av_unsigned(k) <= unsigned(Av(k));
42      END GENERATE;
43
44      outputgen1: FOR k IN 0 TO 8 GENERATE
45        Bv_unsigned(k) <= unsigned(Bv(k));
46      END GENERATE;
47
48
49      Cv_tmp_gen: FOR t_0 IN 0 TO 8 GENERATE
50        add_temp(t_0) <= resize(Av_unsigned(t_0), 33) + resize(Bv_unsigned(t_0), 33);
51
52        Cv_tmp(t_0) <= X"FFFFFFFF" WHEN add_temp(t_0)(32) /= '0' ELSE
53            add_temp(t_0)(31 DOWNTO 0);
54      END GENERATE Cv_tmp_gen;
55
56
57      outputgen: FOR k IN 0 TO 8 GENERATE
58        Cv(k) <= std_logic_vector(Cv_tmp(k));
59      END GENERATE;
```

```
1      function out = foo(x, y, z, a)
2
3 -    c = fi(3,0,7,4);
4
5 -    if a > fi(0,0,7,4)
6 -        i = x + c;
7 -    else
8 -        i = x - c;
9 -    end
10
11 -   if a > fi(0,0,7,4)
12 -       j = y + c;
13 -   else
14 -       j = y - c;
15 -   end
16
17 -   if a > fi(0,0,7,4)
18 -       k = z + c;
19 -   else
20 -       k = z - c;
21 -   end
22
23 -   out = i + j + k;
24
25 -   end
26
```

```
72      BEGIN
73        sub_cast := to_unsigned(16#00#, 8);
74        add_cast := to_unsigned(16#00#, 8);
75        sub_cast_0 := to_unsigned(16#00#, 8);
76        add_cast_0 := to_unsigned(16#00#, 8);
77        sub_cast_1 := to_unsigned(16#00#, 8);
78        add_cast_1 := to_unsigned(16#00#, 8);
79        --HDL code generation from MATLAB function: foo
80        IF a_unsigned > to_unsigned(16#00#, 7) THEN
81          add_cast := resize(x_unsigned, 8);
82          tmp := add_cast + to_unsigned(16#30#, 8);
83        ELSE
84          sub_cast := resize(x_unsigned, 8);
85          tmp := sub_cast - to_unsigned(16#30#, 8);
86        END IF;
87        IF a_unsigned > to_unsigned(16#00#, 7) THEN
88          add_cast_0 := resize(y_unsigned, 8);
89          tmp_0 := add_cast_0 + to_unsigned(16#30#, 8);
90        ELSE
91          sub_cast_0 := resize(y_unsigned, 8);
92          tmp_0 := sub_cast_0 - to_unsigned(16#30#, 8);
93        END IF;
94        IF a_unsigned > to_unsigned(16#00#, 7) THEN
95          add_cast_1 := resize(z_unsigned, 8);
96          tmp_1 := add_cast_1 + to_unsigned(16#30#, 8);
97        ELSE
98          sub_cast_1 := resize(z_unsigned, 8);
99          tmp_1 := sub_cast_1 - to_unsigned(16#30#, 8);
100       END IF;
101       add_cast_2 := resize(tmp, 9);
102       add_cast_3 := resize(tmp_0, 9);
103       add_temp := add_cast_2 + add_cast_3;
104       add_cast_4 := resize(add_temp, 10);
105       add_cast_5 := resize(tmp_1, 10);
106       out_tmp <= add_cast_4 + add_cast_5;
107     END PROCESS foo_1_output;
```

```
72      BEGIN
73        sub_cast := to_unsigned(16#00#, 8);
74        add_cast := to_unsigned(16#00#, 8);
75        sub_cast_0 := to_unsigned(16#00#, 8);
76        add_cast_0 := to_unsigned(16#00#, 8);
77        sub_cast_1 := to_unsigned(16#00#, 8);
78        add_cast_1 := to_unsigned(16#00#, 8);
79        --HDL code generation from MATLAB function: foo
80        IF a_unsigned > to_unsigned(16#00#, 7) THEN
81          add_cast := resize(x_unsigned, 8);
82          tmp := add_cast + to_unsigned(16#30#, 8);
83          add_cast_0 := resize(y_unsigned, 8);
84          tmp_0 := add_cast_0 + to_unsigned(16#30#, 8);
85          add_cast_1 := resize(z_unsigned, 8);
86          tmp_1 := add_cast_1 + to_unsigned(16#30#, 8);
87        ELSE
88          sub_cast := resize(x_unsigned, 8);
89          tmp := sub_cast - to_unsigned(16#30#, 8);
90          sub_cast_0 := resize(y_unsigned, 8);
91          tmp_0 := sub_cast_0 - to_unsigned(16#30#, 8);
92          sub_cast_1 := resize(z_unsigned, 8);
93          tmp_1 := sub_cast_1 - to_unsigned(16#30#, 8);
94        END IF;
95        add_cast_2 := resize(tmp, 9);
96        add_cast_3 := resize(tmp_0, 9);
97        add_temp := add_cast_2 + add_cast_3;
98        add_cast_4 := resize(add_temp, 10);
99        add_cast_5 := resize(tmp_1, 10);
100       out_tmp <= add_cast_4 + add_cast_5;
101     END PROCESS foo_1_output;
```

Improved optimizations for conditional subsystems

You can now more effectively apply optimizations such as streaming, resource sharing, and clock rate pipelining to enabled and triggered subsystems that contain integer delays that have a delay length greater than one. For example, you can apply resource sharing where you have the same resource across multiple conditional subsystems within your DUT.

Optimizations will not be applied if the configuration parameter `TransformNonZeroInitValDelay` is disabled and the delay has a nonscalar expandable initial value.

Delay-balancing behavior standardization in BalanceDelays=off network

Turning off delay balancing during code generation is discouraged. If any optimizations are enabled, you must balance any pipelines introduced as a part of those optimizations. If you do not balance automatically inserted pipeline delays, you might encounter issues in generated code deployed in hardware. You can select an additional diagnostic option check `TreatBalanceDelaysOffAs` to detect the presence of unbalanced delays in the generated HDL code to highlight that `BalanceDelays` is set to off. The goal is to avoid unbalanced delays in your model. For more information, see “Check for presence of unbalanced delays in generated HDL code”.

Starting in R2021b, the `BalanceDelays` option is no longer a global option in the model configuration parameters. If you decide to disable delay balancing for your model by using the command line (not recommended), see “Delay Balancing Considerations”. If you turn off `BalanceDelays`, also turn off any features that automatically generate pipelines or add latency, such as optimizations like resource sharing, streaming, pipelining, or enabling the `MapToRAM` HDL block property of a lookup table (LUT) block. Manually balance the delays instead.

Lookup Table blocks mapping to RAM and adaptive pipelining

In R2021b, the mapping of Lookup Table blocks to RAM is not associated with the adaptive pipelining optimization option. You can now independently enable the mapping of Lookup Table blocks to RAM by using the `Map lookup tables to RAM` option. For more information, see “Map lookup tables to RAM”. Select this option to add non-reset delays next to the lookup tables and to save resources on the target FPGA hardware by mapping lookup tables to block RAM.

In R2021b, the default `MapToRAM` HDL block property default is `Inherit`. The default option prevents you from having to search through your model and individually set the `MapToRAM` option to `on` for each Lookup Table block.

Prior to R2021b, when you enabled adaptive pipelining, Lookup Table blocks were mapped to RAM. Adaptive pipelining inserted pipeline registers at the input port, output port, or both input and output ports of certain blocks to create patterns that efficiently mapped blocks to DSP units on the target FPGA device. You should enable adaptive pipelining based on your design requirements. For more information, see “Design Patterns That Require Adaptive Pipelining”.

You can highlight the Lookup Table blocks in your model that are mapped to RAM by enabling `Highlight lookup tables mapped to RAM`. For more information, see “Highlight lookup tables mapped to RAM”.

IP Core Generation and Hardware Deployment

Microsemi Libero System On A Chip (SoC) support for IP core generation workflow

In R2021b, HDL Coder enhances support for the Microsemi® Libero® SoC synthesis tool. You can now use the Microsemi Libero SoC tool for IP core generation workflow in HDL Workflow Advisor. Prior to R2021b, Microsemi Libero SoC tool supported only the Generic/ASIC FPGA workflow.

To generate an IP core by using the Microsemi Libero SoC synthesis tool, use `Generic Microchip Platform` as your **Target Platform**. The support extends for Microchip FPGA families such as IGLOO 2, RTG4®, Polarfire®, and SmartFusion® 2. You can use the generic microchip platform with the external ports interface and AXI4 (AXI4 Master and AXI4 slave) interface. AXI4-Lite interface is only supported for the Polarfire FPGA.

Using the IP core generation workflow, you can instantiate the IP core to the Microsemi Libero SoC SmartDesign. HDL Coder generates a `TCL` file to instantiate the IP core in Microsemi Libero SoC SmartDesign. Then you can generate the bitstream for your design by using the Microsemi Libero SoC.

MATLAB Prototyping API Enhancements: Support complex data in AXI4 Stream Interface and input register readback in AXI4 Interface

In R2021b, HDL Coder has enhanced the MATLAB prototyping API to support complex data and input register readback.

HDL Coder supports complex data type streaming on the AXI4 stream interface. You can now specify the complex data for your DUT ports in the `hdlcoder.DUTPort` API. A new property `IsComplex` is added in `hdlcoder.DUTPort` API to set the complex data type for the ports. The ports support complex data for a 64-bit word length.

You can readback the input registers for your DUT ports in the `hdlcoder.DUTPort` API by using the AXI-Slave interface. The readback feature is supported in both the Xilinx and Intel reference designs. When you select **Enable readback on AXI4-Slave write registers**, in your generated setup script, the input ports directions are mapped to the `INOUT` in `hdlcoder.DUTPort` API.

Upgrade to Intel Quartus Pro 20.2

HDL Coder has been tested with Intel Quartus Pro 20.2.

Inserted JTAG AXI Master at fixed frequency to avoid timing issue

In R2021a, when you inserted MATLAB JTAG AXI Master to control a generated IP Core and set the target frequency to greater than 100 MHz, you might have experienced timing failure for your reference design.

In R2021b, HDL Coder resolves the timing failure issue by controlling the clock frequency of the MATLAB JTAG AXI Master based on the target frequency of the reference design. The clock frequency of the inserted MATLAB JTAG AXI Master is fixed to 50 MHz when the target frequency of your design exceeds 100 MHz. For the target frequency less than or equal to 100 MHz, the MATLAB JTAG AXI Master is driven at the same clock frequency as your Design Under Test (DUT) clock.

Unsupported tool version in HDL workflow advisor

In R2021b, a new UI check box **Allow unsupported version** is added to the HDL Workflow Advisor in the **Set Target Device and Synthesis Tool** dialog box. HDL Workflow Advisor now runs the check for the unsupported tool version and asks whether you want to use the unsupported tool version. You can use the unsupported tool version by selecting the **Allow unsupported version** checkbox and creating a project in the HDL Workflow Advisor.

This option is supported for all the third-party FPGA synthesis tools. The support is extended for these targeted workflows:

- Generic ASIC/FPGA
- IP Core Generation
- FPGA Turnkey
- Simulink Real-Time™ FPGA I/O
- FPGA-in-the-Loop

HDL Coder continuously upgrades support for the latest tool version. It is recommended to use the supported tool version in the HDL Workflow Advisor for better synthesis results. For more information, see “HDL Workflow Advisor Tasks”.

Multicycle path constraint packaging for IP core

In R2021b, when you apply the multicycle path constraints in the IP core generation workflow, the generated MCP constraints file is now packaged with the IP core. With this enhancement, you can directly use the generated MCP constraints to the IP core in the Vivado and Qsys project. You do not need to manually insert the constraints by using a separate TCL script. HDL Coder supports this packaging for the Xilinx and Intel workflows.

HDL Coder Workflow Advisor: Option to expose DUT clock enable port and clock enable output port

In R2021b, the HDL Coder Workflow Advisor enables you to expose the design under test (DUT) block clock enable input and clock enable output port. You can use the clock enable input port to trigger the DUT from upstream IPs. Use the clock enable output port to drive or synchronize with other custom IPs.

Devicetree generation for IP cores

When using the Hardware-Software (HW/SW) codesign workflow from HDL Coder, you must configure the ARM® processor with a correct devicetree. A devicetree is a file that describes hardware devices accessible to the ARM processor. On system-on-chip (SoC) boards, the devicetree must include details about accessing the FPGA interfaces.

Previously, HDL Coder allowed only fixed, precompiled devicetree files to be specified for a reference design. In R2021b, HDL Coder allows devicetree source files to be specified for boards and reference designs. It also supports the dynamic generation of parts of the devicetree corresponding to the HDL Coder generated IP core. You can:

- Register new devicetree files to the board plugin file and reference design plugin file by using the `addDeviceTree` function.

- Specify an include file to compile the code against by using the addDeviceTreeIncludeDirectory function.
- Declare the existence of a processing system (PS) in the reference design by using the HasProcessingSystem function.
- Enable generation of devicetree nodes for the HDL Coder generated IP core by using the GeneratedIPCoreDeviceTreeNodes function.
- Add AXI4-Slave and AXI4-Stream interfaces to the devicetree if these interfaces connect to the ARM processor. Use the addAXI4SlaveInterface and addAXI4StreamInterface functions.

See "Generate Device tree for IP Core".

Updates to addAXI4StreamInterface function for fpga hardware connection object

In R2021b, you can specify the read data and write data widths on the AXI4-Stream interface for your fpga hardware connection object by using the ReadDataWidth and WriteDataWidth name-value arguments. The allowed width values are 8, 16, 32, 64, and 128. The HDL Coder generated software interface script includes the ReadDataWidth and WriteDataWidth name-value arguments.

Reset AXI4-Stream TLAST counter

When mapping DUT ports to an AXI4-Stream master interface, you can optionally model a TLAST protocol signal. If you do not model the TLAST signal, HDL Coder generates this signal for you. It also generates a programmable register in the IP core that contains the frame length. In the generated IP core, the TLAST signal is asserted when the number of valid samples counts up to the frame length. In previous releases, changing the frame length through the programmable register did not reset the state of the counter, causing the TLAST signal to not asset properly for partial frame transfers. Starting in R2021b, if the frame length register is changed in the middle of a frame, the counter state is reset to zero and the TLAST signal is asserted early.

HDL Coder Workflow Advisor: Improved code generation times

In R2021b, HDL Coder improves HDL Coder Workflow Advisor code generation times when using the MATLAB command-line interface by reducing the number of times a model is compiled. The code generation time has been reduced by a factor of three when using the HDL Coder Workflow Advisor command-line interface workflow. For example, HDL Coder Workflow Advisor command-line code generation times for models prior to R2021b that took 60 minutes now takes 20 minutes.

HDL Coder Workflow Advisor: Resource and timing report enhancement

In R2021b, the resource report and timing report of the HDL Workflow Advisor has been enhanced. The resource summary now shows the available FPGA resources such as LUT Slices, DSPs, Slice Registers, and many more in a target device and its utilization percentage in your design. The timing summary shows clock frequency of the design. You can view this enhanced report in HDL Workflow Advisor UI and from MATLAB command-line interface. This enhancement works for the Xilinx and Intel (Altera®) FPGA devices.

Data type for Speedgoat PCIe Interface: Map bus data types to Speedgoat PCIe Interface

When using the Simulink Real-Time FPGA I/O workflow, in the **Target Platform Interface table**, you can map bus signals at the DUT ports to PCIe interfaces.

The bus signals workflow includes an **IP Core Generation** report that displays address offsets of PCIe interface-accessible registers generated for each bus element scalar and vector datatype in the **Register Address Mapping** section. The top-level and sub-level bus data types do not have a register offset address. The address mapping for scalar and vector bus elements is not contiguous. See “Map Bus Data Types to PCIe Interface”.

HDL Coder Support Package for Xilinx RFSoC Devices: Generate IP core and deploy reference designs on Xilinx RFSoC devices

HDL Coder Support Package for Xilinx RFSoC Devices enables the generation of IP cores that can integrate into RFSoC devices. You can generate an HDL IP core by mapping the DUT ports to I/Os and AXI interfaces. This feature enables you to connect your algorithm to the RF tiles and the external DDR memory.

This support package includes reference designs for popular RFSoC development kits, including Xilinx Zynq® UltraScale+™ RFSoC ZCU111 and Xilinx Zynq UltraScale+ RFSoC ZCU216 evaluation kits.

You can use the IP core generation workflow to automate integration, execution, and verification of reference designs for RFSoC platforms. You can also interactively send and retrieve a frame of data to an FPGA. You can control the AXI registers in your generated IP core from MATLAB by using the generated MATLAB software interface script. For more information, see “Generate Software Interface Script to Probe and Rapidly Prototype HDL IP Core”.

You can use the SoC Blockset™ product for system-level modeling of RFSoC devices, exportation of custom reference designs for Xilinx RFSoC devices, and deployment of complete SoC applications, including executables for ARM Cortex®-A53 processors.

For the complete documentation of this support package, see “HDL Coder Support Package for Xilinx RFSoC Devices”.

Simscape Hardware-in-the-Loop Workflow

Support multiple solver times in Simscape models

In R2021b, Simscape HDL Workflow Advisor enables you to apply different solver times to different domains within the same Simscape™ model. Previously, in Simscape HDL Workflow Advisor, you could not apply different solver times to different domains within the same Simscape model.

Enable FPGA parameters in the protected model

Previously, simulation of protected models failed due to a mismatch in the parameter values between the protected model and top model:

- Family
- Device Name
- Package Name
- Speed Value
- Target Frequency

In R2021b, HDL Coder enables a difference in the values of these parameters between the protected model and the top model so that you can simulate and generate HDL code for top model and protected models while retaining top model synthesis parameters settings.

RAM mapping for partition solver

In R2021b, HDL Coder optimizes the resource utilization of your Simscape models deployed to FPGAs by enabling you to map state-space parameters to RAM. For more information, see “Map State Space Parameters to RAMs”.

R2021a

Version: 3.18

New Features

Bug Fixes

Compatibility Considerations

Model and Architecture Design

Half precision floating-point example for Field-Oriented Control algorithm

You can now generate HDL code for the Field-Oriented Control (FOC) algorithm for a Permanent Magnet Synchronous Machine (PMSM) that is implemented by using half-precision floating-point types.

See Floating Point Support: Field-Oriented Control Algorithm.

This table compares synthesis results for half-precision floating-point type, single-precision floating-point type, and signed fixed-point type with word length 16 and fraction length 10 (`fixdt(1,16,10)`) for a Xilinx Virtex®-7 xc7v2000t device.

Resources	FOC_Half	FOC_Single	FOC_Sfix16_En10
Fmax (MHz)	100.47	82.69	63.123
Slices	3348	5928	336
LUTs	9539	15853	1119
DSPs	19	40	21

Comments tab in Global Settings pane and option to disable comments

In the Configuration Parameters dialog box, on the **HDL Code Generation** > **Global Settings** tab, you can now use a new **Comments** tab to specify comment options for HDL code generation. On this tab, you can specify whether to enable or disable the comment options by using a new **Enable Comments** check box.

See Generate Code with Annotations or Comments.

HDL Code Advisor check for file extension based on target language

Previously, the HDL Code Advisor check **Check VHDL file extension** identified whether the file extension is specified as `.vhd` when you generate code with VHDL as the target language.

In R2021a, the check is renamed as **Check file extension** folder. When you run the check, it also identifies whether the file extension is specified as `.v` when you generate code with Verilog as the target language.

See Check file extension.

Hard Floating Point Support using Intel Quartus Pro

In R2021a, you can select Intel Quartus Pro as the target synthesis tool when using the `makehdl` function and the Generic ASIC/FPGA workflow. These limitations apply when using the Intel Quartus Pro tool:

- The HDL code generation language must be set to VHDL. For more information, see Language
- The floating-point library must be set to `Altera Megafunctions (ALTERA FP FUNCTIONS)`. For more information, see Floating Point IP Library

Block Enhancements

Enhancement to parameterized HDL code generation for 1-D and 2-D mask values

You can now specify 1-D vectors and 2-D matrices as mask values and generate parameterized HDL code from the masked subsystems by using the Generate parameterized HDL code from masked subsystem parameter.

HDL code generation for For Each Subsystem block with 1-D and 2-D partitioning of mask parameters

You can now generate HDL code for the For Each Subsystem block with 1-D and 2-D partitions of mask parameter values. To partition the mask parameters, in the **Parameter Partition** tab of the For Each block, select each mask parameter that you want to partition, and then specify the **Partition Dimension** and **Partition Width**.

See Repeat an Algorithm Using a For Each Subsystem.

HDL code generation for For Each Subsystem block with matrix ports

In R2021a, you can use 2-D matrices at the ports of the For Each Subsystem block for HDL code generation. You can partition the input into elements, vectors, and subarrays. After performing the computations, the elements are concatenated at the output to form the 2-D matrix result.

See Generate HDL Code for Blocks Inside For Each Subsystem.

HDL code generation for Interval blocks and additional Detect blocks

HDL Coder now supports code generation for these blocks:

- Detect Fall Negative
- Detect Fall Nonpositive
- Detect Rise Nonnegative
- Detect Rise Positive
- Interval Test
- Interval Test Dynamic

You can also access these blocks from the **HDL Coder > Logic and Bit Operations** in the Simulink Library Browser.

ShiftAdd architecture for Product block to avoid DSP consumption

In R2021a, you can use the `ShiftAdd` architecture for the Product block. Use this architecture to map the Product block implementation to lookup tables instead of DSP units on the target FPGA device because it performs multiple shift and add operations.

See UsePipelines.

HDL Coder library for fixed-point mathematical function blocks with latency

You can now use a `HDLMathLib` library to generate HDL code for fixed-point mathematical function blocks. The library includes these blocks with control ports:

- Sqrt
- atan2
- Sin
- Cos
- Sincos
- Cos+jSin
- Divide
- Reciprocal

To use these blocks in your model, at the MATLAB Command Window, enter:

```
HDLMathLib
```

For examples that describe more about the blocks and how to use them, see Implement Control Signals Based Mathematical Functions using HDL Coder.

Count hit port for HDL Counter block to indicate when count value resets

In the Block Parameters dialog box of the HDL Counter block, you can now specify a **Count hit port**. This port indicates when the counter resets by outputting the value `1` when the count value resets to the **Initial value**. The **Count hit port** port and **Count direction port** are mutually exclusive.

3-D lookup table support

Starting in R2021a, HDL Coder supports 3-D direct lookup tables and 3-D n-D lookup tables. When you use inputs and breakpoints that are floating-point values, use 3-D n-D lookup tables. When you use inputs that are fixed-point values, use 3-D direct lookup tables. 3-D n-D lookup tables support flat and linear interpolation from your table data. Floating-point support for the blocks is limited to `single`, `double`, and `half` data types for HDL code generation.

See Direct Lookup Table (n-D), n-D Lookup Table.

HDL Code Generation for Data Type Conversion block supports enumerated data types

You can now use enumerated signals at the ports of the Data Type Conversion block for HDL code generation. You can use the Data Type Conversion block to convert the enumeration data to integer or integer-to-enumeration data. Specify your enumerated data to your Data Type Conversion block. For example:

```
Simulink.defineIntEnumType('ModelName', ...
{'Red','Yellow','Orange','Blue','Green'},[40;50;60;70;80],'DefaultValue','Red');
```

Enhancement to HDL code generation for Sqrt block

In R2021a, HDL code generated for a Sqrt block with its **Architecture** set to `SqrtFunction` and **UseMultiplier** set to `off` has improved area and operating frequency compared to previous releases.

For example, this table illustrates the performance of the square root operation by setting **UseMultiplier** to `off`, and **LatencyStrategy** to `inherit` for a Xilinx device.

UseMultiplier	Fmax (MHz)	LUTs	DSP Slices
`off`	345	543	255
`on`	235	921	423

New HDL-optimized Simulink blocks for reciprocal, divide, and modulo

Starting in R2021a, Fixed-Point Designer™ has additional Simulink blocks for performing reciprocal, division and modulo operations:

- Complex Divide HDL Optimized
- Real Divide HDL Optimized
- Real Reciprocal HDL Optimized
- Divide by Constant HDL Optimized
- Modulo by Constant HDL Optimized

These blocks use hardware-friendly control signals and provide an efficient hardware implementation. These blocks support HDL code generation using HDL Coder.

Reduced HDL resource utilization in fixed-point matrix library blocks

In R2021a, blocks in the **Fixed-Point Designer HDL Optimized > Matrices and Linear Algebra** library that operate on complex inputs have improved algorithms to reduce resource utilization on hardware-constrained target platforms.

Wireless HDL Toolbox Reference Applications: Implement 5G NR MIB recovery for FR2, OFDM interleaver and deinterleaver, and WLAN time and frequency synchronization

The NR HDL MIB Recovery for FR2 (Wireless HDL Toolbox) reference application builds on the NR HDL MIB Recovery (Wireless HDL Toolbox) reference application by adding support for millimeter wave frequencies.

The HDL OFDM Transmitter (Wireless HDL Toolbox) and HDL OFDM Receiver (Wireless HDL Toolbox) reference applications include these enhancements:

- Interleaver and deinterleaver blocks are added to the header and data chain blocks in transmitter and receiver designs to improve burst error performance. For more information about interleaving and deinterleaving, see the HDL Interleaver and Deinterleaver (Wireless HDL Toolbox) example.
- The header field cyclic redundancy check (CRC) polynomial length is increased from 8 bits to 16 bits to improve error detection performance.

- An input valid port is added to the transmitter to access payload data from an external source.

The WLAN HDL Time and Frequency Synchronization (Wireless HDL Toolbox) example shows how to perform packet detection and time and frequency synchronization operations according to wireless local area network (WLAN) standards 802.11a, 802.11b, 802.11g, 802.11n, 802.11ac, and 802.11ax. These operations are essential for proper demodulation and decoding of packet information. This example supports 20, 40, and 80 MHz bandwidth options.

These examples support HDL code generation and are ready for deployment to hardware.

Wireless HDL Toolbox Blocks: Model OFDM Equalizer, NR CRC Encoder, and NR CRC Decoder

The OFDM Equalizer (Wireless HDL Toolbox) block equalizes OFDM data using a channel estimate and noise variance in the frequency domain. The block uses zero forcing (ZF) and minimum mean square error (MMSE) equalization methods.

The NR CRC Encoder (Wireless HDL Toolbox) block generates cyclic redundancy check (CRC) code bits and appends them to input data. The NR CRC Decoder (Wireless HDL Toolbox) block detects errors in input data using CRC bits. The blocks support these six cyclic generator polynomials specified in the 5G NR standard: CRC6, CRC11, CRC16, CRC24A, CRC24B, and CRC24C.

External Memory Modeling Examples: Model and deploy streaming video algorithms that require random access to memory (requires SoC Blockset product)

The Vertical Video Flipping Using External Memory (Vision HDL Toolbox) example shows how to use SoC Blockset blocks to model random-access external memory for streaming vision applications. Then, to generate code for FPGA and processor designs, and deploy the design on a board, the example uses the **SoC Builder** tool.

The Contrast Limited Adaptive Histogram Equalization with External Memory (Vision HDL Toolbox) example shows how to use the SoC Blockset workflow to model frame buffer memory for a CLAHE design.

Multipixel-Multicomponent Video Streaming: Implement Pixel Stream Aligner, Pixel FIFO, and ROI Selector blocks for high-frame-rate color video

The Pixel Stream Aligner (Vision HDL Toolbox), Pixel Stream FIFO (Vision HDL Toolbox), and ROI Selector (Vision HDL Toolbox) blocks now support streams that are both multicomponent and multipixel.

The HDL implementation replicates the algorithm for each pixel and component in parallel.

The blocks support input and output matrices of *NumPixels*-by-*NumComponents* pixels. The **ctrl** ports remain scalar, and the control signals in the `pixelcontrol` bus apply to all pixels in the matrix.

You can simulate System objects with a multipixel streaming interface, but they are not supported for HDL code generation. Use the equivalent blocks to generate HDL code for multipixel algorithms.

Functionality being removed or changed

The HDL Reciprocal block is not recommended

Still runs

The HDL Reciprocal block is not recommended. This block has been moved from the HDL Coder/Math Operations library to the Simulink/Simulink Extras/Additional Math library. Use the `reciprocal` function with the Newton-Raphson method in the Math Function block instead. Existing models continue to work.

Code Generation and Verification

Improvement to HDL code generated for Stateflow Moore Chart blocks

Starting in R2021a, the generated HDL code for Stateflow Moore Chart blocks complies with HDL coding standards. The generated HDL code for Moore Chart blocks is also more structured and readable because it uses separate `process` or `always` statements for updating the state variables and computing the next state.

Stateflow Chart property Initialize Outputs Every Time Chart Wakes Up cleared for HDL code generation

You can now clear the Stateflow Chart property **Initialize Outputs Every Time Chart Wakes Up** for HDL code generation. When the property is disabled, the Moore Chart includes an additional register that holds the previous value instead of passing the initial value to the output when the Chart wakes up.

See Generate HDL for Mealy and Moore Finite State Machines.

HDL block property GenericList for Subsystem blocks with BlackBox architecture

Starting in R2021a, when you generate VHDL code, you can specify whether to flatten the vector ports into a set of scalar ports only at the DUT level instead of the entire model. This optimization speeds up code generation especially for large models that have many vector inputs.

This setting has been renamed as **Scalarize ports**. You can specify this setting when generating HDL code for the MATLAB to HDL workflow and for your Simulink model by using Scalarize ports parameter. See Scalarization of Vector Ports in Generated VHDL Code

Single file for identical Simulink systems (Atomic and Virtual)

Starting in R2021a, HDL Coder can identify logically identical Simulink subsystems and generate a single reusable file to represent the subsystem logic. You can generate this reusable logic regardless of whether the system is marked `'Atomic'` or `'Virtual'`. This file represents and instantiates multiple identical systems. At the command line, you can toggle this feature by setting the `SubsystemReuse` flag to these options:

- `'Atomic and Virtual'`: Causes the code generator to create a single reusable file for multiple identical systems.
- `'Atomic Only'`: The code generator creates reusable code only for atomic systems. This option is the default setting.

Setting `SubsystemReuse` to `'Atomic and Virtual'` reduces artificial algebraic errors and improves the recognition of identical subsystems, irrespective of their topology within the rest of the design. Identification of similar subsystems can help resource sharing.

To set these values to your required setting, in the MATLAB Command Window, enter:

```
hdlset_param('myHDLModel', 'SubsystemReuse', 'Atomic and Virtual')
```

Alternatively, you can set this option from the top-level **HDL Code Generation** pane in the Configuration Parameters dialog box. Under **Global Settings > Coding style**, you can change the **Code reuse** setting to the required option.

The previous commands set the `SubsystemReuse` option for your project. To set this option for only the current code generation session, enter:

```
makehdl(<DUT system>, 'SubsystemReuse', 'Atomic and Virtual')
```

Speed and Area Optimizations

Improved delay balancing support for multiple instances of atomic subsystems

In 2021a, HDL code generation adds support for multiple instances of atomic subsystems that have different input and output configurations, or have different optimization settings across different hierarchical paths.

Improved streaming in presence of scalar expanded constants

In R2021a, HDL code generation has improved streaming optimization in the presence of scalar-expanded constants from Constant blocks. The code generator optimizes the design by reducing the serialization logic for the constants, which reduces the overall area of the system. This optimization also increases opportunities to stream logic, if the constants exist at a lower level of the model hierarchy. The improved streaming works with scalar constants and nonscalar constants that have identical values.

Enhancement to optimization that removes redundant logic for atomic subsystems and model references

In R2021a, when your model contains multiple instances of atomic subsystems, model references, or For Each Subsystem blocks, if these blocks are determined to be active during HDL code generation, then all ports are preserved in the generated code. Components connected upstream to these ports are also considered active.

This optimization is enabled by default. To disable it, either clear the Remove Unused Ports check box in the Configuration Parameters dialog box or set the `DeleteUnusedPorts` property to `off` by using `hdlset_param`.

```
hdlset_param(gcs, 'DeleteUnusedPorts', 'off')
```

See Remove Redundant Logic and Unused Blocks in Generated HDL Code.

Enhancement to sharing optimization for matrix data types

The Serializer and Deserializer blocks now support matrix inputs for HDL code generation. You can therefore use the resource sharing optimization with matrix types instead of having to use Reshape blocks to convert between matrix and vector types.

See Resource Sharing Parameters for Subsystems and Floating-Point IPs.

Adaptive pipelining optimization disabled on model by default

By default, the adaptive pipelining optimization is now disabled on a model. If you decide to use this optimization, you must manually enable it.

In many situations, you can manually insert pipelines in your model and generate efficient HDL code without enabling the adaptive pipelining optimization. However, for certain design patterns, you must

enable adaptive pipelining before generating code. See Design Patterns That Require Adaptive Pipelining.

Compatibility Considerations

In previous releases, the adaptive pipelining optimization was enabled by default. If you now load a pre-R2021a model, the optimization is disabled on the model.

If the adaptive pipelining report for your legacy model showed that the code generator inserted adaptive pipelines in previous releases, manually enable adaptive pipelining when you generate code for that model in R2021a. See Adaptive Pipelining Report.

To reenable the optimization on the model, either select the Adaptive pipelining check box in the Configuration Parameters dialog box, or set the `AdaptivePipelining` property to `on` by using `hdlset_param`.

Generation of target-specific timing databases for critical path estimation

Use the `genhdltdb` function to generate timing databases for a specified target device, target device speed grade, and target tool. To find the critical path in your design, use these custom timing databases. For more information on critical path estimation, see Critical Path Estimation Without Running Synthesis.

IP Core Generation and Hardware Deployment

Updates to supported software

HDL Coder has been tested with:

- Xilinx Vivado Design Suite 2020.1
- Intel Quartus Pro 20.2
- Intel Quartus Pro 20.1

See HDL Language Support and Supported Third-Party Tools and Hardware.

Data Type Support for AXI4 Slave: Map bus data types to AXI4 slave interfaces in IP Core generation

When using the `IP Core Generation` workflow, in the **Target platform interface table**, you can map bus signals at the DUT ports to `AXI4` or `AXI4-Lite` interfaces.

The bus signals workflow includes an **IP Core Generation** report that displays address offsets of AXI4 interface-accessible registers generated for each bus element scalar and vector data type in the **Register Address Mapping** section. The top-level and sub-level bus data types do not have a register offset address. The address mapping for scalar and vector bus elements is not contiguous. For more information, see Custom IP Core Generation.

HDL Workflow Advisor Enhancements

Before generating code, expedite checks on your models by using the optimized HDL Workflow Advisor checks.

The HDL Workflow Advisor has these enhancements:

- Replaced the **Check Global Settings**, **Check Algebraic Loops**, **Check Block Compatibility**, and **Check Sample Times** tasks with the **Check Model Settings** task. The algebraic loop, block compatibility, and sample time checks are now optional. You can perform these checks by using the HDL Code Advisor and additional HDL compatibility checks.
- Merged the **Set Basic Options**, **Set Report Options**, **Set Advanced Options**, **Set Optimization Options**, and **Set Testbench Options** tasks into **Set HDL Options**.
- Enabled HDL DUT port generation for test points directly from the **Set Target Interface** task, which simplifies the mapping of test points to IP core interfaces.

For more information, see HDL Workflow Advisor.

Compatibility Considerations

In R2021a, in the **Prepare Model for HDL Code Generation** checks in the HDL Workflow Advisor these tasks have been removed:

- **Check Global Settings**
- **Check Algebraic Loops**

- **Check Block Compatibility**
- **Check Sample Times**

In R2021a, in the **Set Code Generation Options** tasks in the HDL Workflow Advisor these tasks have been removed:

- **Set Basic Options**
- **Set Report Options**
- **Set Advanced Options**
- **Set Optimization Options**
- **Set Testbench Options**

FPGA Data Capture in HDL Workflow Advisor supports sequential trigger

When you use FPGA Data Capture in HDL Workflow Advisor, you can now provide a sequence of trigger conditions at multiple stages to read data from an FPGA. To enable this feature, specify the maximum number of trigger stages as a value greater than 1 for the **FPGA Data Capture maximum sequence depth** parameter in step 3.2 **Generate RTL Code and IP Core** of HDL Workflow Advisor. For more information on capturing data, see Data Capture Workflow (HDL Verifier).

To use this feature, you must install the HDL Verifier™ Support Package for Xilinx or Intel FPGA boards. To access supported hardware for the HDL Verifier product, see HDL Verifier Supported Hardware (HDL Verifier).

FPGA Data Capture integration with IP Core Generation workflow for generic Xilinx and generic Intel targets

The generic Xilinx platform and the generic Intel platform now support FPGA Data Capture in the IP Core Generation workflow of HDL Workflow Advisor. For more information on the IP Core Generation workflow, see IP Core Generation.

To use this feature, you must install the HDL Verifier Support Package for Xilinx or Intel FPGA boards. To access supported hardware for the HDL Verifier product, see HDL Verifier Supported Hardware (HDL Verifier).

Multirate IP Core Generation: Support AXI4-Stream interface on slower-rate DUT ports

HDL Coder now supports the IP Core generation workflow for designs that map the AXI4-Stream interface to ports that have slower rates than other DUT ports. You can now design fully functional multirate designs for AXI4-Stream interfaces. Your design now supports these options:

- DUT ports running at a rate faster than AXI4-Stream interface ports.
- Resource sharing where the model runs at a single rate and optimizations introduce a faster rate.
- Clock rate pipelining at a rate faster than the design rate.
- AXI4-Stream master and slave channels run at different rates. All AXI4-Stream master ports must be at the same rate. All AXI4-Stream slave ports must be at the same rate.

For more information, see Multirate IP Core Generation.

Complex data type on AXI4-Stream data port

In R2021a, you can now map complex data types as input signals to the AXI4-Stream interface.

High Bandwidth AXI Stream: Generate IP cores that have bit-widths greater than 128 bits on AXI4-Stream data ports

You can now generate an HDL IP Core that has a greater bit-width than 128-bit data on AXI4-Stream interfaces.

Simulink supports fixed-point data types that have word lengths of up to 128 bits. To model your data ports that have word lengths greater than 128 bits, use vector data types. For example, to model a 512-bit data port, use a vector port that has four 128-bit elements. The vector elements are packed into a 512-bit AXI4-Stream data port on the HDL IP core interface.

Generation of HDL IP cores that have greater than 128 bits on external IO interfaces and external ports

When you create your own custom board design that has external IO interfaces and external ports by using the addExternalIOInterface and addExternalPortInterface methods of the hdlcoder.Board class, you can now map DUT ports that have flattened word lengths greater than 128 bits to external IO interfaces and external ports. You can then connect the HDL IP core more easily to other IPs in the reference design that have word lengths greater than 128 bits.

Simulink supports fixed-point data types that have word length of up to 128 bits. To model your data ports that have word lengths greater than 128 bits, use vector data types. For example, to model a 512-bit data port, use a vector port that has four 128-bit elements. The vector elements are packed into a 512-bit AXI4-Stream data port on the HDL IP core interface.

Interface option to customize initial value of AXI4 Master and AXI4 Stream registers

When you run the IP Core Generation workflow and map the DUT ports to AXI4 Master and AXI4-Stream master interfaces, you can customize the initial value of the default read and write base addresses for the AXI4 Master and default frame length for the AXI4-Stream master.

In the **Set Target Interface** task, when you map the DUT port to AXI4 Master interfaces, click the **Options** button that appears on the **Interface Options** column of the target platform interface table. Specify the **DefaultReadBaseAddress** or **DefaultWriteBaseAddress**. If you have already specified the default base read and write addresses in the reference design file you cannot specify the addresses in the interface option.

In the **Set Target Interface** task, when you map the DUT port to AXI4-Stream master interfaces, click the **Options** button that appears on the **Interface Options** column of the target platform interface table. Specify the **DefaultFrameLength**. If you have already mapped the TLAST signal for the AXI4-Stream master the **DefaultFrameLength** option is disabled.

This setting is saved on the DUT port in the model as the HDL block property **IOInterfaceOptions**. For example, if you map a DUT input port to AXI4-Stream interface, set **DefaultFrameLength** to

`512`, and then generate the IP core. The **IOInterfaceOptions** property of that input port is saved with the value `{'DefaultFrameLength','512'}`.

See Initial Value of AXI4 Slave Registers.

Simscape Hardware-in-the-Loop Workflow

Partitioning solver: Use partitioning solver to generate HDL code from nonlinear models

When you use nonlinear Simscape models, you can now use the partitioning solver to generate the HDL implementation model and then generate HDL code and deploy the code to Speedgoat® Simulink-Programmable I/O modules.

The partitioning solver converts the entire system of equations for the Simscape network into several smaller sets of switched linear equations that are connected through nonlinear functions. By using this solver, you can run the Simscape HDL Workflow Advisor without having to remove nonlinear blocks in your model or replacing them with the corresponding Simulink blocks.

To use the partitioning solver, on the Solver Configuration (Simscape) block, set the **Solver type** block parameter to `partitioning` and then run the Simscape HDL Workflow Advisor.

Optimal value of oversampling factor automatically set on HDL implementation model

Previously, when you ran the Simscape HDL Workflow Advisor, the **Oversampling factor** was set to `60` on the HDL implementation model. When you used this value to generate HDL code from the model, delay balancing was sometimes unsuccessful, and you had to increase this value to accommodate the latency introduced by the floating-point operations. In some cases, to run your design at the maximum achievable target frequency on the target design, you had to further adjust the oversampling factor.

Starting in R2021a, the Simscape HDL Workflow automatically sets an optimal value for the **Oversampling factor**. You can then generate HDL code from the model and run your design at the maximum achievable target frequency with minimal or no modifications to this value.

R2020b

Version: 3.17

New Features

Bug Fixes

Compatibility Considerations

Model and Architecture Design

Half-Precision Native Floating Point: Generate target-independent synthesizable RTL code from half-precision floating-point models

In R2020b, if you have half-precision data types in your Simulink model, you can use HDL Coder native floating-point support to generate target-independent HDL code. You can deploy the generated code on any generic ASIC or FPGA platform. For applications that require smaller dynamic range, you can use `half` types without having to convert your design to use fixed-point types. Using `half` types consumes much less memory, has lower latency, and saves FPGA resources. See Getting Started with HDL Coder Native Floating-Point Support.

HDL Coder supports basic math operators and various kinds of Delay blocks that have `half` types for HDL code generation. See Simulink Blocks Supported with Native Floating-Point and Latency Values of Floating Point Operators.

For an example, see Floating Point Support: Field-Oriented Control Algorithm.

HDL code generation for lookup tables that have floating-point types

Previously, for the 1-D Lookup Table, 2-D Lookup Table, and n-D Lookup Table blocks, you could use floating-point types for the table data and output.

In R2020b, you can also use floating-point types for the inputs and breakpoints, and generate HDL code for the blocks in `Native Floating Point` mode. Floating-point support for the blocks is limited to `single` and `double` data types for HDL code generation. **Breakpoints specification** parameter supports `Evenly spaced` and `Explicitly specified`.

To learn about HDL block properties you can specify for the blocks, see HDL Code Generation (Simulink).

HDL Code Advisor check for blocks that introduce latency with fixed-point types

Previously, the HDL Code Advisor check **Check for blocks with nonzero output latency** identified blocks that had nonzero output latency with floating-point signals in `Native Floating Point` mode.

In R2020b, the check is moved to the **Check for blocks and block settings** folder. When you run the check, it identifies blocks that introduce latency in the generated HDL code with both fixed-point and floating-point types. You can then add the appropriate number of delays adjacent to the blocks in the original model, and therefore simulate the model with latency. The code generator absorbs those delays and does not introduce additional latency in the generated HDL code.

See Check for blocks that have nonzero output latency.

Automatically package protected models with their dependencies

When you create a protected model, you can now automatically package it with its dependencies and a harness model in a project archive. When the recipient extracts the contents of the project archive

and opens the harness model, they should be able to simulate the protected model without needing to define missing variables or objects. Before sharing the project archive, check whether the project contains all of the necessary supporting files, and update the harness model as needed.

In the Create Protected Model dialog box, set **Contents** to `Protected model (.slxp) and dependencies in a project`. For **Name of project archive (.mlproj)**, use the default name or specify a name. The project inside the project archive uses the same name.

Alternatively, use the `Simulink.ModelReference.protect` function with the comma-separated pair consisting of `'Project'` and `true`. To specify a project name, also use the comma-separated pair consisting of `'ProjectName'` and the desired name specified as a character vector. If you do not specify a project name, the function uses the default name.

For more information, see Package and Share Protected Models.

Block Enhancements

Optimized Square Root: Generate high-frequency fixed-point HDL implementation of square root operations

When you use the `SqrtFunction` architecture of the Sqrt block with fixed-point data types, you can now use the **LatencyStrategy** and **CustomLatency** settings in the HDL Block Properties dialog box to specify whether to use zero, maximum, or a custom latency value between zero and maximum value. You can use the custom latency implementation to choose from a range of frequency values.

Depending on the **UseMultiplier** and **LatencyStrategy** settings, you can use a pipelined multiplication algorithm or a shift and add algorithm to compute the square root. See Implement Control Signals Based Mathematical Functions Using HDL Coder.

Compatibility Considerations

The **SqrtBitSet** architecture of the Sqrt block is no longer available in R2020b. If you load a pre-R2020b model that you saved with `SqrtBitSet` as the architecture, the HDL architecture is now saved as `SqrtFunction`. For fixed-point types, the `SqrtFunction` architecture implements the same functionality as the `SqrtBitSet` algorithm.

Custom latency for math and trigonometric blocks with fixed-point types

You can now specify a custom latency value for these blocks with fixed-point types and generate HDL code. In the HDL Block Properties dialog box for these blocks, you can use the **LatencyStrategy** and **CustomLatency** settings to specify whether to use zero, maximum, or a custom latency value between zero and maximum value. You can use the custom latency implementation to choose from a range of frequency values when targeting the generated code onto an FPGA device.

- Divide and Reciprocal blocks that have `ShiftAdd` as the HDL architecture.
- Sqrt block that has `SqrtFunction` as the HDL architecture. See “Optimized Square Root: Generate high-frequency fixed-point HDL implementation of square root operations” on page 3-4.
- Trigonometric Function block that has **Function** set to `sin`, `cos`, `sincos`, `cos+jsin`, or `atan2` and **Approximation method** as `CORDIC`.

See Implement Control Signals Based Mathematical Functions Using HDL Coder.

Compatibility Considerations

Starting in R2020b, in the HDL Block Properties dialog box, the HDL architecture has been renamed from `SinCosCordic` to `Cordic`. The **UsePipelinedKernel** setting is no longer available in the HDL Block Properties dialog box. If you load a pre-R2020b model with the **UsePipelinedKernel** property saved on the model, you can access the property value by using `hdlget_param`.

Modulo option for HDL Counter block

In the Block Parameters dialog box of the HDL Counter block, you can select a `Modulo` option for the **Counter type** parameter. In this mode, depending on the count direction, the counter counts up or

down from the **count from value** to the **count to value**, and then wraps back to a value that is determined by a wrapping step value.

This mode of the HDL Counter differs from the `Count limited` because the count value does not exceed the **count to value**. To learn more, see Count Limited and Modulo Operation Modes.

HDL code generation for Scoped tag visibility for Goto block

In R2020b, you can generate HDL code for the Goto block that has the **Tag visibility** block parameter specified as `Scoped`. See Required HDL Settings for Goto and From Blocks.

Product block enhancements for HDL code generation

You can now specify `/*` for **Number of inputs** block parameter and generate HDL code for the Product block with both fixed point and floating point types. In this mode, the second input is divided by the first input to calculate the result.

5G NR HDL MIB Recovery Reference Application: Implement 5G NR MIB recovery subsystem on FPGA or ASIC

The NR HDL MIB Recovery (Wireless HDL Toolbox) reference application in Wireless HDL Toolbox™ builds on the NR HDL Cell Search (Wireless HDL Toolbox) reference application by decoding the broadcast channel and recovering the master information block (MIB). This design supports HDL code generation with HDL Coder and is ready for deployment to hardware.

OFDM Transmitter and Receiver Reference Applications: Implement custom OFDM wireless communication system on FPGA or ASIC

The HDL OFDM Transmitter (Wireless HDL Toolbox) and HDL OFDM Receiver (Wireless HDL Toolbox) reference applications in Wireless HDL Toolbox implement an orthogonal frequency-division multiplexing (OFDM) based wireless communication system designed using Simulink blocks. The design supports HDL code generation with HDL Coder and is ready for deployment to hardware.

The HDL OFDM MATLAB References (Wireless HDL Toolbox) shows how to model a wireless communication hardware algorithm in MATLAB as a step toward developing a Simulink HDL-friendly model. Use this MATLAB reference to verify the Simulink reference application model.

HDL-optimized FIR Decimation block and System object: Downsample signals using a FIR decimation filter with a hardware-friendly interface and architecture

The FIR Decimation HDL Optimized block in DSP System Toolbox™ downsamples signals using a transposed or systolic filter architecture. The block provides an efficient hardware implementation and uses hardware-friendly control signals.

This algorithm is also available with the `dsp.HDLFIRDecimation` System object™ in DSP System Toolbox.

Gigasample-per-second (GSPS) CIC Decimation HDL-Optimized Block: Increase throughput of CIC decimation by using frame-based input

You can now generate frame-based waveforms from the CIC Decimation HDL Optimized block in DSP System Toolbox. The block accepts and returns a column vector of elements that represent samples in time. The input vector can contain up to 64 samples. When you use frame-based input, you must use a fixed decimation factor.

This capability increases throughput in hardware designs. For a list of all blocks that support frame-based input and output for HDL code generation, see High Throughput HDL Algorithms (DSP System Toolbox).

This feature is also available with the `dsp.HDLCICDecimation` System object in DSP System Toolbox.

Corner Detector Block and System Object: Detect features using Harris algorithm

The Corner Detector block in Vision HDL Toolbox now provides a choice between the FAST algorithm and the Harris and Stephens interconnecting edges algorithm.

Region of Interest (ROI) Resource Sharing: Share hardware resources and streaming control signals between vertically aligned regions

The ROI Selector block in Vision HDL Toolbox provides an option to share hardware resources when selecting vertically aligned regions. Regions in the same column share the same `pixelcontrol` bus output.

Select the **Reuse output ports for vertically aligned regions** checkbox, and provide a set of regions that are aligned in columns and do not overlap vertically within each column. You can specify up to 1024 regions per column. To divide a frame into tiled regions that are compatible with vertical reuse, use the `visionhdlframetoregions` function.

Blob Analysis Example: Detect and label connected components in streaming video

The Blob Analysis (Vision HDL Toolbox) example in Vision HDL Toolbox shows how to implement a single-pass 8-way connected component labeling algorithm, and perform blob analysis to give the centroid, bounding box, and area of each blob. The model supports up to 1080p@60fps video.

HDL Minimum Resource FFT and HDL Streaming FFT blocks have been removed

The HDL Minimum Resource FFT and HDL Streaming FFT blocks have been removed. Use the FFT HDL Optimized block instead.

- When replacing the HDL Minimum Resource FFT block, set the **Architecture** parameter of the FFT HDL Optimized block to `Burst Radix 2`.
- When replacing the HDL Streaming FFT block, set the **Architecture** parameter of the FFT HDL Optimized block to `Streaming Radix 2^2`.

For more information, see Implement FFT for FPGA Using FFT HDL Optimized Block (DSP System Toolbox).

Code Generation and Verification

Option to scalarize vector ports only at DUT level in VHDL code

Starting in R2020b, when you generate VHDL code, you can specify whether to flatten the vector ports into a set of scalar ports only at the DUT level instead of the entire model. This optimization speeds up code generation especially for large models that have many vector inputs.

This setting has been renamed as **Scalarize ports**. You can specify this setting when generating HDL code for the MATLAB to HDL workflow and for your Simulink model by using Scalarize ports parameter. See Scalarization of Vector Ports in Generated VHDL Code

See also "IP core generation workflow for scalarization of vector ports only at DUT level in VHDL code" on page 3-12.

HDL code generation for models that have comment through blocks

You can now generate HDL code for models containing blocks that are comment through. When you right-click a block and select **Comment Through**, the block is excluded from simulation and HDL code generation. The signals are passed through from the input of the block to the output. Use this capability for debugging your model and for identifying blocks that are not supported for HDL code generation.

See Terminate Unconnected Block Outputs and Usage of Commenting Blocks.

HDL code generation for models that have Subsystem Reference blocks

You can now generate HDL code for models containing Subsystem Reference blocks. By using Subsystem Reference blocks, you can save the contents of a subsystem in a separate `SLX` file and reuse it multiple times. To convert a Subsystem block to a Subsystem Reference, select that Subsystem, and in the Simulink Toolstrip, on the **Subsystem Block** tab, select **Convert** > **Convert to Referenced Subsystem**. See Subsystem Reference (Simulink).

By default, HDL Coder generates separate HDL files for referenced subsystems. To generate a single HDL file, convert the **Subsystem Reference** block to an atomic subsystem and set `DefaultParameterBehavior` property as `Inlined`.

Enhancement to HDL code generation for nontop DUT

In R2020b, when you generate HDL code for a nontop DUT subsystem or a DUT that is not at the top level of the model, the code generator does not covert the subsystem to a model reference. You can generate HDL code for the nontop DUT without restrictions that previously applied when converting to a model reference.

For example, you can now use masked subsystems where the mask parameter values are initialized as the nontop DUT. You can also use Bus Element ports inside the nontop DUT, and then generate HDL code.

HDL code generation for nonboolean inputs at control ports

Previously, to generate HDL code for blocks that have control ports such as a Delay block with an external enable port, you used `boolean` or `ufix1` types as inputs to the ports. Starting in R2020b, in addition to `boolean` and `ufix1`, you can use other data types for the control ports of these blocks and generate HDL code:

- Delay block with an external reset or enable port. That is, Enabled Delay, Resettable Delay, and Enabled Resettable Delay blocks.
- Unit Delay Enabled Synchronous, Unit Delay Resettable Synchronous, and Unit Delay Enabled Resettable Synchronous blocks.
- Triggered Subsystem, Enabled Subsystem, Enabled Synchronous Subsystem, and Resettable Synchronous Subsystem blocks.

HDL code generation for absolute time temporal logic in Stateflow

Starting in R2020b, you can use absolute-time temporal logic in Stateflow charts for HDL code generation. Support of absolute-time temporal logic is limited to charts that execute with a fixed time period, and does not include charts that are conditionally executed or triggered. Time periods for absolute-time temporal logic operators are defined based on the simulation time of the chart in your Simulink model. For more information, see Control Chart Execution by Using Temporal Logic (Stateflow).

Default HDL simulation command vsim -novopt has changed to vsim -voptargs=+acc

The default value of the `HDLSimCmd` property is now `'-voptargs=+acc %s.%s\n'`. See Simulation command.

Compatibility Considerations

Prior to R2020b, the default HDL simulation command was `vsim -novopt %s.%s\n`. Mentor Graphics® ModelSim® versions prior to 10.7 support the former syntax. If you use a more recent ModelSim version, you must use the `-voptargs=+acc` syntax.

UseMatrixTypesInHDL property not recommended

The **UseMatrixTypesinHDL** HDL block property on the MATLAB Function block and Stateflow Chart is not recommended for use. In R2020b, this property no longer appears in the HDL Block Properties dialog box for these blocks. The **UseMatrixTypesinHDL** property does not affect the HDL code generation behavior.

Compatibility Considerations

If you now load a pre-R2020b model that has the **UseMatrixTypesinHDL** property saved on the model, you can still access this property from the command line by using `hdlget_param`.

Speed and Area Optimizations

Option to control removal of unused ports in generated HDL code

In R2020b, when you generate HDL code, you can specify whether to remove unused ports in the model that are not at the top level.

This optimization is enabled by default. To disable it, either clear the Remove Unused Ports check box in the Configuration Parameters dialog box or set the `DeleteUnusedPorts` property to `off` by using `hdlset_param`.

```
hdlset_param(gcs, 'DeleteUnusedPorts', 'off')
```

See Remove Redundant Logic and Unused Blocks in Generated HDL Code.

Hierarchy flattening report

When you select the **Generate optimization report** check box and set the HDL block property **FlattenHierarchy** to `on` for a subsystem in your model, HDL Coder now generates a hierarchy flattening report. The report displays subsystems in your model that have **FlattenHierarchy** set to `on` and `off`, hierarchy flattening status, and the HDL files that are inlined.

See Hierarchy Flattening Report.

Optimization enhancements for Sum of Elements and MinMax blocks

In R2020b, when you enable **FlattenHierarchy** for the MinMax block, it generates a single HDL file. In addition, you can now use various optimizations for the Sum of Elements and Product of Elements blocks that use `Linear` architecture because the blocks are expanded into multiple, simpler blocks during code generation.

IP Core Generation and Hardware Deployment

Rapid prototyping of HDL IP core by using software interface script

When you run the `IP Core Generation` workflow for your Simulink model, you can now rapidly prototype the HDL IP core by using a generated MATLAB software interface script.

In R2020b, the **Generate software interface model** task has been renamed as **Generate software interface** task. To generate the script, on the **Generate software interface** task, select the **Generate MATLAB software interface script** check box, and then run this task. You can also generate a software interface script by running the HDL Workflow at the command line by using `RunTaskGenerateSoftwareInterface`, and properties `GenerateSoftwareInterfaceModel` and `GenerateSoftwareInterfaceScipt`.

For standalone FPGA boards that do not have an embedded ARM processor, the **Generate software interface model** task was not available in the HDL Workflow Advisor. In R2020b, when you enable the reference design parameter **Insert JTAG MATLAB as AXI Master** (requires HDL Verifier license), you can still generate a software interface script to prototype the generated HDL IP core. The script uses the MATLAB AXI Master to control the AXI4 slave registers.

Compatibility Considerations

In R2020b, you can import and run an HDL Workflow script that used the `RunTaskGenerateSoftwareInterfaceModel` setting. If `RunTaskGenerateSoftwareInterfaceModel` was saved as `true`, then `RunTaskGenerateSoftwareInterface`, and properties `GenerateSoftwareInterfaceModel` and `GenerateSoftwareInterfaceScipt` are set to `true`. If `RunTaskGenerateSoftwareInterfaceModel` was saved as `false`, then `RunTaskGenerateSoftwareInterface` and `GenerateSoftwareInterfaceModel` are set to `false`, and the task is skipped.

See Generate Software Interface to Probe and Rapidly Prototype the HDL IP Core and Create Software Interface Script to Control and Rapidly Prototype HDL IP Core.

Interface option to customize initial value of AXI4 slave registers

In R2020b, when you run the `IP Core Generation` workflow and map the DUT ports to AXI4 slave interfaces, you can customize the initial value for the AXI4 slave registers.

In the **Set Target Interface** task, when you map the DUT port to AXI4 slave interfaces, click the **Options** button that appears on a new **Interface Options** column of the target platform interface table, and then specify the **RegisterInitialValue**. When you run the workflow to the **Generate RTL Code and IP Core** task, the IP core report displays the **RegisterInitialValue** that you specified.

This setting is saved on the DUT port in the model as the HDL block property **IOInterfaceOptions**. For example, if you map a DUT input port to AXI4-Lite interface, set **RegisterInitialValue** to `5`, and then generate the IP core, the **IOInterfaceOptions** property of that input port is saved with the value `{'RegisterInitialValue','5'}`.

See Initial Value of AXI4 Slave Registers.

Generation of HDL IP cores that have greater than 128 bits on internal IO interface

When you create your own custom reference design that has internal IO interfaces by using the `addInternalIOInterface` method of the `hdlcoder.ReferenceDesign` class, you can now map DUT ports that have flattened word lengths greater than `128` bits to internal IO interfaces. You can then connect the HDL IP core more easily with other IPs in the reference design that have word lengths greater than `128` bits.

Simulink supports fixed-point data types that have word length of up to `128` bits. To model your DUT ports that have word lengths greater than `128` bits, use vector data types. For example, to model a `512`-bit Data port, use a vector port with four `128`-bit scalar ports. You can then map these DUT Data ports to internal input and output interfaces in the Target platform interface table, generate the HDL IP core, and integrate the IP core into your Vivado or Qsys reference designs.

IP core generation workflow for scalarization of vector ports only at DUT level in VHDL code

When you run the `IP Core Generation` workflow with `VHDL` as the **Language**, to speed up code generation for models with vector ports, you can now flatten the vector ports into a set of scalar ports only at the external interface of the DUT instead of the entire model.

For your Simulink model, in the HDL Workflow Advisor, on the **HDL Code Generation** > **Set Code Generation Options** > **Set Advanced Options** > **Ports** tab, set Scalarize ports to `DUT level`. To learn how to specify this setting for your MATLAB algorithm, see “Option to scalarize vector ports only at DUT level in VHDL code” on page 3-8 and Scalarization of Vector Ports in Generated VHDL Code.

Compatibility Considerations

If you now load a pre-R2020b model that had **Scalarize ports** set to `on`, when you run the **Generate RTL Code and IP Core** task, the Advisor generates a warning that indicates using `DUT level` setting for faster code generation.

Intel Quartus Pro SoC Targeting: Generate generic HDL IP core or integrate IP core into Intel reference designs

You can now specify `Intel Quartus Pro` as the **Synthesis tool**, and then select `IP Core Generation` as the **Target workflow** to generate a generic HDL IP core for the Intel platform or integrate the IP core into Intel Quartus Pro reference designs.

Before you specify the **Synthesis tool**, set up the tool path by using the `hdlsetuptoolpath` function. HDL Coder supports these family of devices for Intel Quartus Pro:

- `Arria 10`
- `Cyclone 10 GX`
- `Stratix 10`

See also HDL Language Support and Supported Third-Party Tools and Hardware.

Arria 10 SoC AXI4 Slave reference design

In R2020b, you can target reference designs that are based on the Arria® 10 SoC board when you specify Altera Quartus-II or Intel Quartus Pro as the **Synthesis tool**. Depending on the **Synthesis tool**, you can select `Altera Arria 10 SoC development kit` or the `Intel Arria 10 SoC development kit` as the **Target platform**, and then target the `Default system` or the `Default system with External DDR4 Memory Access` reference designs.

To use the Early I/O release capability on the Arria 10 SoC, you can split the bitstream into core and peripheral RBF files. To generate a split bitstream, make sure that the new reference design parameter `GenerateSplitBistream` of the `hdlcoder.ReferenceDesign` class is set to `true`.

See Getting Started with Targeting Intel Quartus Pro based Devices.

Speedgoat I/O Modules IO331 and IO333 being removed

Speedgoat I/O modules `Speedgoat IO331`, its variant `Speedgoat IO331-6`, and `Speedgoat IO333` that you used with the `Simulink Real-Time FPGA I/O` workflow are no longer supported in R2020b.

Compatibility Considerations

In R2020b, if you load a pre-R2020b model that was saved by using the target platform `Speedgoat IO331`, `Speedgoat IO331-6`, or `Speedgoat IO333`, and then open the HDL Workflow Advisor, HDL Coder generates a warning. Before you run the `Simulink Real-Time FPGA I/O` workflow, install the Speedgoat I/O Blockset and the Speedgoat HDL Coder Integration Packages.

See also Speedgoat - HDL Coder Integration Packages, Speedgoat FPGA Support with HDL Workflow Advisor, and Xilinx HDL Support with Speedgoat IO Modules.

Audio filter reference application for Intel SoC device

HDL Coder provides two examples in R2020b that target the Intel SoC device.

- Authoring a Reference Design for Audio System on Intel board shows how to author an audio reference design on an Intel SoC device.
- Running an Audio Filter on Live Audio Input using Intel Board extends the audio filter algorithm in Running an Audio Filter on Live Audio Input Using a Zynq Board to target the Intel Arrow® SoC development kit.

Updates to supported software

HDL Coder has been tested with:

- Xilinx Vivado Design Suite 2019.2
- Intel Quartus Pro 19.4
- Microsemi Libero SoC 12.0

See HDL Language Support and Supported Third-Party Tools and Hardware.

Simscape Hardware-in-the-Loop Workflow

Automatic replacement of Simscape subsystem with state-space implementation

In R2020b, the Simscape HDL Workflow Advisor automatically replaces the Simscape subsystem with the state-space implementation in the HDL implementation model. You do not have to modify the HDL implementation model, and can directly generate HDL code for the model and then deploy the code onto Speedgoat FPGA I/O modules.

In addition, when you select the **Generate validation logic for the implementation model** check box, the Advisor generates the HDL implementation model and a state-space validation model.

See Modeling Guidelines for Simscape Subsystem Replacement.

Automatic setting of number of solver iterations in Simscape HDL Workflow Advisor

Previously, when you ran the Simscape HDL Workflow Advisor, in the **Generate implementation model** task, the **Number of solver iterations** was specified as `5` for switched linear models. When you generated the HDL implementation model, in some cases, you had to iterate multiple times to get the optimal number of solver iterations.

In R2020b, depending on your Simscape model, the Advisor automatically determines an optimal number of solver iterations that causes the model simulation to converge and avoids exceeding the threshold value for real-time deployment.

See Using Number of Solver Iterations.

Mapping of state-space parameters to RAM in HDL implementation model

For large Simscape models, the generated HDL implementation model can have large state-space parameters. These state-space parameter matrices consumed a large number of FPGA lookup table resources on the FPGA, and might cause the design to not fit on the target FPGA device.

To save lookup table resources, you can now map the state-space parameter matrices to Block RAM resources on the FPGA. In the **Generate implementation model** task, the **Map state space parameters to RAMs** setting is specified as `Auto`, which maps state-space parameters to RAMs.

See Generate Optimized HDL Implementation Model from Simscape.

Duplicate configurations removed in generated HDL implementation model

Starting in R2020b, the Simscape HDL Workflow Advisor removes duplicate configurations when computing the state-space equations. Duplicate configurations are configurations that have the same state-space matrices. The duplicate configuration removal saves area on the target FPGA device because it reduces the number of state-space coefficients that has to be stored on the FPGA.

For example, the Swiss rectifier example model, `sschdlexSwissRectifierExample`, previously reported `113` configurations. `23` of these configurations are duplicate, which is now removed when extracting the state-space equations, resulting in lower resource consumption on the FPGA.

R2020a

Version: 3.16

New Features

Bug Fixes

Compatibility Considerations

Model and Architecture Design

Additional HDL modeling guidelines added to documentation

In R2020a, these HDL modeling guidelines have been added to the documentation.

- Guidelines for Using Delays and Goto and From Blocks for HDL Code Generation
- Modeling Efficient Multiplication and Division Operations for FPGA Targeting
- Using Persistent Variables and fi Objects Inside MATLAB Function Blocks for HDL Code Generation
- Guidelines for HDL Code Generation Using Stateflow Charts

See Guidelines for Supported Blocks and Data Type Settings.

Functionality being removed or changed

Starting in R2020a, the HDL Model Checker is being renamed to HDL Code Advisor. This is a name change only. To open the HDL Code Advisor, you now use the `hdlcodeadvisor` function. The functionality remains the same and the original function name, `hdlmodelchecker`, will continue to work.

See Getting Started with the HDL Code Advisor.

Compatibility Considerations

Although the function `hdlmodelchecker` will continue to work, if your code uses `hdlmodelchecker`, consider updating the code to use `hdlcodeadvisor`.

Block Enhancements

Inverse of streaming matrix input using Gauss-Jordan elimination method

HDL Coder now supports the Gauss-Jordan elimination algorithm for performing a streaming matrix inverse computation. You can use square matrices, and use both `single` and `double` data types when computing the matrix inverse by using Gauss-Jordan elimination.

Previously, you computed the inverse of a streaming matrix input by using the Cholesky decomposition method. In R2020a, you use the **AlgorithmType** HDL block property of the MatrixInverse block to specify whether you want to compute the inverse of a streaming input matrix by using `GaussJordanElimination` or `CholeskyDecomposition`. The default **AlgorithmType** is `GaussJordanElimination`.

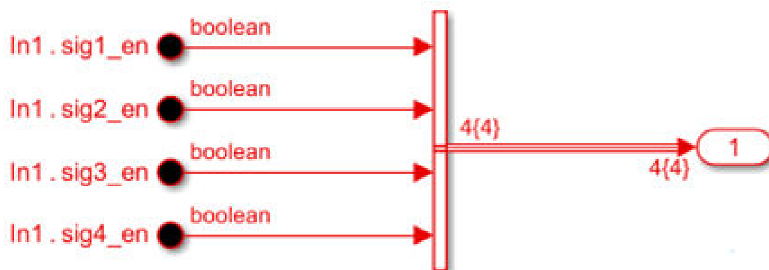
For an example, see HDL Code Generation for Streaming Matrix Inverse System Object.

Improvement to readability of bus element port names in HDL code

Previously, when you generated HDL code for models that contained bus element ports, the ports were named according to the format `portName_Inport_portNumber_signalName` or `portName_Outport_portNumber_signalName`.

Starting in R2020a, HDL Coder uses more readable and concise names in the generated code for bus element ports. The ports now use the format `portName_signalName` irrespective of whether the port is an input or output bus element port.

For example, consider this model that uses Bus Element In ports.



The generated VHDL code shows the entity declaration for the bus element ports as illustrated in the table.

<table>
<tr><th>Before R2020a</th><th>In R2020a</th></tr>
<tr><td><pre>
ENTITY Subsystem1 IS
  PORT(In1_Inport_1_sig1_en   :   IN   std_logic;
        In1_Inport_1_sig2_en   :   IN   std_logic;
        In1_Inport_1_sig3_en   :   IN   std_logic;
        In1_Inport_1_sig4_en   :   IN   std_logic;
        Out1_sig1_en   :   OUT   std_logic;
        Out1_sig2_en   :   OUT   std_logic;
        Out1_sig3_en   :   OUT   std_logic;
        Out1_sig4_en   :   OUT   std_logic
        );
END Subsystem1;
</pre></td><td><pre>
ENTITY Subsystem IS
  PORT(In1_sig1_en   :   IN   s
        In1_sig2_en   :   IN   s
        In1_sig3_en   :   IN   s
        In1_sig4_en   :   IN   s
        Out1_sig1_en   :   OUT   s
        Out1_sig2_en   :   OUT   s
        Out1_sig3_en   :   OUT   s
        Out1_sig4_en   :   OUT   s
        );
END Subsystem;
</pre></td></tr>
</table>

See Use Bus Signals to Improve Readability of Model and Generate HDL Code.

New Fixed-Point Designer Simulink block library

Fixed-Point Designer now has a Simulink block library for math operations and matrix operations. These blocks use hardware-friendly control signals and provide an efficient hardware implementation. These blocks support HDL code generation using HDL Coder.

Math Operations

Use the Hyperbolic Tangent HDL Optimized block to compute the CORDIC-based hyperbolic tangent.

Use the Normalized Reciprocal HDL Optimized block to compute the normalized reciprocal of an input value.

Matrix Operations

The Real Burst QR Decomposition and Complex Burst QR Decomposition blocks use Givens rotations to compute the QR decomposition of an input matrix.

Use the Real Burst Matrix Solve Using QR Decomposition or Complex Burst Matrix Solve Using QR Decomposition blocks to compute the value of x in the equation $Ax = b$.

The Real Burst Q-less QR Decomposition and Complex Burst Q-less QR Decomposition blocks use Givens rotations to compute the R factor of the QR decomposition without computing Q.

Use the Real Burst Matrix Solve Using Q-less QR Decomposition or Complex Burst Matrix Solve Using Q-less QR Decomposition blocks to compute the value of x in the equation $A'Ax = b$.

LTE HDL Toolbox name change to Wireless HDL Toolbox

The Wireless HDL Toolbox name reflects the expansion of the toolbox to include algorithms for 5G New Radio (NR) and general communications designs along with LTE algorithms.

This table shows the updated library hierarchy.

Old Libraries	Updated Libraries
LTE HDL Toolbox Error Detection and Correction General Communications I/O Interfaces Modulation Utilities	Wireless HDL Toolbox Error Detection and Correction I/O Interfaces Modulation Utilities

All blocks from the LTE HDL Toolbox™ libraries are available in the Wireless HDL Toolbox libraries. The blocks formerly in the **General Communications** library are now split between the **Error Detection and Correction** and **Modulation** libraries. This table shows the updated location of each of these blocks.

Block	Updated Library
Convolutional Encoder	Error Detection and Correction
Depuncturer	Error Detection and Correction
OFDM Demodulator	Modulation
Puncturer	Error Detection and Correction
Viterbi Decoder	Error Detection and Correction

5G NR Signal Synchronization Reference Application: Use primary and secondary synchronization signals (PSS and SSS) to detect connection to valid cell

The NR HDL Cell Search (Wireless HDL Toolbox) reference application, in Wireless HDL Toolbox, performs PSS and SSS signal synchronization as per the 5G NR standard. This design supports HDL code generation with HDL Coder and is ready for deployment to hardware.

There is also a NR HDL Cell Search MATLAB Reference (Wireless HDL Toolbox) example that shows how to model the 5G NR cell search hardware algorithm in MATLAB as a step towards developing a Simulink HDL implementation. This MATLAB reference is then used to verify the Simulink reference application model.

5G NR Polar Encoder and Decoder, 5G NR LDPC Encoder and Decoder blocks

These blocks in Wireless HDL Toolbox provide hardware-friendly implementations that comply with the 5G NR standard.

- NR Polar Encoder and NR Polar Decoder
- NR LDPC Decoder and NR LDPC Encoder

OFDM Modulator, OFDM Channel Estimator, and RS Decoder blocks

These blocks in Wireless HDL Toolbox provide hardware-friendly implementations that support standard and custom communication protocols.

- OFDM Modulator

- OFDM Channel Estimator
- RS Decoder

Variable CIC Decimation Factor: Specify decimation factor as an input to the CIC Decimation HDL Optimized block

You can specify the decimation factor for the CIC Decimation HDL Optimized block as an input port. You can also now optionally enable automatic gain correction.

These features are also available when using the `dsp.HDLCICDecimation` System object.

To use this block and object you must have DSP System Toolbox installed.

Gigasample-per-second (GSPS) NCO: Generate frame-based output from HDL-optimized NCO for high speed applications (requires HDL Coder for code generation)

You can now generate frame-based waveforms from the NCO HDL Optimized block. The block returns a vector where each element represents a sample in time. Set the **Samples per frame** parameter to the desired output vector size.

This capability increases throughput in hardware designs. For a list of all blocks that support frame-based input and output for HDL code generation, see High Throughput HDL Algorithms (DSP System Toolbox).

This feature is also available when using the `dsp.HDLNCO` System object. Set the `SamplesPerFrame` property to the desired output vector size.

To use this block and object you must have DSP System Toolbox installed.

Corner Detector Block and System Object: Detect features using FAST algorithm

The Corner Detector block, in Vision HDL Toolbox, detects corners using the features-from-accelerated-segment test (FAST) algorithm. You can specify a minimum contrast threshold as a parameter or port and select from three metrics that determine a corner: 5 out of 8, 7 out of 12, or 9 out of 16 pixels. These metrics specify how many pixels in a circle of pixels must meet the minimum contrast for the center pixel to be considered a corner.

Line Buffer with No Padding: Specify option to not add padding for blocks that use line buffer memory

You can now configure the Line Buffer block and blocks that use an internal line buffer to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the **Padding method** parameter to `None`.

This change affects these blocks:

- Line Buffer
- Bilateral Filter
- Corner Detector
- Edge Detector
- Image Filter
- Median Filter
- Binary morphology blocks: Closing, Dilation, Erosion, and Opening

To use these blocks you must have Vision HDL Toolbox installed.

Code Generation and Verification

Obfuscated HDL Output: Generate plain-text HDL code with randomized identifier names

To share HDL code with a third party without revealing the intellectual property, you can now generate obfuscated HDL code from Simulink models. Obfuscation reduces readability of the code. The generated HDL code does not have any comments, newlines, or spaces, and replaces identifier names with random names.

When you enable HDL code obfuscation and generates code, HDL Coder produces a Code Obfuscation Report. The report displays the status of HDL code obfuscation and whether the model uses Configuration Parameters that are incompatible with code obfuscation.

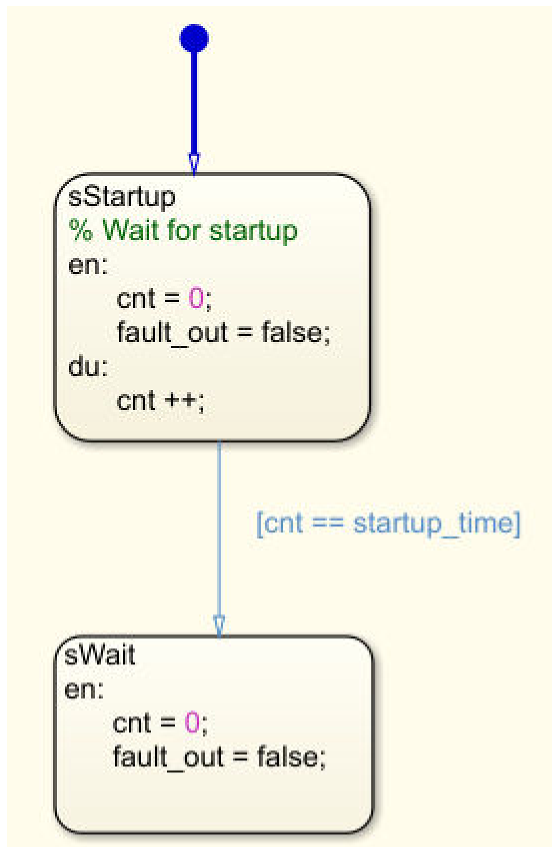
See Obfuscate Generated HDL Code from Simulink Models.

Improvements to HDL code generated for Stateflow charts

When you generate HDL code from Stateflow charts, HDL Coder may create intermediate variables that are related to state transitions or perform certain computations. In previous releases, the intermediate variables were sometimes defined and used only in one branch of a conditional statement such as an if-else or case statement. Deploying this HDL code to a target device may result in the synthesis tool inferring a latch.

In R2020a, when the code generator identifies temporary variables that are not used in all branches of a conditional statement, it initializes these temporary variables to zero. This coding style prevents synthesis tools from potentially inferring a latch in the HDL code.

For example, consider this Stateflow Chart.



The generated VHDL code shows that, in R2020a, as the temporary variable `add_temp` is used only in the `ELSE` branch, the variable is initialized to zero before entering the `CASE` statement. This initialization avoids a potential latch inference by the synthesis tool.

Before R2020a	In R2020a

VARIABLE add_temp : unsigned(11 DOWNTO 0); BEGIN
is_fault_dly_sm_next <= is_fault_dly_sm;
cnt_next <= cnt;
--Gateway: inferred_latches/fault_dly_sm
fault_out <= '0';
--During: inferred_latches/fault_dly_sm
CASE is_fault_dly_sm IS
WHEN IN_sStartup =>
--During 'sStartup'
IF cnt = startup_time_unsigned THEN
--Transition
is_fault_dly_sm_next <= IN_sWait;
--Entry 'sWait'
cnt_next <= to_unsigned(16#000#, 11);
fault_out <= '0';
ELSE
add_temp := resize(cnt, 12) + to_unsigned(16#001#, 12);
IF add_temp(11) /= '0' THEN
cnt_next <= "11111111111";
ELSE
cnt_next <= add_temp(10 DOWNTO 0);
END IF;
END IF;
	VARIABLE add_temp : unsigned(11 DOWNTO 0); BEGIN
	add_temp := to_unsigned(16#000#, 12);
	is_fault_dly_sm_next <= is_fault_dly_sm;
	cnt_next <= cnt;
	--Gateway: inferred_latches/fault_dly_sm
	fault_out <= '0';
	--During: inferred_latches/fault_dly_sm
	CASE is_fault_dly_sm IS
	WHEN IN_sStartup =>
	--During 'sStartup'
	IF cnt = startup_time_unsigned THEN
	--Transition
	is_fault_dly_sm_next <= IN_sWait;
	--Entry 'sWait'
	cnt_next <= to_unsigned(16#000#, 11);
	fault_out <= '0';
	ELSE
	add_temp := resize(cnt, 12) + to_unsigned(1
	IF add_temp(11) /= '0' THEN
	cnt_next <= "11111111111";
	ELSE
	cnt_next <= add_temp(10 DOWNTO 0);
	END IF;
	END IF;

Speed and Area Optimizations

Upsampling signals without latency using Rate Transition blocks

Previously, to generate HDL code for the Rate Transition block, in the Block Parameters dialog box, you selected the **Ensure data integrity during data transfer** and **Ensure deterministic data transfer (maximum delay)** check boxes. In this case, the Rate Transition block incurred a unit delay of latency during simulation and in the generated HDL code.

In R2020a, you can upsample your signals by using the Rate Transition block without incurring the unit delay of latency. The Rate Transition block acts like a no-op for simulation and as a wire in the generated HDL code.

To perform this upsampling, in the Block Parameters dialog box of the Rate Transition block:

- Clear the **Ensure data integrity during data transfer** check box.

  Clearing this check box makes the **Ensure deterministic data transfer (maximum delay)** check box to disappear.
- Specify a fractional value of `1/n` for **Sample time multiple** where `n` is an integer. This setting configures the output port sample time of the block to be an integer multiple of the input port sample time. You can choose any value for the block parameter **Output port sample time options**.

IP Core Generation and Hardware Deployment

AXI4-Stream for MIMO: Generate IP cores with multiple input and output channels

You can now generate an HDL IP core with multiple AXI4-Stream interfaces, AXI4-Stream Video interfaces, or AXI4 Master interfaces.

To specify more than one AXI4-Stream, AXI4-Stream Video, or AXI4 Master channel. run the `IP Core Generation` workflow for your `Generic Xilinx Platform` or `Generic Altera Platform`. In the **Set Target Interface** task, on the Target platform interface table, you can then select **Add more ...** to specify additional **Target platform interfaces**. After you run this task, the additional interfaces specified are saved on the DUT subsystem as the HDL block property **AdditionalTargetInterfaces**.

If you are targeting your own custom reference design, in the `plugin_rd` file, you can specify insertion of multiple AXI4-Stream interfaces and AXI4-Stream Video interfaces by using an `addAXI4StreamInterface` method and an `addAXI4StreamVideoInterface` method of the `hdlcoder.ReferenceDesign` class.

To learn more, see Generate HDL IP Core with Multiple AXI4-Stream and AXI4 Master Interfaces.

For an example, see Running Audio Filter with Multiple AXI4-Stream Channels on ZedBoard.

High-Bandwidth AXI Master: Generate IP cores with up to 512 bits on AXI4 Master data ports

When you generate an HDL IP core with AXI4 Master interfaces, you can now use word lengths of up to `512` bits on the data port. Use the larger bit widths to integrate your HDL IP core into larger reference designs, and to achieve higher throughput when you use the AXI4 Master port to access external DDR memory.

Simulink supports fixed-point data types with word length of up to `128` bits. To model your DUT ports with word lengths greater than `128` bits, use vector data types. You can map these DUT Data ports to `AXI4 Master Read` or `AXI Master Write` ports in the Target platform interface table, generate the HDL IP core, and integrate the IP core into your Vivado or Qsys reference designs.

See Model Design for AXI4 Master Interface Generation.

Performance improvement to AXI4 Master write operations

Previously, while performing an AXI4 Master write operation, when you wrote multiple bursts of data, the AXI4 Master wrote the first burst of data and then waited for the response channel before issuing the next burst. The response is indicated by assertion of the `wr_bvalid` signal, which then asserts the `wr_complete` signal, after which the AXI4 Master sends the next burst of data. In this case, the average latency between two data bursts can be approximately `20` clock cycles, which can result in a large latency overhead, especially for multiple, small, continuous bursts.

In R2020a, the AXI4 Master write operation is more pipelined, which reduces the latency overhead and improves throughput. In this case, the `wr_complete` signal does not wait for the `wr_bvalid` signal to assert. The average latency between two data bursts can be approximately `3` clock cycles,

which is a significant improvement to write throughput, especially for multiple, small, continuous bursts.

The AXI4 Master protocol supports a maximum burst size of 256. When you have a large burst of size greater than 256, the AXI Master interface in the generated HDL IP core divides the large burst into multiple smaller bursts with size 256. With the improvement to write throughput in R2020a, you see a performance improvement to the AXI4 Master write operation even for large bursts of data.

For additional performance improvement, instead of using the `wr_complete` signal, you can use the `wr_ready` signal as an indicator for starting the next burst. When `wr_ready` asserts, the AXI4 Master can write the next data, which further reduces the average latency between two data bursts to less than 3 clock cycles.

See Model Design for AXI4 Master Interface Generation.

Dynamic customization of reference design based on reference design parameters

In R2020a, you can customize your reference design dynamically based on the value specified for the reference design parameters. You specify the value by using a new callback function, `CustomizeReferenceDesignFcn`, of the `hdlcoder.ReferenceDesign` class. By using this callback function, you can customize the block design Tcl file, reference design interfaces, reference design interface properties, and IP repositories in your reference design. For example, you can create a reference design parameter that specifies the data bitwidth of the AXI4-Stream Interface master and slave channels. In the callback function, you can add the AXI4-Stream interface by using the chosen bitwidth value.

```
% ...

% Add AXI4-Stream interface by parameterizing data width

DataWidth = hRD.getParamValue(paramValue)

if ~isempty(DataWidth)
   hRD.addAXI4StreamInterface(
    'MasterChannelEnable',     'true', ...
    'SlaveChannelEnable',      'true', ...
    'MasterChannelConnection', 'ByPass_0.AXI4_Stream_Slave', ...
    'SlaveChannelConnection',  'ByPass_0.AXI4_Stream_Master', ...
    'MasterChannelDataWidth',   DataWidth, ...
    'SlaveChannelDataWidth',    DataWidth);
end

% ...
```

Save the callback function file to the same folder as the reference design definition file, `plugin_rd.m`. You can then reference the callback function inside the `plugin_rd` file.

In this example, when you target your custom reference design for the board, in the **Set Target Reference Design** task, the **Data Width** parameter is displayed. If you specify a value for the **Data Width** such as 64 and run this task, in the **Set Target Interface** task, the callback function is evaluated. The AXI4-Stream interface that has 64-bit data width is displayed in the Target platform interface table.

See Customize Reference Design Dynamically Based on Reference Design Parameters.

Option to insert JTAG MATLAB AXI Master in standalone FPGA reference designs (requires HDL Verifier)

Previously, you could insert the JTAG MATLAB AXI Master IP in Intel and Xilinx SoC reference designs, and FPGA reference designs that already had an AXI4 slave interface or used the `hRD.addAXI4SlaveInterface`.

In R2020a, you can also specify insertion of the JTAG MATLAB AXI Master IP in standalone FPGA reference designs that do not have an AXI4 slave interface or do not use the `hRD.addAXI4SlaveInterface`. By using the JTAG MATLAB AXI Master IP, you can easily access the AXI registers in the generated DUT IP core on an FPGA board from MATLAB through the JTAG connection.

See Specify Insertion of JTAG MATLAB as AXI Master IP and https://www.mathworks.com/help/releases/R2020a/hdlcoder/examples/ip-core-generation-workflow-without-an-embedded-arm-processor-xilinx-kintex-7-kc705.html.

socExportReferenceDesign Function: Automatically create reference design (requires SoC Blockset)

Use the `socExportReferenceDesign` function to export a custom reference design from your SoC Blockset Simulink model. The `socExportReferenceDesign` function requires SoC Blockset.

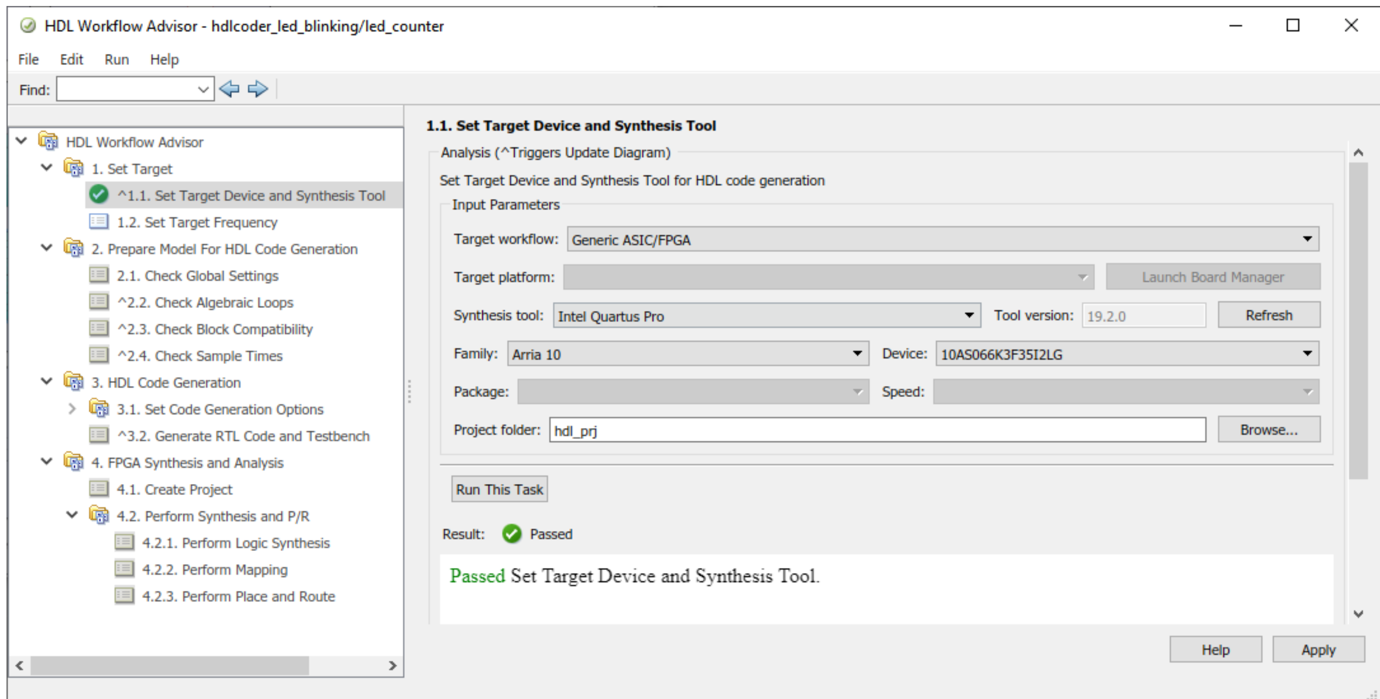
You can then use `IP Core Generation` workflow to generate a custom IP core and integrate it into your SoC reference design. For an example, see Export Custom Reference Design (SoC Blockset).

Intel Quartus Pro Targeting: Synthesize and implement generated HDL code on Intel FPGAs by using HDL Workflow Advisor

You can now specify `Intel Quartus Pro` as the **Synthesis tool** and then select `Generic ASIC/FPGA` as the **Target workflow** to synthesize and implement the generated HDL code on Intel Quartus Pro FPGA devices.

HDL Coder supports these family of devices with Intel Quartus Pro:

- `Arria 10`
- `Cyclone 10 GX`
- `Stratix 10`



Before you specify `Intel Quartus Pro` as the **Synthesis tool**, set up the tool path by using the `hdlsetuptoolpath` function. Make sure that you have already installed Intel Quartus Pro.

```
hdlsetuptoolpath('ToolName','Intel Quartus Pro','ToolPath',...
 'C:\intel\19.2_pro\quartus\bin64');
```

See also Tool Setup and HDL Language Support and Supported Third-Party Tools and Hardware.

Speedgoat IO Modules IO331 and IO331-6 being removed

Speedgoat IO modules `Speedgoat IO331` and its variant `Speedgoat IO331-6` that are based on Xilinx ISE, and use the Xilinx Spartan®-6 FPGA with the `Simulink Real-Time FPGA I/O` workflow will no longer be supported in R2020b.

Compatibility Considerations

In R2020a, you can still run the `Simulink Real-Time FPGA I/O` workflow with the Speedgoat IO modules `Speedgoat IO331` and `Speedgoat IO331-6`. However, in R2020b, if you load a pre-R2020b model that was saved with the target platform `Speedgoat IO331` or `Speedgoat IO331-6`, and then open the HDL Workflow Advisor, HDL Coder generates a warning. To avoid this warning, when you run the `Simulink Real-Time FPGA I/O` workflow, use `Speedgoat IO333-325K` or a later Speedgoat IO module that is based on Xilinx Vivado.

See also HDL Language Support and Supported Third-Party Tools and Hardware and Xilinx HDL Support with Speedgoat IO Modules.

Updates to supported software

HDL Coder has been tested with:

- Xilinx Vivado Design Suite 2019.1
- Intel Quartus Pro 19.2

See HDL Language Support and Supported Third-Party Tools and Hardware.

Simscape Hardware-in-the-Loop Workflow

Simscape Hardware-in-the-Loop: Generate HDL implementation model from multiple Simscape networks

When your Simscape model contains many switching elements, the state-space representation can contain a large number of modes. The generated HDL implementation model for such a large design can consume a significantly large number of resources, and may even fail to synthesize on the target FPGA device.

To reduce the number of modes, you can now partition the Simscape network into multiple networks, and then run the Simscape HDL Workflow Advisor. The generated HDL implementation model contains an HDL Subsystem that models the state-space equations for each Simscape network. You can also generate the validation logic that compares each Simscape network with the corresponding state-space implementation in the HDL implementation model numerically.

See Partition Simscape Models Containing a Large Network into Multiple Smaller Networks and Generate HDL Code for Simscape Models with Multiple Networks.

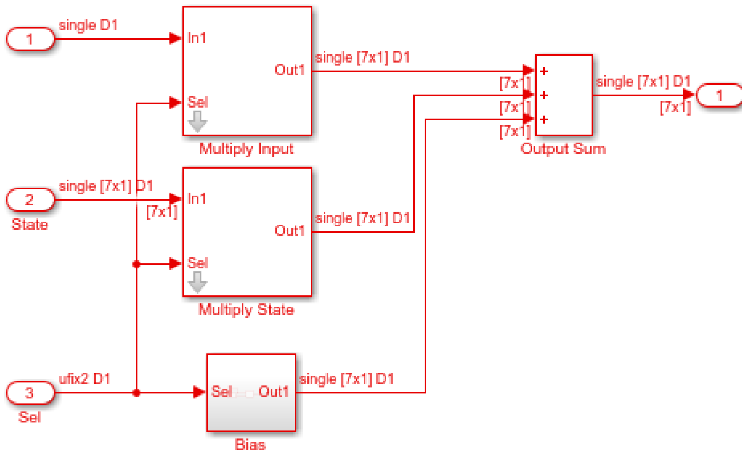
Reduction in latency of HDL implementation model generated from Simscape algorithm

When you run the Simscape HDL Workflow Advisor, the Rate Transition blocks in the HDL implementation model now have the **Ensure data integrity during data transfer** check box in the Block Parameters dialog box cleared. When this parameter is disabled, the Rate Transition blocks upsample the input without incurring a unit delay of latency. This reduces the latency of the HDL implementation model from `2` to `1`.

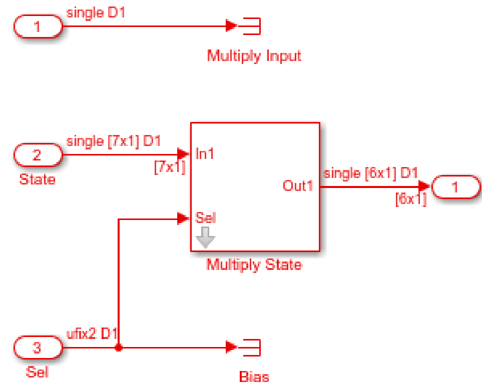
This latency reduction also enables the input values to the state space equations to be consumed one clock cycle earlier. For details on how HDL Coder upsamples the input signal to a Rate Transition block without incurring additional latency, see the release note “Upsampling signals without latency using Rate Transition blocks” on page 4-11.

Improvement to single-rate resource sharing in HDL implementation model

In R2020a, you can share the masked subsystem blocks that perform state updates and compute the output. For example, consider the HDL implementation model generated from the Simscape bridge rectifier example model, `sschdlexBridgeRectifierExample`.



Inside State Update Subsystem



Inside Output Subsystem

Inside the HDL Algorithm subsystem, if you set a **SharingFactor** of 2 on the Multiply State and Multiply Input subsystems, you can now share these subsystems, which saves area on the target FPGA device.

Sharing Report

Subsystem: Multiply State

SharingFactor: 2

Highlight shared resources and diagnostics

Group Id	Resource Type	I/O Wordlengths	Group Size	Block Name	Color Legend
1	SubSystem		2	dot_product_4	
2	SubSystem		2	dot_product_4	
3	SubSystem		2	dot_product_4	

For an example, see Generate HDL Code for Simscape Models.

R2019b

Version: 3.15

New Features

Bug Fixes

Model and Architecture Design

HDL code generation for MATLAB Function block in native floating-point mode

In R2019b, HDL Coder supports code generation for the MATLAB Function block by using floating-point data types in `Native Floating Point` mode. You can use a wider subset of MATLAB functions to develop complex floating-point algorithms.

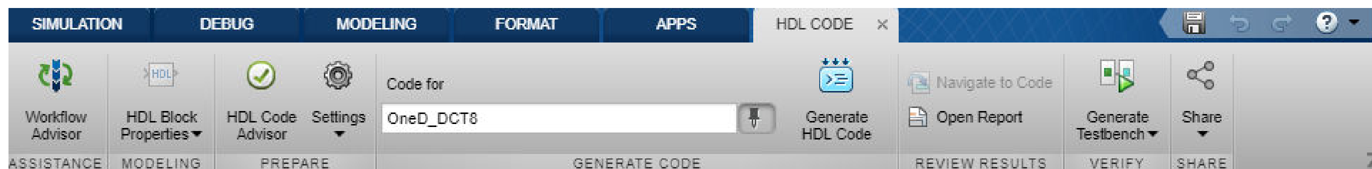
By default, floating-point support in HDL Coder uses a new `MATLAB datapath` architecture of the MATLAB Function block. This architecture treats the MATLAB Function block like a Subsystem block. The generated HDL code with the `MATLAB datapath` architecture is more readable.

To learn more, see Generate Target-Independent HDL Code with Native Floating-Point.

HDL Coder contextual tab on Simulink Toolstrip

In R2019b, the Simulink Toolstrip replaces the Simulink menu bar. For details, see **Simulink Toolstrip: Access and discover Simulink capabilities when you need them** in the Simulink release notes.

Several HDL Coder features and buttons are located in contextual tabs. The Simulink Toolstrip contextual tabs appear only when you need to access them. To access the **HDL Code** tab, open the **HDL Coder** app from the **Apps** tab on the Simulink Toolstrip.



Access options in the **HDL Code** tab, such as:

- To display the HDL Block Properties for a block or a Subsystem in your model, select that block or Subsystem. The options change contextually depending on what you select in the model.
- To access the HDL Model Checker, click the **HDL Code Advisor** button.
- To change report settings, in the **Settings** tab, select **Report Options**. To access the reports, click the **Open Report** button. The Code Generation Report opens if this report exists. Otherwise, the HDL Check Report opens.

See Set HDL Code Generation Options.

Documentation revision for HDL code generation support for blocks

Previously, the HDL Coder documentation contained a page for each supported block that described the options and limitations for HDL code generation. This information is now in the **Extended Capabilities > HDL Code Generation** section of the block page in the product that owns the block. For instance, see the HDL Code Generation section of the Divide block page in the Simulink documentation. This change consolidates information about simulation and HDL code generation to a single page for each block and avoids duplicate search results.

For a list of which blocks are supported for HDL code generation, click **Blocks** in the blue bar at the top of the Help window, and then select the **HDL code generation** check box at the bottom of the left column. The blocks are listed in their respective products. Use the table of contents in the left column to navigate between products and categories.

Documentation All Examples Functions Blocks Apps Search Help

CONTENTS Close

« Documentation Home
« Blocks

Category

DSP System Toolbox
- Signal Generation, Manipulation, and Analysis 21
- Filter Implementation 10
- Transforms and Spectral Analysis 3
- Statistics and Linear Algebra 3
- Fixed-Point Design 8

HDL Coder
HDL Verifier
LTE HDL Toolbox
Mixed-Signal Blockset
SerDes Toolbox
SimEvents
Simulink Test

Extended Capability
- C/C++ Code Generation 34
- HDL Code Generation 36
- PLC Code Generation 4
- Fixed-Point Conversion 28

DSP System Toolbox — Blocks

By Category | Alphabetical List

Results are filtered

Signal Generation, Manipulation, and Analysis

Signal Operations

Downsample	Resample input at lower rate by deleting samples
Repeat	Resample input at higher rate by repeating values
Sample and Hold	Sample and hold input signal
Upsample	Resample input at higher rate by inserting zeros
DC Blocker	Block DC component

Signal Generation

Constant	Generate constant value
NCO	Generate real or complex sinusoidal signals
NCO HDL Optimized	Generate real or complex sinusoidal signals—optimized for HDL code generation
Sine Wave	Generate continuous or discrete sine wave

Scopes and Data Logging

Spectrum Analyzer	Display frequency spectrum
Time Scope	Display and analyze signals generated during simulation and log signal data to MATLAB
Matrix Viewer	Display matrices as color images
Waterfall	View vectors of data over time
To Workspace	Write data to MATLAB workspace

Block Enhancements

Discrete FIR Filter HDL Optimized block supports complex coefficient values

The Discrete FIR Filter HDL Optimized block from the DSP System Toolbox now supports complex-valued coefficients. If both coefficients and input data are complex, the block implements each filter tap with three multipliers. If either data or coefficients are complex but not both, the block uses two multipliers for each filter tap. You can use complex coefficients with all architectures and with programmable coefficients.

This feature is also available with the `dsp.HDLFIRFilter` System object.

Process high-frame-rate or high-resolution video with multipixel streaming interface

To support high-frame-rate or high-resolution video processing, such as 4k UHD, the Vision HDL Toolbox streaming video interface can now process 4 or 8 pixels on each cycle.

When you configure the Frame To Pixels and Pixels To Frame blocks, set the **Number of pixels** parameter to `4` or `8`. With this setting, the output of the Frame To Pixels block is a vector of 4 or 8 pixel values on each time step.

The Image Filter, Edge Detector, and Median Filter blocks now support input and output vectors of 4 or 8 pixels. The **ctrl** ports remain scalar, and the control signals in the `pixelcontrol` bus apply to all pixels in the vector. The Line Buffer block can accept an input vector of 4 or 8 pixels and returns a *KernelHeight*-by-*NumberOfPixels* matrix.

Video formats for multipixel streams must have horizontal dimensions divisible by the **Number of pixels**. These horizontal dimensions are: **Active pixels per line**, **Total pixels per line**, **Front porch**, and **Back porch**. The standard video protocols 480p, 720p, 1080p, and 4k UHD support both 4 and 8 pixels at a time.

This feature is not supported for use with System objects.

OFDM Demodulator, Convolutional Encoder, and Puncturer blocks for custom wireless communication protocols

LTE HDL Toolbox provides these blocks, which are configurable to support most wireless communication protocols:

- The OFDM Demodulator block supports configurable cyclic prefix and FFT length.
- The Convolutional Encoder and Puncturer blocks support continuous, terminated, and truncated modes.

Symbol Demodulator and 1536-point FFT for LTE and NR (5G) designs

LTE HDL Toolbox provides these blocks:

- The FFT 1536 block implements a 1536-point FFT by using a single 512-point FFT.
- The LTE Symbol Demodulator and NR Symbol Demodulator blocks demodulate complex phase-shift keying (PSK) or quadrature amplitude modulation (QAM) symbols using either hard or soft decision decoding.

HDL-optimized CIC Decimation block and System Object

If you have DSP System Toolbox installed, you can use the CIC Decimation HDL Optimized block to downsample signals by using a CIC filter. The block provides an efficient hardware implementation and uses hardware-friendly control signals. The block supports HDL code generation with HDL Coder.

This algorithm is also available as a System object, `dsp.HDLCICDecimation`.

Enhancements to fixed-point Division and Reciprocal operators

In R2019b, when you use the Divide and Reciprocal blocks with fixed-point data types, you can specify a new `ShiftAdd` architecture. This architecture uses a nonrestoring division algorithm that performs multiple shift and add operations to compute the quotient. This architecture provides improved accuracy compared to the Newton-Raphson approximation method. You also achieve a higher maximum clock frequency on the target FPGA device.

For example, this table illustrates the performance of the division operation by using the `ShiftAdd` architecture on a Xilinx Virtex-7 FPGA.

Division Architecture	Latency	Fmax (MHz)	LUTs	Registers
`Linear`	0	29.69	451	-
`ShiftAdd`	20	534.19	444	636

See UsePipelines.

FWFT mode for HDL FIFO block

In R2019b, the code generator supports a First Word Fall Through (FWFT) mode for the HDL FIFO block. In this mode, the first word falls through to the output before you provide the read enable signal. The FWFT mode is especially useful when you apply the back-pressure with AXI4-Stream interfaces.

To specify this mode, in the Block Parameters dialog box of the HDL FIFO block, set **Mode** to `FWFT`. The FWFT mode provides you with the data on the same clock cycle that you request. The `Classic` mode provides you with the data on the clock cycle after you request.

HDL code generation enhancements to matrix support

In R2019b, HDL Coder extends the matrix data type support to these blocks:

- Additional blocks in the **Math Operations** library including Increment Stored Integer, Increment Real World, Decrement Stored Integer, Decrement Real World, Sum of Elements, and Product of Elements

- Blocks in the **HDL Operations** library including Multiply-Add and Multiply-Accumulate
- Discrete-Time Integrator

See Signal and Data Type Support.

Block-level option to control HDL code generated for Multiport Switch block

Previously, when you generated HDL code for the Multiport Switch block, HDL Coder used if-else statements in the Verilog code and when-else statements in the VHDL code.

Starting in R2019b, you can specify whether you want to generate HDL code with case statements or if-else statements when you use a Multiport Switch block. By default, HDL Coder generates if-else statements. To specify generation of case statements in the Verilog code or case-when statements in the VHDL code, in the HDL Block Properties dialog box for the Multiport Switch block, on the **General** tab, set CodingStyle to `case`.

HDL code generation for partitioning of mask parameters in For Each Subsystem

HDL Coder now supports code generation for Simulink models that partition the tunable mask parameters inside a For Each Subsystem. Inside the For Each Subsystem block, you can use the tunable parameter as the **Constant value** in Constant blocks and as the **Gain** parameter in Gain blocks.

See For Each and Generate HDL Code for Blocks Inside For Each Subsystem.

HDL code generation for Fcn block

In R2019b, you can generate HDL code for the Fcn block in `Native Floating Point` mode. The block accepts `single` and `double` inputs and outputs a scalar value corresponding to the mathematical expression that you specify for the block.

When you generate code, HDL Coder creates a Subsystem block with native floating-point operators that compute the mathematical expression you specify on the block.

Code Generation and Verification

UltraRAM mapping in Xilinx devices

In R2019b, when you deploy your design containing RAM blocks to a Xilinx target device that contains UltraRAM memory, you can specify the attribute to map the RAM blocks to UltraRAM. In the **HDL RAMs** library, except for Dual Port RAM and Dual Rate Dual Port RAM blocks, you can map all other RAM blocks to UltraRAM.

See RAMDirective.

Speed and Area Optimizations

Enhanced optimization support for MATLAB Function block

In R2019b, HDL Coder supports various optimizations such as resource sharing and streaming inside the MATLAB Function block with floating-point and fixed-point types. Previously, the MATLAB Function blocks acted as a barrier to the optimizations. The optimizations that you specified applied only to blocks that surrounded the MATLAB Function block.

To optimize the MATLAB Function blocks in your design, use the new `MATLAB datapath` architecture of the MATLAB Function block. Floating-point support in HDL Coder uses this architecture by default.

See HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture.

HDL optimizations across MATLAB Function blocks and other Simulink blocks

In R2019b, HDL Coder supports various optimizations such as resource sharing and streaming for the MATLAB Function block with floating-point and fixed-point types. You can use these optimizations across the MATLAB Function block with other Simulink blocks and MATLAB Function blocks in your model.

To use the optimizations, specify the new `MATLAB datapath` architecture of the MATLAB Function block. Floating point support in HDL Coder uses this architecture by default. HDL Coder treats the `MATLAB datapath` architecture of the block as a regular Subsystem block. The code generator converts the MATLAB algorithm to a Simulink block diagram.

For more information, see HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture.

Flattening of subsystems in presence of optimizations

Previously, when you used the hierarchy flattening optimization, you disabled optimizations such as resource sharing and streaming. In R2019b, you can use the hierarchy flattening optimization seamlessly in the presence of these optimizations. Flattening the subsystem hierarchy significantly reduces the number of HDL source files generated from your Simulink model.

To flatten hierarchy, set **FlattenHierarchy** to `On` for the top-level DUT Subsystem.

See also Hierarchy Flattening.

IP Core Generation and Hardware Deployment

Optimization of AXI4 slave readback logic

When your model contains many output registers and you want to read back data from multiple AXI4 slave registers, the read back logic becomes a long mux chain that can affect the target frequency. If you select the **Enable readback on AXI4 slave write registers** setting in the **Generate RTL Code and IP Core** task, the mux chain logic can further increase in length.

In R2019b, you can optimize the readback logic and achieve the target frequency that you want when you run the `IP Core Generation` workflow by using the **AXI4 slave port to pipeline register ratio** setting in the **Generate RTL Code and IP Core** task.

See Optimize AXI4 Slave Read Back Logic.

Customization of AXI4 Slave ID width in Generic IP Core Generation workflow

Previously, when you defined multiple AXI Master interfaces to access the HDL DUT AXI4 slave interface, you specified an ID width value in the reference design definition file depending on the number of AXI Master interfaces that you wanted to connect to.

In R2019b, when you run the Generic IP Core Generation workflow for your Simulink model, you can specify the AXI4 Slave ID Width from the HDL Workflow Advisor UI by using the **AXI4 Slave ID Width** setting in the **Generate RTL Code and IP Core** task.

See Generate Board-Independent HDL IP Core from Simulink Model.

Option to insert JTAG MATLAB AXI Master in SoC reference designs (requires HDL Verifier)

Starting in R2019b, you can specify insertion of the JTAG MATLAB AXI Master IP in the reference design that you are targeting. By using the JTAG MATLAB AXI Master IP, you can easily access the AXI registers in the generated DUT IP core on an FPGA board from MATLAB through the JTAG connection. See also Set Up for MATLAB AXI Master (HDL Verifier).

To use this capability:

- You must have the HDL Verifier hardware support packages installed and downloaded. See Download FPGA Board Support Package (HDL Verifier).
- Do not target standalone boards that do not have the `hRD.addAXI4SlaveInterface` or boards that are based on Xilinx ISE.

For an example, see Using JTAG MATLAB as AXI Master to control the HDL Coder IP Core.

For more information and requirements for JTAG as AXI Master IP insertion when targeting your own custom reference design, see Define Custom Parameters and Register Callback Function Handle.

Performance improvement to AXI Master interfaces in HDL DUT IP core

When you generate an HDL IP core with AXI Master interfaces, if the data width of the downstream slave device is wider than the data width of the DUT AXI Master, you might see a performance improvement. In the generated HDL IP AXI Master interface, the `AWCACHE` and `ARCACHE` signals are now set to `4'b0011`, which enables the downstream slave to modify the data transfer pattern of AXI transactions. Therefore, the slave can now pack data up to wider widths, which improves the AXI Master performance.

For example, suppose you have an RFSoC that uses a data of AXI bitwidth `128-bit` on the DUT AXI Master and a data of AXI bitwidth `512-bit` on the DDR4 memory controller IP. The AXI Interconnect can now pack the data up to `512-bit` and transfer up to four data in each clock cycle at the DDR memory controller IP side, which reduces congestion on the DDR memory access.

Updates to supported software

HDL Coder has been tested with:

- Xilinx Vivado Design Suite 2018.3
- Intel Quartus Prime Standard Edition 18.1

See Supported Third-Party Tools and Hardware.

Simscape Hardware-in-the-Loop Workflow

Enhanced HDL implementation model for Simscape and Simulink plant in feedback loop

Previously, when you ran the Simscape HDL Workflow Advisor to generate an HDL implementation model, the `HDL Algorithm` used Rate Transition blocks that ran the state update at a faster rate and the output calculation at a slower rate. When you used a plant model that contained Simscape and Simulink components inside a feedback loop, to balance the delays inside the loop, you had to use the `Zero` latency strategy in `Native Floating Point` mode.

Starting in R2019b, the HDL Algorithm runs at the single, fastest rate. The Rate Transition blocks are now placed outside the feedback loop. With this implementation, you can use the `Min` or `Max` latency strategy in `Native Floating Point` mode. Using these strategies improves the area and timing of the HDL implementation model on the target FPGA device.

For an example, see Troubleshoot Conversion of Simscape™ Permanent Magnet Synchronous Motor to HDL-Compatible Simulink Model.

Number display of differential and algebraic variables in Simscape HDL Workflow Advisor

When you run the Simscape HDL Workflow Advisor, if the **Check switched linear** task passes, the task now reports the number of differential and algebraic variables. The task also provides links to the blocks in your Simscape model that are related to these variables. Differential variables consume a quadratic amount of multiplier resources on the target FPGA device. Algebraic variables consume a linear amount of multiplier resources. You can use this information to determine how many multiplier resources your Simscape design consumes on the FPGA device and whether your design is ready for conversion to state-space representation.

See Simscape HDL Workflow Advisor Tips and Guidelines.

Separation of Get state-space parameters task for extracting and discretizing equations

The **Get state-space parameters** task in the Simscape HDL Workflow Advisor is now split into two tasks: **Extract Equations** and **Discretize Equations**. The **Extract Equations** task simulates the model to extract the differential algebraic equations. The **Discretize Equations** task discretizes the differential algebraic equations to state-space parameters.

Previously, if you changed only the **Sample time** of your model and left the model unchanged, to obtain the modified state-space parameters, you reran the **Get state-space parameters** task. Running this task could take a long time because it simulated the model to generate the differential algebraic equations, which were later discretized to state-space parameters. In R2019b, if you change the **Sample time**, you have to run only the **Discretize Equations** task. This task runs much faster because it only has to discretize the differential algebraic equations to state-space parameters.

For more information, see Simscape HDL Workflow Advisor Tasks.

Generation of implementation model with coefficients as single type and computation of results in double type

In R2019b, the **Generate implementation model** task has an enhanced layout. In the UI, you see that the **Floating-point precision** setting has an additional option, `Single coefficient, double computation`. By using this option, you can save memory usage on the target FPGA device by storing the coefficients `A`, `B`, `C`, and `D` as `single` data types. The implementation model accuracy is also improved compared to using `Single` as the **Floating-point precision** because the matrix computations are performed in double precision.

To learn more, see Simscape HDL Workflow Advisor Tips and Guidelines.

R2019a

Version: 3.14

New Features

Bug Fixes

Model and Architecture Design

Protected Model Code Generation: Share protected Simulink models with the option to allow HDL code generation

To share a model with a third-party vendor while hiding your model's intellectual property, protect the model. In R2019a, you can create a protected model that supports HDL code generation. The model that you want to protect must be a referenced model. In the parent model:

1. Right-click the model reference block that you want to protect and select **Subsystem and Model Reference > Create Protected Model for Selected Model Block**.
2. Enable HDL code generation support for the protected model by selecting **Use generated HDL code** in the Create Protected Model dialog box. You can enable password protection, which protects the model contents by using AES-256 encryption.

You can generate HDL code for models that contain protected model references created for HDL code generation. Before you generate HDL code, you must authorize the protected model references that are password-protected. For authorization, right-click the protected model reference blocks and select **Authorize**.

To learn more, see Model Protection.

Enhancements to single-precision native floating-point operators support

ULP Accuracy and DSP Usage Improvements

ULP of these native floating-point operators with single data types have improved:

Operator	Before R2019a	In R2019a
log	3	1
asinh	3	2
atanh	4	3

In addition, the maximum latency value of the log operator increased from `20` to `27`. The minimum latency of the operator is unchanged.

See also ULP of Native Floating-Point Operators.

Improvements to Rounding Function

For the Rounding Function block, in R2019a, native floating-point in HDL Coder has:

- Support for custom latency.
- Improvements to area usage and target frequency.

Additional block support with double-precision native floating-point code generation

HDL Coder now supports code generation for these blocks that have double data types in the `Native Floating Point` mode.

- Sqrt and Reciprocal Sqrt
- Rounding Function
- Data Type Conversion for conversions between double type and fixed-point data types
- MinMax
- Dot Product
- Sum of Elements and Product of Elements
- Single Port RAM, Simple Dual Port RAM, Dual Port RAM, and Dual Rate Dual Port RAM
- HDL FIFO
- Discrete-Time Integrator

To see all blocks that HDL Coder supports with double-precision data types, see Simulink Blocks Supported with Native Floating-Point.

Additional Verilog constructs supported with HDL import

HDL import now has support for more synthesizable Verilog constructs that you can use when importing your HDL file to generate the corresponding Simulink model. You can now use:

- Implicit data type conversion such as in arithmetic operations, data type conversion, bit selection, and concatenation.
- Constructs that infer RAM blocks in your Simulink model. The blocks that are inferred include Single Port RAM, Dual Port RAM, and the System-Object based RAM blocks, Single Port RAM System and Dual Port RAM System.
- Constructs that infer Compare To Constant and Gain blocks in your model.
- for loop and loop generate constructs such as for-generate, if-generate, and case-generate constructs.
- casex and casez statements.

To learn more about the supported constructs, see Supported Verilog Constructs for HDL Import.

For examples, see `importhdl`.

HDL Coder contextual tab in Simulink Toolstrip

In R2019a, you have the option to turn on the Simulink Toolstrip. See **Simulink Toolstrip Tech Preview replaces menus and toolbars in the Simulink Desktop** release note in the Simulink release notes for more details.

The Simulink Toolstrip includes contextual tabs that appear only when you need them. The HDL Coder contextual tabs include options for completing actions that apply only to HDL Coder.

- To access the **HDL Code** tab, open the **HDL Coder** app from the **Apps** tab within the Simulink Toolstrip.

- To access options in the **HDL Code** tab such as to display the HDL Block Properties for a block or a Subsystem in your model, select that block or Subsystem. The options change contextually depending on what you select in the model.

Documentation does not reflect addition of the HDL Coder contextual tabs.

HDL Coder Modeling Guidelines in Documentation

In R2019a, the HDL Coder documentation contains a list of modeling guidelines. These guidelines are general recommendations for creating Simulink models, MATLAB Function blocks, and Stateflow charts for code generation with HDL Coder.

The guidelines are divided into three sections:

- Model Design and Compatibility Guidelines: Consists of guidelines for usage of basic blocks, clock and reset signals, buses and vectors, and how to model your design hierarchically.
- Guidelines for Supported Blocks and Data Type Settings: Consists of guidelines for using various blocks in the HDL Coder block library and about the support data types.
- Guidelines for Speed and Area Optimizations: Consists of guidelines for optimizing your design for speed or area for deployment on to the target hardware.

For more information, see HDL Modeling Guidelines.

Block Enhancements

Streaming Matrix Multiply and Streaming Matrix Inverse Reference Applications

HDL Coder provides two examples that illustrates how you can perform streaming matrix inverse and streaming matrix multiplication for code generation.

- HDL Code Generation for Streaming Matrix Multiply System Object
- HDL Code Generation for Streaming Matrix Inverse System Object

Partition Offset parameter support in For Each Subsystem block

You can now generate HDL code for models that contain For Each Subsystem blocks with a nonzero value specified for the **Partition Offset** parameter of the For Each block.

See also For Each and Generate HDL Code for Blocks Inside For Each Subsystem.

Enhancements to Assignment and Selector blocks

Enhancements to Assignment block

In R2019a, HDL Coder supports all indexing modes of the Assignment block for 1-D vectors and 2-D matrices. With 1-D vectors, you can also use array of buses for all indexing modes of the block. To learn more about modeling with array of buses, see Generating HDL Code for Subsystems with Array of Buses.

Previously, the code generator supported `Index vector (port)` as the **Index Option** mode for 1-D vectors and assignment to a scalar element for 2-D matrices. The supported modes for 1-D vectors now include:

- `Assign All`
- `Index vector (dialog)`
- `Index vector (port)`
- `Starting index (dialog)`
- `Starting index (port)`

For 2-D matrices, you can now index both dimensions by using any combination of these indexing modes. To learn more about the indexing modes of the block and how to use them, see Assignment.

Enhancements to Selector block

HDL Coder now supports all indexing modes of the Selector block for 1-D vectors and 2-D matrices.

Previously, the code generator supported all **Index Option** modes for 1-D vectors and indexing of scalar elements for 2-D matrices. In R2019a, you can use any indexing mode for 1-D vectors and use any combination of these indexing modes to index both dimensions of a 2-D matrix:

- `Select All`

- `Index vector (dialog)`
- `Index vector (port)`
- `Starting index (dialog)`
- `Starting index (port)`

To learn more about the indexing modes of the block and how to use them, see Selector.

Enhancements to Discrete FIR Filter HDL Optimized block and frame-based Discrete FIR Filter block

Enhancements to Discrete FIR Filter HDL Optimized Block

The Discrete FIR Filter HDL Optimized block now provides the option to use programmable coefficients with a fully parallel systolic architecture. When you use a partly serial systolic architect, the block now optimizes symmetric and antisymmetric coefficients and provides an optional reset port. To use this block, you must have DSP System Toolbox installed.

These features are also available with the `dsp.HDLFIRFilter` System object.

HDL code generation support for programmable coefficients with frame-based Discrete FIR Filter block

The Discrete FIR Filter block now supports specifying coefficients from an input port when you use frame-based input. To use this feature, you must have DSP System Toolbox.

LTE Reference Applications: Transmitter example and TDD support for SIB recovery

LTE HDL Toolbox provides two examples:

- The LTE HDL PBCH Transmitter (LTE HDL Toolbox) reference application generates the baseband waveform specified by LTE standard TS 36.211. The waveform includes the primary synchronization signal (PSS), secondary synchronization signal (SSS), cell-specific reference signals (Cell-RS), and the master information block (MIB) for transmission through the Physical Broadcast Channel (PBCH).
- The LTE HDL SIB1 Recovery (LTE HDL Toolbox) reference application now shows how to decode SIB1 data for LTE networks that use either TDD or FDD.

Both designs support HDL code generation with HDL Coder and are ready for deployment to hardware.

OFDM Modulator block and LTE and 5G Symbol Modulator blocks

The OFDM Modulator block implements an algorithm for modulating LTE signals specified by LTE standard TS 36.212. The block modulates an encoded resource grid into time-domain OFDM samples.

The LTE Symbol Modulator block and the NR Symbol Modulator block map groups of bits to complex data symbols according to a dynamic modulation scheme. These supported modulation schemes are specified by LTE standard TS 36.211 and 3GPP 5G standard TS 38.211.

- LTE Symbol Modulator: BPSK,QPSK,16/64/256-QAM
- NR Symbol Modulator: pi/2-BPSK, BPSK,QPSK,16/64/256-QAM

To use these blocks, you must have LTE HDL Toolbox installed. Each of these three blocks provides an interface and architecture for HDL code generation and hardware deployment.

Increased kernel size limits for Image Filter block

The Image Filter block now allows for a coefficient kernel with up to 64-by-64 elements. Previously, the block restricted the coefficient kernel size to 16-by-16 elements. You can use this block if you have Vision HDL Toolbox installed.

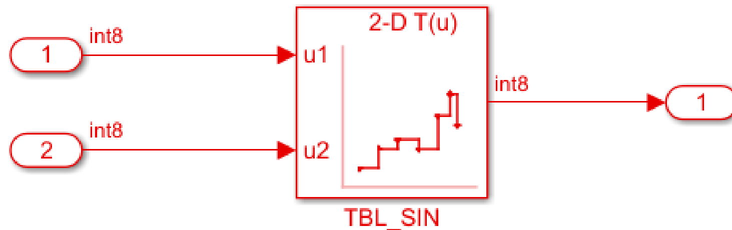
Code Generation and Verification

Customization of constant name in VHDL code generated for Lookup Table data

Previously, when you generated VHDL code for models that contain lookup tables, the CONSTANT name for the lookup table data was assigned as `nc` in the generated code, irrespective of the lookup table name.

In R2019a, the generated code uses the name of the Lookup table block followed by the postfix `_data` for the CONSTANT name in the generated code, based on the name of the Lookup table block in your Simulink model. This naming customization makes it easier to trace between the model and the generated code.

For example, consider this model that contains a Lookup Table with the name `TBL_SIN`.



The generated code for the lookup table data is as shown in table:

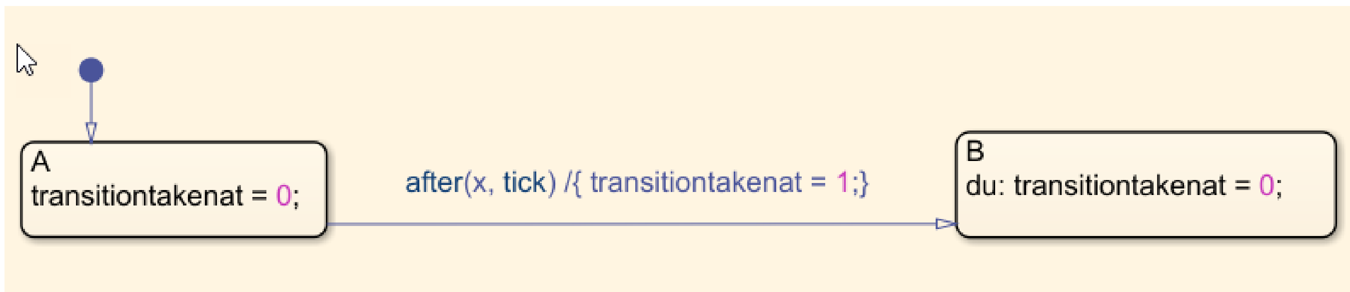
Before R2019a	In R2019a

`-- Constants`
`CONSTANT nc : vector_of_signed8(0 TO 3) :=`
`(to_signed(16#04#, 8), to_signed(16#10#, 8),`
`to_signed(16#05#, 8), to_signed(16#13#, 8));  -- sfix8 [4]`
	`-- Constants`
	`CONSTANT TBL_SIN_table_data : vector_of_signed8(0 TO 3) :=`
	`(to_signed(16#04#, 8), to_signed(16#10#, 8),`
	`to_signed(16#05#, 8), to_signed(16#13#, 8));  -- sfix8 [4]`

Optimized counters in generated HDL code for Stateflow temporal logic

Temporal logic operators produce integer or fixed-point type counters in the generated HDL code. Previously, the counter data type in the generated code was returned as `uint8`, `uint16`, or `uint32`, irrespective of the size of the fixed-point type.

In R2019a, the counters in the generated HDL code are optimized based on the operator and the type of threshold. For example, consider this Stateflow chart in your Simulink model. The variable `x` uses the data type `fixdt(0,5,0)`.



The generated Verilog code for the chart is as shown in the table. The fixed-point type is optimized to `ufix5` instead of returning as `uint8`.

Before R2019a	In R2019a

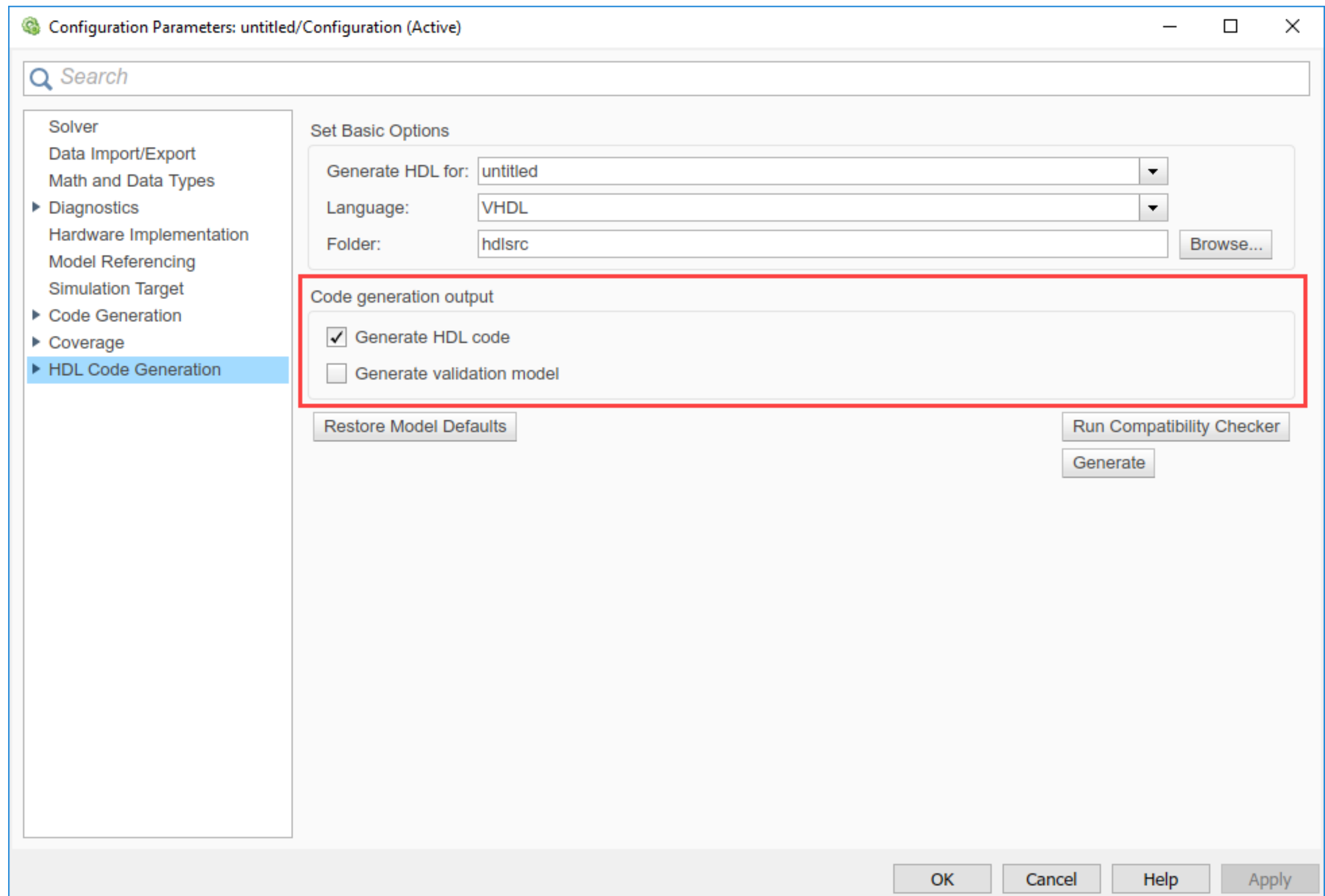
module Chart (clk, reset, enb,
          x, transitiontakenat);
 	
...
 	
  reg is_Chart; // uint8
  reg [31:0] transitiontakenat_1; // uint32
  reg [7:0] temporalCounter_i1; // uint8
  reg is_Chart_next; // enum type is_Chart (2 enums)
  reg [7:0] temporalCounter_i1_next; // uint8
  reg [7:0] temporalCounter_i1_temp; // uint8
  reg [7:0] t_0; // ufix8
 	
...
	module Chart (clk, reset, enb,
	          x, transitiontakenat);
	
	...
	
	  reg is_Chart; // uint8
	  reg [31:0] transitiontakenat_1; // uint32
	  reg [4:0] temporalCounter_i1; // ufix5
	  reg is_Chart_next; // enum type is_Chart (2 enums
	  reg [4:0] temporalCounter_i1_next; // ufix5
	  reg [4:0] temporalCounter_i1_temp; // ufix5
	
	...

HDL Coder Workflow: Enhanced options for model generation

In R2019a, HDL Coder provides options for model generation in the **HDL Code Generation** pane for better usability and performance. In the **Model Generation** tab, you can select the types of the models that you want to generate. You can customize the name of the generated model and the validation model by using **Naming options**. To control the layout of the generated models, use the **Layout options**. For more information, see Model Generation for HDL Code.

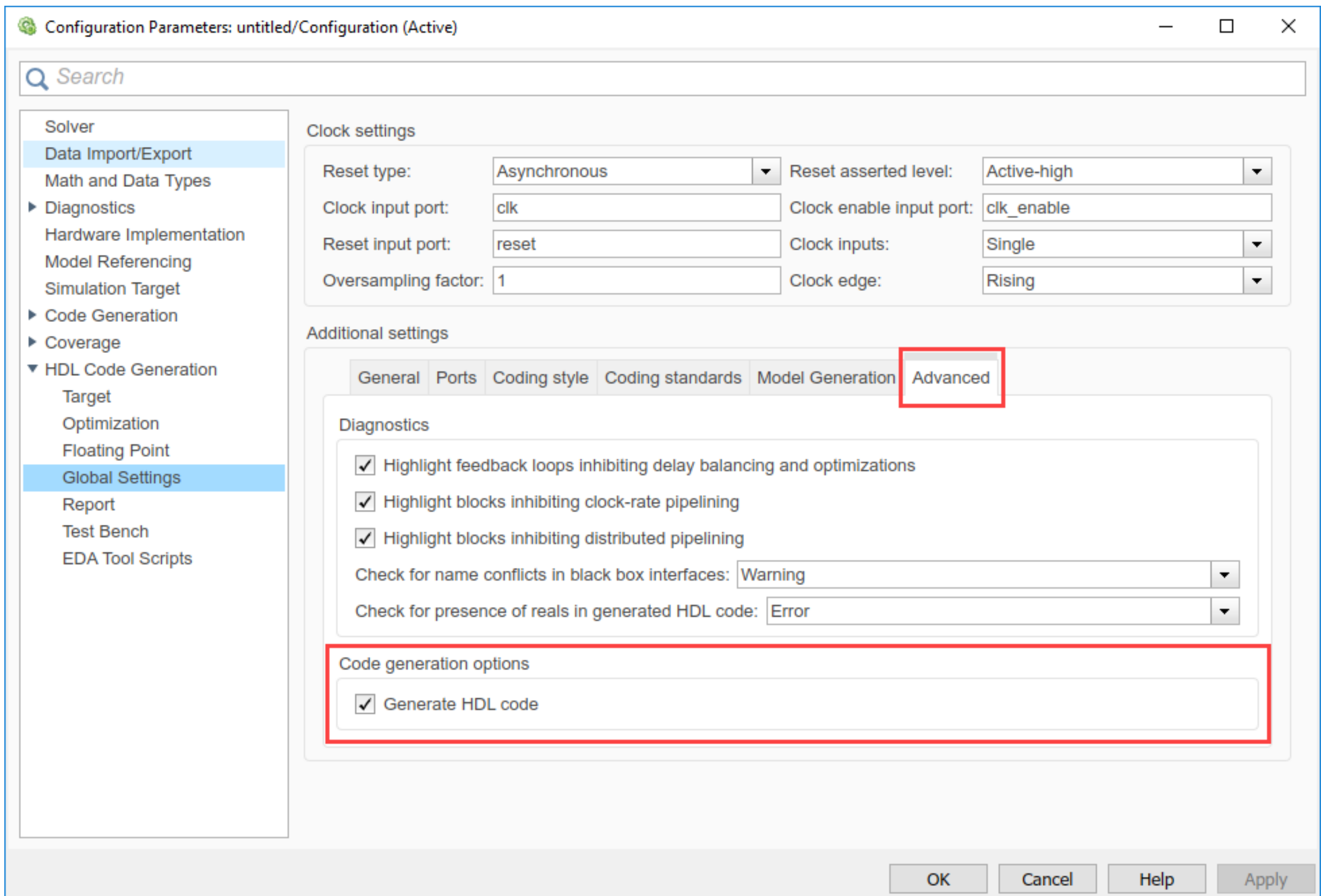
HDL Code Generation: Diagnostics tab renamed to Advanced

Before R2019a, **Code generation output** option was available when you selected **HDL Code Generation** in the left pane.



Starting in R2019a, the **Code generation output** option is available under the **Advanced** tab when you select **HDL Code Generation > Global Settings** in the right pane.

In releases before R2019a, the **Global Settings** pane had a **Diagnostics** tab. This tab has now been replaced by the **Advanced** tab, which contains the parameters listed under the **Diagnostics** section.



For more information, see Diagnostics for Optimizations, Diagnostics for Reals and Black Box Interfaces, and Code Generation Output.

Speed and Area Optimizations

Improvements to element-wise matrix transformation

Previously, when you performed element-wise matrix operations with blocks such as Add or Product, the code generator expanded the matrix operation to multiple equivalent scalar operations. The scalar operations extracted each element, performed the computation, and then combined the scalar elements to output the matrix result. This transformation made the generated model appear complex, especially for large matrices.

In R2019a, when you perform element-wise matrix operations, the code generator transforms the element-wise matrix operations in the generated model to column vectors instead of expanding it to multiple scalar operations. This transformation enables you to more effectively use optimizations such as streaming when performing matrix operations in your design.

Optimization of unconnected port for removing redundant logic in design

You can remove redundant logic in your design by deleting the unconnected ports. The optimization for unconnected ports removes all unconnected input and output ports from the generated code. It does not remove ports from the top-level DUT models or subsystems.

The optimization includes removing unconnected vector and scalar ports, bus element ports, and bus ports. Removing unconnected ports improves the readability of the generated VHDL or Verilog code and reduces code size and area usage.

For more information, see Remove Redundant Logic in Design.

IP Core Generation and Hardware Deployment

DUT AXI4 slave interface connection to multiple AXI Master interfaces in reference designs

In R2019a, HDL Coder enables you to connect AXI4 slave interfaces in the DUT HDL IP core to multiple AXI Master interfaces in the custom reference design.

Previously, when you used the `addAXI4SlaveInterface` method, you could specify only one AXI Master interface connection to the DUT AXI4 slave IP core. In R2019a, you can define multiple AXI Master interfaces which enables you to simultaneously connect your HDL DUT IP core to two or more AXI Master IP in the reference design, such as the JTAG AXI Master IP and the ARM processor in the Zynq Processing System.

To learn how you can specify multiple AXI Master interfaces in reference designs, see Define Multiple AXI Master Interfaces in Reference Designs to access DUT AXI4 Slave Interface.

Default system with External DDR4 Memory Access reference design

You can use a new `Default system with External DDR4 Memory Access` reference design when you specify `Xilinx Zynq UltraScale+ MPSoC ZCU102 evaluation kit` as the target platform.

You must have HDL Verifier and the HDL Coder Support Package for Xilinx Zynq Platform.

Generation of HDL IP core without AXI4 slave interfaces

In R2019a, when you run the Generic IP Core Generation workflow, you can generate an HDL IP core without any AXI4 slave interfaces. Use this capability if you do not want to generate an AXI4 slave interface to tune the IP core parameters.

To run this workflow, open the HDL Workflow Advisor, specify `Generic Xilinx Platform` or `Generic Altera Platform` as the target platform, and make sure that you map the DUT ports to only External IO, Internal IO, or AXI4-Stream interface with TLAST mapping. In addition, when you generate the HDL IP core, in the **Generate RTL Code and IP Core** task, clear the `Generate default AXI4 slave interface` check box, and then select **Run This Task**.For more information, see Custom IP Core Generation and Generate Board-Independent HDL IP Core from Simulink Model.

You can also create a custom reference design without an AXI4 slave interface that you can use to target standalone FPGA boards. In the reference design definition file `plugin_rd.m`, remove any mention of the `addAXI4SlaveInterface` method. For examples, see:

- https://www.mathworks.com/help/releases/R2019a/hdlcoder/examples/ip-core-generation-workflow-without-an-embedded-arm-processor-xilinx-kintex-7-kc705.html
- https://www.mathworks.com/help/releases/R2019a/hdlcoder/examples/ip-core-generation-workflow-without-an-embedded-arm-processor-arrow-deca-max-10-fpga-evaluation-kit.html

Improved synchronization of global reset signal to IP core clock domain

In R2019a, when you run the `IP Core Generation` workflow, the code generator automatically inserts a piece of logic that synchronizes the global reset signal to the IP core clock domain. The addition of this synchronization logic prevents metastability in flipflops that can occur when the reset signal changes within the setup or hold edge of the clock.

The synchronization logic works differently depending on whether you specify the **Reset type** as `Synchronous` or `Asynchronous` on the model.

- Asynchronous reset: The logic asserts the reset signal asynchronously and de-asserts the reset signal synchronously.
- Synchronous reset: The logic asserts and de-asserts the reset signal synchronously.

For more information, see Synchronization of Global Reset Signal to IP Core Clock Domain.

Minimization of clock enable signals in IP Core Generation workflow

When you run the `IP Core Generation` workflow, in the **Set Advanced Options** task of the HDL Workflow Advisor, on the **Ports** tab, if you select the **Minimize clock enables** check box, you can now minimize or remove clock enable signals in the generated HDL IP core.

To use this capability, you must:

- Specify `Free running` as the **Processor/FPGA synchronization** mode.
- Assign one of the DUT ports to the `Ready` port when mapping to AXI4-Stream Master or AXI4-Stream Video Master interfaces.

Updates to supported software

HDL Coder has been tested with Xilinx Vivado Design Suite 2018.2.

See Supported Third-Party Tools and Hardware.

Simscape Hardware-in-the-Loop Workflow

Double-precision floating-point support for HDL code generation from Simscape models

You can now generate an HDL implementation model with double data types when you run the Simscape HDL Workflow Advisor for your original Simscape model. To generate the implementation model with double data types, in the **Generate implementation model task**, specify `double` as the **Floating-point precision**.

Previously, you could generate an implementation model with single data types. In R2019a, `single` is the default data type. It is recommended that you use `single` data types, and then simulate the generated implementation model to see if your design meets the numerical accuracy requirements. If your design does not meet the requirements, use `double` as the **Floating-point precision**.

See also Simscape HDL Workflow Advisor Tasks and Validate HDL Implementation Model to Simscape Algorithm.

Validation logic verification for functional equivalence of HDL implementation model with Simscape model

In R2019a, you can specify insertion of a validation logic subsystem in the HDL implementation model when you run the Simscape HDL Workflow Advisor. By using the validation logic, you can verify the functional equivalence of the generated HDL implementation model with the original Simscape algorithm.

To insert this logic, in the **Generate implementation model task**, select the **Generate validation logic for the implementation model** check box. When you run this task and open the HDL implementation model, you see a `Validation` Subsystem block that compares the output of the Simscape algorithm with the HDL implementation.

If the HDL implementation model does not meet the numerical accuracy requirements of your design, you can increase the **Validation logic tolerance** or change the **Floating-point precision** to `double`. To learn more, see Validate HDL Implementation Model to Simscape Algorithm.

Simscape to HDL Workflow Reference Applications

Simscape to HDL workflow provides two examples:

- Replacing variable resistors illustrates how a nonlinear model that consists of variable resistors can be replaced by using an equivalent switched linear model for compatibility with Simscape to HDL workflow and generation of HDL implementation model.
- Simscape Hardware-in-the-Loop (HIL) on Speedgoat FPGA I/O Modules illustrates how you can target your Simscape algorithm onto Speedgoat FPGA I/O modules and perform Hardware-in-the-Loop (HIL) simulation.

R2018b

Version: 3.13

New Features

Bug Fixes

Compatibility Considerations

Model and Architecture Design

Verilog Import: Import synthesizable Verilog code and generate Simulink model

In R2018b, by using the `importhdl` function, you can import your HDL file that contains synthesizable Verilog code and generate the corresponding Simulink model. The generated Simulink model is an exact representation of the HDL code in terms of functionality and behavior. The imported model uses basic Simulink blocks instead of a black box representation.

By importing the code in Simulink, you can

- Verify the simulation behavior of the handwritten Verilog code in a model-based simulation environment.
- Optimize your design in Simulink and further improve area and timing on the target FPGA device by using the speed and area optimizations in HDL Coder.

To learn more, see Verilog HDL Import: Import Verilog Code and Generate Simulink Model.

Double-Precision Native Floating Point: Generate target-independent synthesizable RTL from double-precision floating-point models

In R2018b, if you have double-precision data types in your Simulink model, you can use HDL Coder native floating-point support to generate target-independent HDL code without converting to fixed point or single-precision data types. You can deploy the generated code on any generic ASIC or FPGA platform.

To see the blocks that HDL Coder supports with double-precision data types, see Simulink Blocks Supported with Native Floating-Point.

Custom latency specification for native floating-point operators

In R2018b, if you use floating-point data types as input to certain blocks in your model, you can specify a custom latency by using the HDL Coder native floating-point support. By using custom latency, you can use more latency choices other than from `Zero`, `Min`, and `Max`, which gives you better speed control. To learn about the blocks that support the custom latency setting and how to specify a custom latency, see NFPCustomLatency.

By specifying a custom latency, you can customize your design to achieve a balance between:

- Clock frequency and power consumption: A higher latency value increases the maximum clock frequency (Fmax), which increases the dynamic power consumption.
- Oversampling factor and sampling frequency: A combination of higher latency value and higher oversampling factor increases the Fmax but reduces the sampling frequency.

See also Latency of Floating Point Operators.

Enhancements to supported blocks and complex data types with single-precision native floating-point

Block Support

HDL Coder now supports the Sqrt block with **Function** set to `signedSqrt` in the native floating-point mode.

Complex Data Type Support

HDL Coder now supports complex types with these blocks in the native floating-point mode.

- Unary Minus
- Sign
- Math Function with **Function** set to `conj`, `transpose`, or `hermitian`
- Data Type Conversion
- Rounding Function with **Function** set to `floor`, `ceil`, `round`, or `fix`

See also Simulink Blocks Supported with Native Floating-Point.

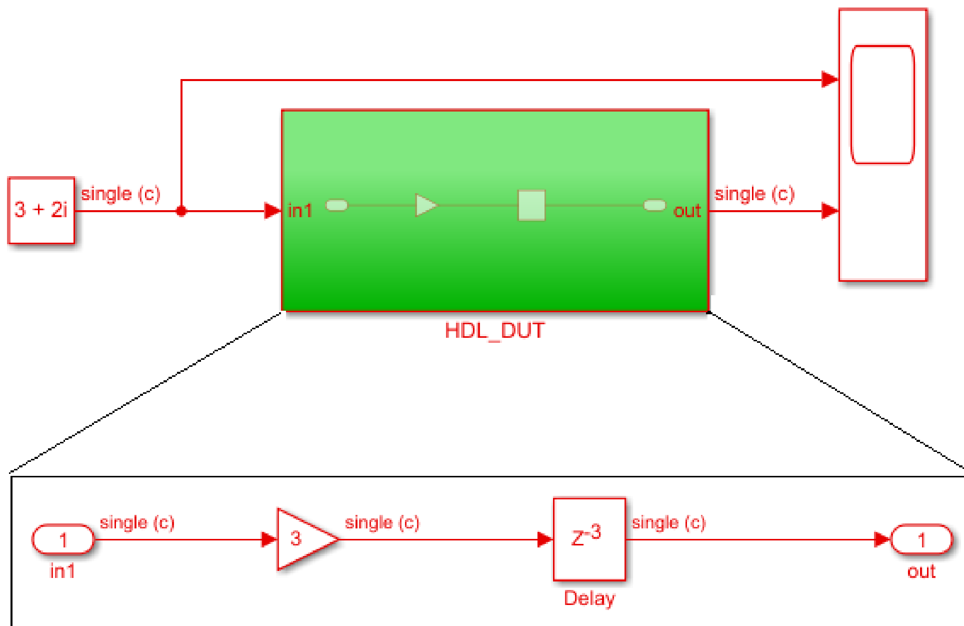
Enhancements to output delay absorption for complex multipliers with single-precision native floating-point

In R2018b, if your design uses single data types and contains complex Gain blocks that take a complex input or have a complex **Gain** parameter, or your design contains complex Product blocks, HDL Coder absorbs the delays at the output of the blocks. To use this delay absorption, before you generate HDL code, enable the `Native Floating Point` mode.

Previously, for the Gain block, HDL Coder absorbed delays when both input and the **Gain** parameter were scalars. Now, for the Gain block, HDL Coder absorbs delays in any of these cases:

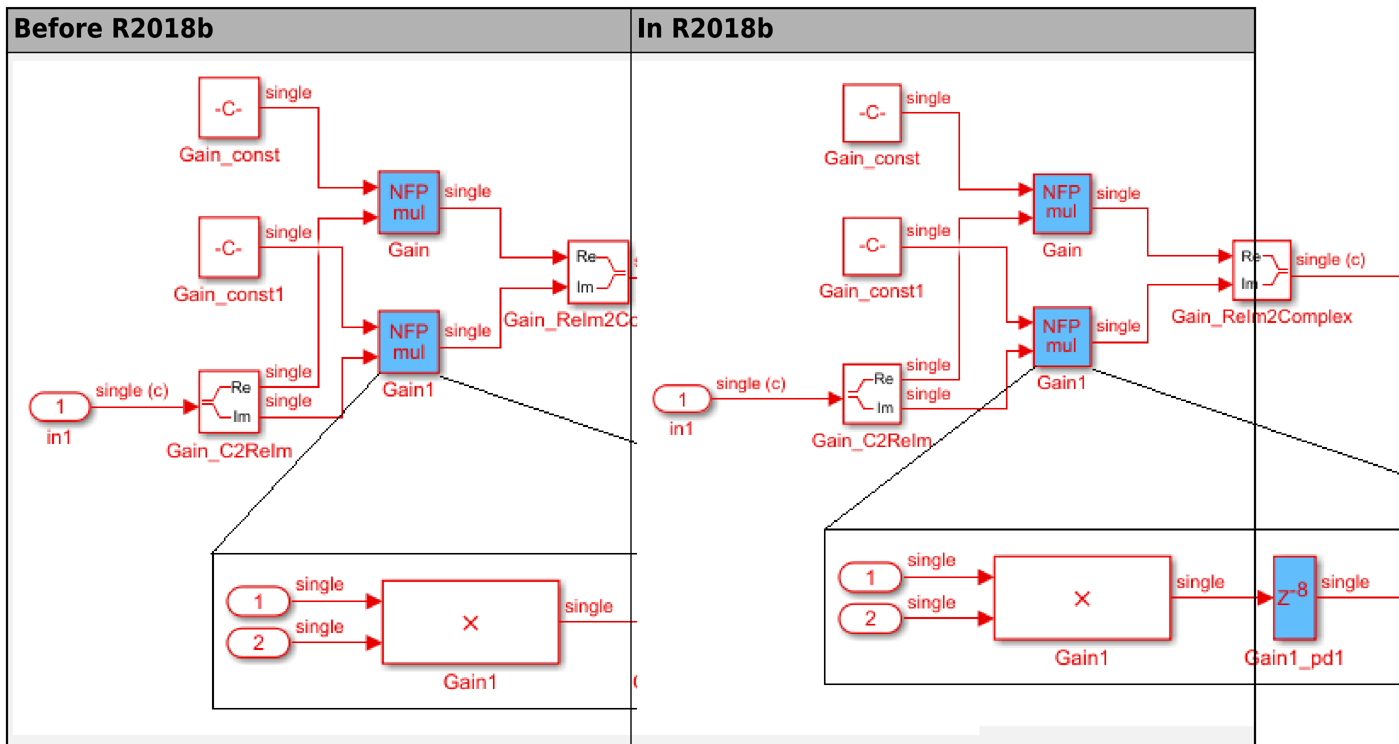
- Scalar input with scalar **Gain** parameter
- Complex input with scalar **Gain** parameter
- Scalar input with complex **Gain** parameter
- Complex input with complex **Gain** parameter

For example, this figure shows a model that uses `single` data types and inputs a complex value to a Gain block that has a scalar **Gain** parameter and an output delay of three.



This table displays the generated model when you generate code for the **HDL_DUT** Subsystem. In R2018b, you see that HDL Coder absorbs the delays as part of the floating-point product operators, **NFP_mul**. To learn more, see Latency Considerations with Native Floating Point.

Before R2018b	In R2018b



Block Enhancements

Enhancements to matrix support for HDL code generation

In R2018b, HDL Coder extends the block support to more blocks that can perform element-wise operations. Other blocks that are supported with matrix types include:

- Assignment
- From
- Goto
- MATLAB Function

To learn about blocks that are supported with matrix types, see Signal and Data Type Support.

HDL code generation support for Probe block and blocks that detect change in input signal value

In R2018b, HDL Coder supports code generation for these blocks:

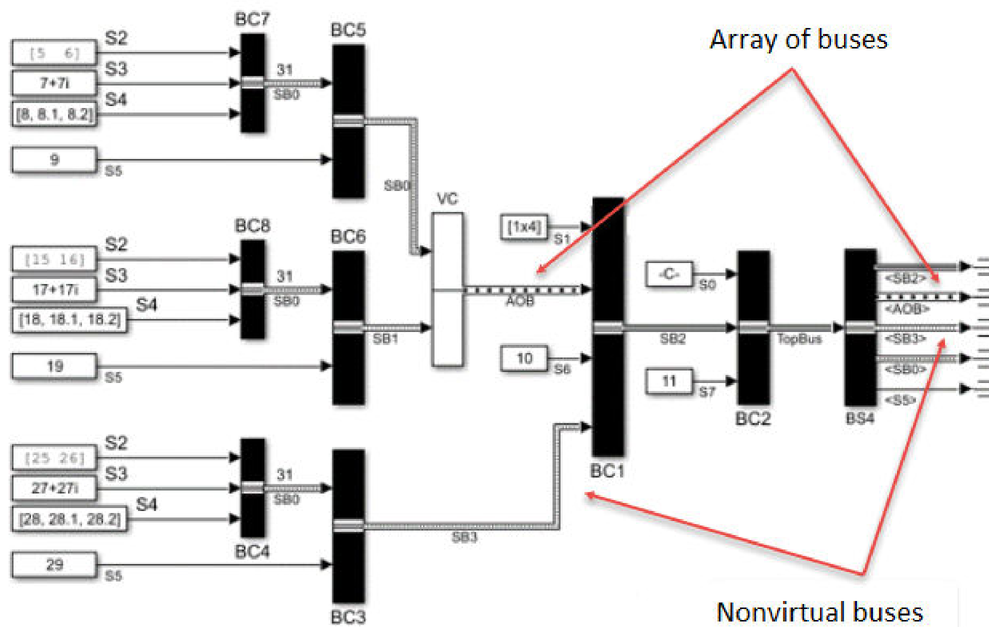
- Probe – probes selected attributes of the input signal such as width, dimensionality, sample time and offset, and whether the signal is complex-valued.
- Detect Change – detects whether there is a change in value of an input signal from the previous value.
- Detect Decrease – detects whether there is a decrease in value of an input signal from the previous value.
- Detect Increase – detects whether there is an increase in value of an input signal from the previous value.

HDL code generation support for Foreach Subsystem with Minimize global resets setting

In R2018b, HDL Coder supports code generation for a design that uses a For Each Subsystem and has the **Minimize global resets** setting enabled. To learn more about this setting, see Minimize Clock Enables and Reset Signals.

HDL Coder support for virtual bus containing nonvirtual subbus

In your design, you can now model virtual buses containing nonvirtual buses or an array of buses, and then generate HDL code for the design.



Viterbi Decoder and Depuncturer Block: Decode bitstreams by using the Viterbi algorithm with puncturing, terminated, and truncated modes (requires LTE HDL Toolbox)

The Viterbi Decoder block supports continuous, terminated, and truncated modes by using hardware-friendly control signals. The block supports punctured code rates and provides an erasure port. The Depuncturer block accepts a puncture vector as either a port or a property and provides an erasure output signal.

Both blocks support HDL code generation.

HDL code generation support for complex input signals or complex coefficients of frame-based Discrete FIR Filter and FIR Decimation blocks (requires DSP System Toolbox)

You can generate HDL code from a frame-based filter that uses either complex input signals and real coefficients or complex coefficients and real input signals. See the "Frame-Based Input Support" sections of Discrete FIR Filter and FIR Decimation.

Discrete FIR Filter HDL Optimized: Select transposed architecture, optimize symmetric and antisymmetric coefficients, and enable reset port (requires DSP System Toolbox)

The Discrete FIR Filter HDL Optimized block now provides:

- An option to use a direct form transposed architecture.

- Optimization of symmetric and antisymmetric coefficients when you select **Direct form systolic** (without **Share DSP resources** enabled) or select **Direct form transposed**. This optimization reduces the number of multipliers and makes efficient use of FPGA DSP resources.
- Optional **Reset** input port.

These features are also available on the dsp.HDLFIRFilter System object.

Compatibility Considerations

Starting in R2018b:

- The **validIn** port is mandatory. The **Enable valid input port** parameter is no longer available.
- The **ready** port is always enabled when you select **Share DSP resources** and disabled when you clear **Share DSP resources**. The **Enable ready output port** parameter is no longer available.

Code Generation and Verification

Test Point Integration with FPGA Data Capture: Use FPGA data capture to specify signals to be captured during FPGA testing by using Test Points in Simulink

You can now use FPGA Data Capture in an HDL Coder workflow. Configure the HDL Workflow Advisor to enable DUT output port generation for test point signals, and then analyze them in MATLAB or Simulink (Requires an HDL Verifier license).

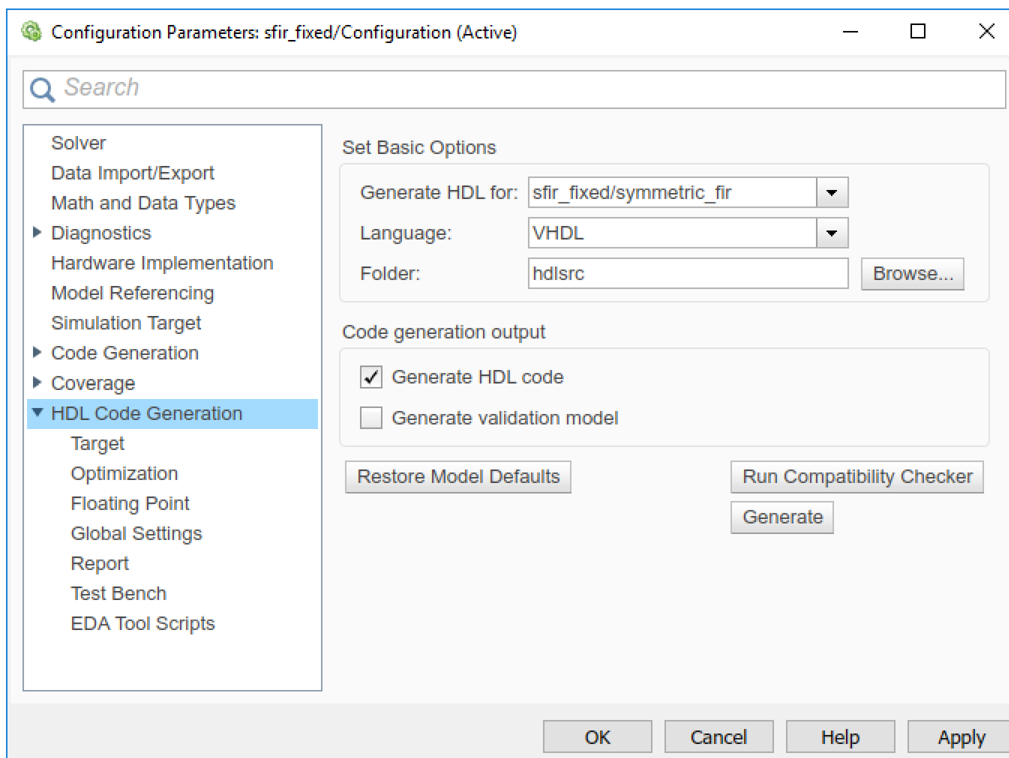
For more information, see FPGA Data Capture.

User-Interface Improvements to HDL Workflow Advisor and HDL Code Generation Pane in Configuration Parameters Dialog Box

HDL Code Generation Pane in Configuration Parameters Dialog Box

The **HDL Code Generation** pane is now reorganized and has new subpanes. The new panes includes subpanes for specifying target device settings, speed and area optimizations, reporting parameters, and so on. Before generating code, you can use this organization to more easily navigate to the parameters of interest and apply them to your model.

This figure shows the new panes added to the **HDL Code Generation** pane.

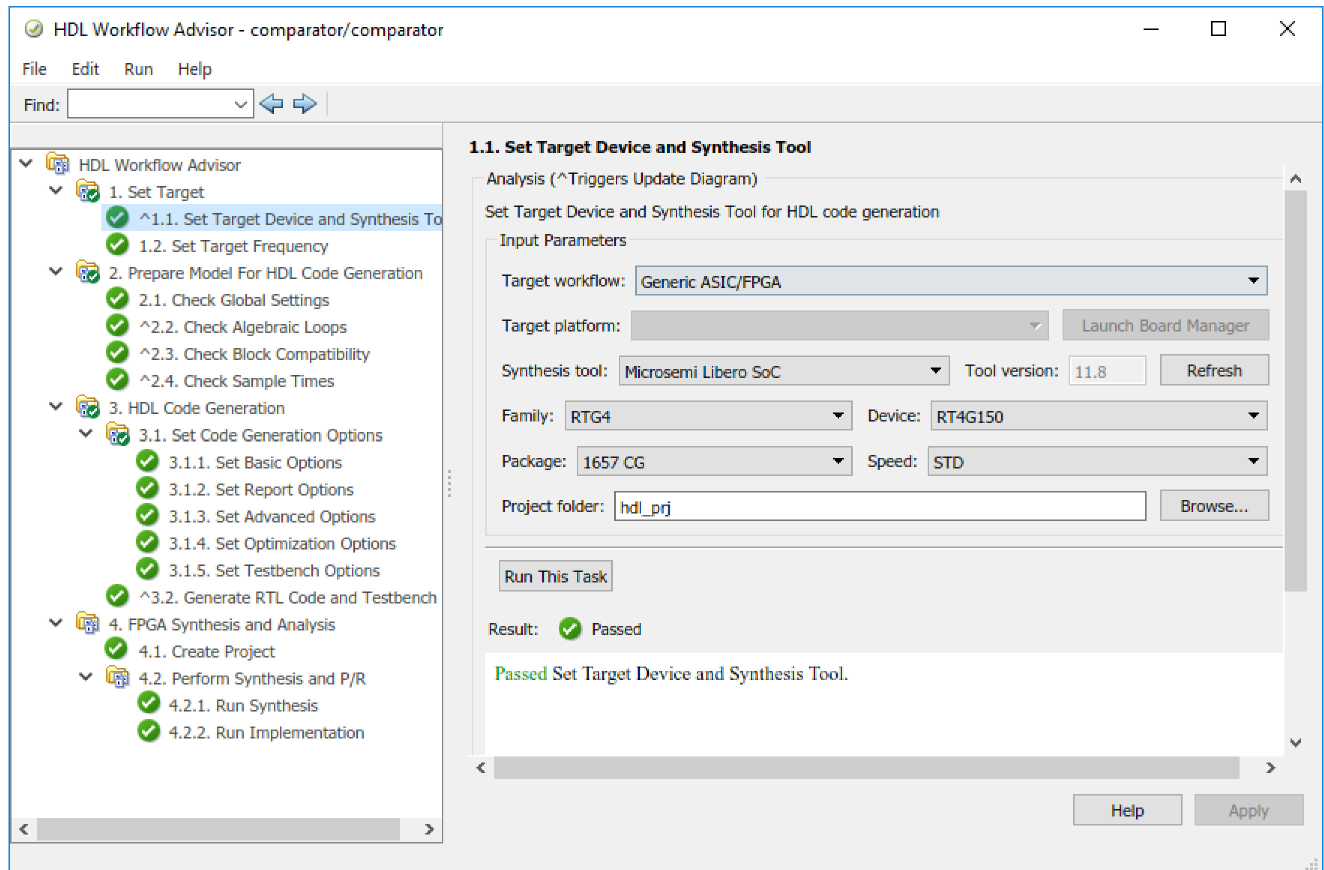


This table lists the changes to the **HDL Code Generation** pane.

HDL code generation parameters in Configuration Parameters Dialog Box	Before R2018b	In R2018b
Target frequency and synthesis tool and device parameters	**Tool and Device** section of the **HDL Code Generation > Target and Optimization** pane.	The parameters moved to a new **HDL Code Generation > Target** pane.
Multicycle path constraints and optimization parameters	**Optimization** and **Multicycle Path Constraints** sections of the **HDL Code Generation > Target and Optimization** pane.	The parameters moved to a new **HDL Code Generation > Optimization** pane.
Native floating-point and target floating-point mapping parameters	**Floating Point Target** tab of the **HDL Code Generation > Global Settings** pane.	The parameters moved to a new **HDL Code Generation > Floating Point**.
Code Generation Report parameters	**Code Generation Report** section of the **HDL Code Generation** pane.	The parameters moved to a new **HDL Code Generation > Report** pane.

HDL Workflow Advisor

The HDL Workflow Advisor has a new **Set Report Options** task. The new task appears after the **Set Basic Options** task. Use this task to specify the **Code Generation Report** parameters that were previously in the **Set Basic Options** task.


HDL Workflow Advisor - comparator/comparator
File Edit Run Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis To
1.2. Set Target Frequency
2. Prepare Model For HDL Code Generation
2.1. Check Global Settings
^2.2. Check Algebraic Loops
^2.3. Check Block Compatibility
^2.4. Check Sample Times
3. HDL Code Generation
3.1. Set Code Generation Options
3.1.1. Set Basic Options
3.1.2. Set Report Options
3.1.3. Set Advanced Options
3.1.4. Set Optimization Options
3.1.5. Set Testbench Options
^3.2. Generate RTL Code and Testbench
4. FPGA Synthesis and Analysis
4.1. Create Project
4.2. Perform Synthesis and P/R
4.2.1. Run Synthesis
4.2.2. Run Implementation
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: Microsemi Libero SoC
Tool version: 11.8
Refresh
Family: RTG4
Device: RT4G150
Package: 1657 CG
Speed: STD
Project folder: hdl_prj
Browse...
Run This Task
Result: Passed
Passed Set Target Device and Synthesis Tool.
Help
Apply

Speed and Area Optimizations

Enhancements to optimization that removes redundant logic in design

In R2018b, HDL Coder made enhancements to the optimization that removes redundant logic in design, which further reduces code size and area usage.

For example, if you have an Enabled Subsystem that does not contain useful logic, the optimization removes the Subsystem after HDL code generation. This optimization does not generate HDL code for the Enabled Subsystem, which reduces the code size and avoids potential synthesis failures with downstream tools when you deploy the generate code onto a target platform. This optimization works with both fixed-point and floating-point data types.

To learn about these enhancements, see the **Removing Subsystems** section in Dead Code Elimination.

Streaming operation modes of Multiply-Accumulate block

You can now use a streaming mode of operation for the Multiply-Accumulate block. Previously, the code generator supported the vector mode of operation. In R2018b, the code generator supports this mode as the default and provides two streaming modes. To specify when to start and stop the accumulation, and when the block output is valid, use the streaming mode control signals.

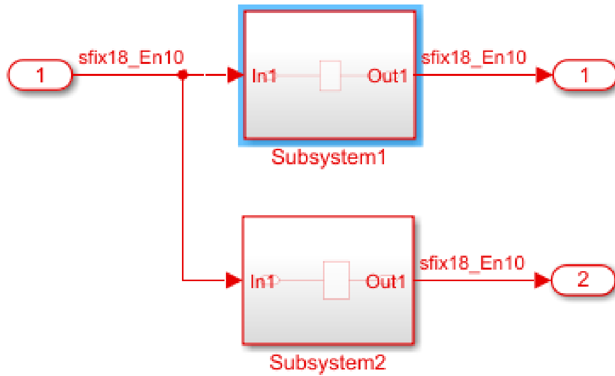
To use the streaming modes, in the Block Parameters dialog box of the Multiply-Accumulate block, for **Operation Mode**, specify either `Streaming - using Start and End ports` or `Streaming - using Number of Samples`.

To learn more, see Multiply-Accumulate.

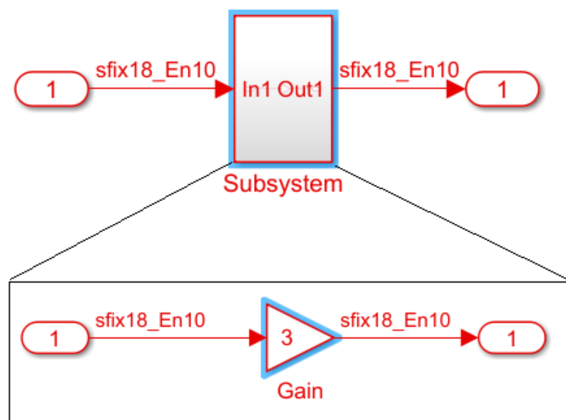
Different output latencies for designs with clock-rate pipelining enabled at output ports

Previously, when you enabled the **Allow clock-rate pipelining of DUT output ports** setting, the code generator used the same latency for sampling the DUT output ports. When you used this optimization, and if you inserted clock-rate pipelines greater than the **Oversampling factor** to any of the output ports, you incurred a simulation mismatch in the validation model.

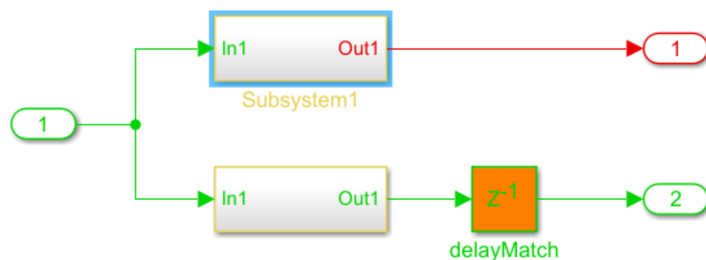
For example, consider this design that has an **Oversampling factor** of `10` and contains two different subsystems with the **ClockRatePipelining** property set to `off`.



Subsystem1 contains a Subsystem that has a Gain block with the **OutputPipeline** property set to 1 and **ClockRatePipelining** set to on. Subsystem2 contains a Subsystem block that has a Gain block with the **OutputPipeline** property set to 12 and **ClockRatePipelining** set to on.

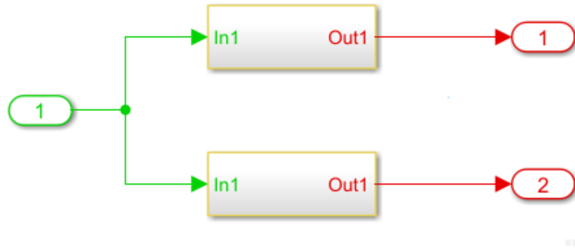


Previously, if you generated HDL code and the validation model for this design, and then simulated the validation model, you observed a simulation mismatch. This mismatch occurred because delay balancing adds a matching delay, which results in the Gain block with **OutputPipeline** set to 12 using one additional latency.



In R2018b, if you enable the **Allow clock-rate pipelining of DUT output ports** setting, the clock-rate pipelining optimization uses different latencies for sampling the DUT output ports. If you now generate HDL code and the validation model for the design containing the Gain block, delay

balancing does not insert additional matching delays, which reduces the latency and avoids the simulation mismatch in the validation model.



IP Core Generation and Hardware Deployment

Xilinx Zynq UltraScale+ MPSoC Targeting: Select from predefined targets and reference designs to generate code for MPSoC devices

In R2018b, you can target Xilinx Zynq UltraScale+ MPSoC devices, and use the `IP Core Generation` workflow to:

1. Generate an HDL IP core for the MPSoC device.
2. Generate the software interface model for the HDL IP core.
3. Integrate the HDL IP core into the HDL Coder reference designs or create your own custom reference design and target the Xilinx Zynq UltraScale+ MPSoC device.

For an example, see Getting Started with Hardware-Software Co-Design Workflow for Xilinx Zynq UltraScale+ MPSoC Platform.

Multirate IP Core Generation: Target AXI4-Stream and AXI4 Master interfaces for designs with multiple sample rates

In R2018b, HDL Coder supports the `IP Core Generation` workflow for designs that have multiple sample rates when you use any of these AXI4 interfaces:

- AXI4-Stream
- AXI4-Stream Video
- AXI4 Master

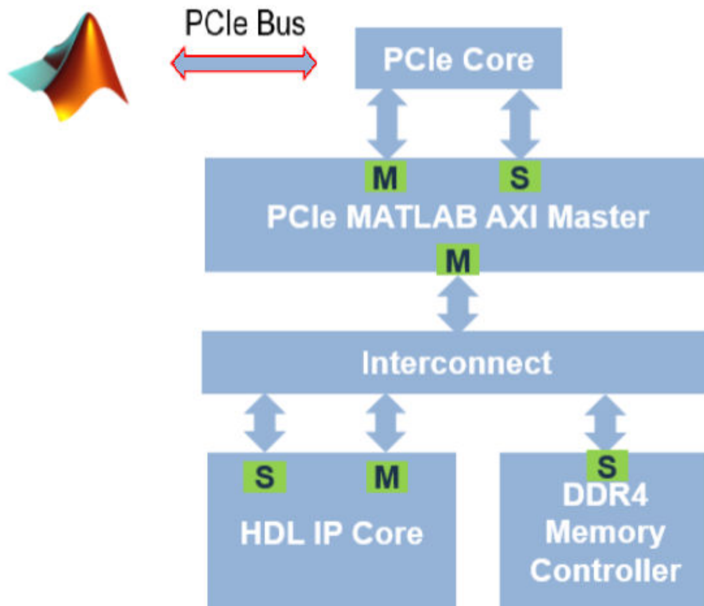
To use this workflow, ensure that the DUT ports that map to these AXI4 interfaces run at the fastest rate of the design after HDL code generation. For examples of this workflow, see Multirate IP Core Generation.

PCIe MATLAB as AXI Master with External DDR4 Memory Access reference design for Intel Arria10 GX FPGA Development kit

You can now specify `Intel Arria10 GX FPGA Development kit` as the target platform and target a new `PCIe MATLAB as AXI Master with External DDR4 Memory Access` reference design by using the `IP Core Generation` workflow. To use this reference design, you must have HDL Verifier installed.

The reference design consists of a PCIe MATLAB AXI Master IP that you can use to access the slave memory locations on board the FPGA from MATLAB. The PCIe MATLAB AXI Master IP connects to an Intel PCIe IP core. Using a PCIe bus, you can send read and write commands from the MATLAB command line to the Intel PCIe IP core, which then communicates to the PCIe MATLAB AXI Master IP. Therefore, you can use the PCIe MATLAB AXI Master IP in the reference design to transfer a large amount of data between MATLAB and the FPGA through a high-speed PCI express interface.

When you target this reference design, the HDL DUT IP core can access the external DDR4 memory by using the AXI4 Master interface. When you run the `IP Core Generation` workflow, you can map the DUT ports to AXI4 Master interfaces. To learn more, see Performing Large Matrix Operation on FPGA using External Memory.



Timing failure check in Build FPGA Bistream step of IP Core Generation workflow

In R2018b, from your Simulink model, when you run the `IP Core Generation` workflow, or the `Simulink Real-Time FPGA I/O` workflow for boards that are based on Xilinx Vivado, if the Vivado or Quartus tool is unable to meet the design timing, HDL Coder reports a timing failure in the **Build FPGA Bitstream** task. Previously, when you ran the **Build FPGA Bitstream** task, the task ignored any timing failures and displayed the results as **Passed**. To identify that there was a timing failure, you had to open the project in Vivado or Quartus and navigate to the **Project Summary** information.

In the event of a timing failure, the **Build FPGA Bitstream** task:

- Reports a message `Timing constraints NOT met!`.
- Reports the worst negative slack.
- Replaces the previous bitstream with a new bitstream that has the same name and uses a postfix `_timingfailure.bit` or `_timingfailure.sof` depending on whether you created a project by using Vivado or Quartus.
- Provides a link to the **timing_report**.
- Provides a link to an Article_on_timing_failures.

Support for read back of AXI4 write registers in IP Core Generation workflow

In R2018b, from your Simulink model, when you run the `IP Core Generation` workflow, or the `Simulink Real-Time FPGA I/O` workflow for boards that are based on Xilinx Vivado, you can read back the value that is written to the AXI4 slave write registers. HDL Coder supports the read back capability for values of scalar and vector data types that are written to the write registers by using AXI4 or AXI4-Lite interfaces.

To use this capability, in the **Generate RTL Code and IP Core** task, select the **Enable readback on AXI4 slave write registers** check box, and then run this task. This setting is saved on the DUT Subsystem for which you are generating the HDL IP core. To access this setting, in the **HDL Block Properties** > **Target Specification** tab, select **AXI4RegisterReadback**.

After you run the `IP Core Generation` workflow and program and connect to the target device, you can read back the value that is written to the registers by using the AXI4 Slave registers in the Linux console of the ARM processor. If you have HDL Verifier installed, you can use the MATLAB as AXI Master IP to read back the values.

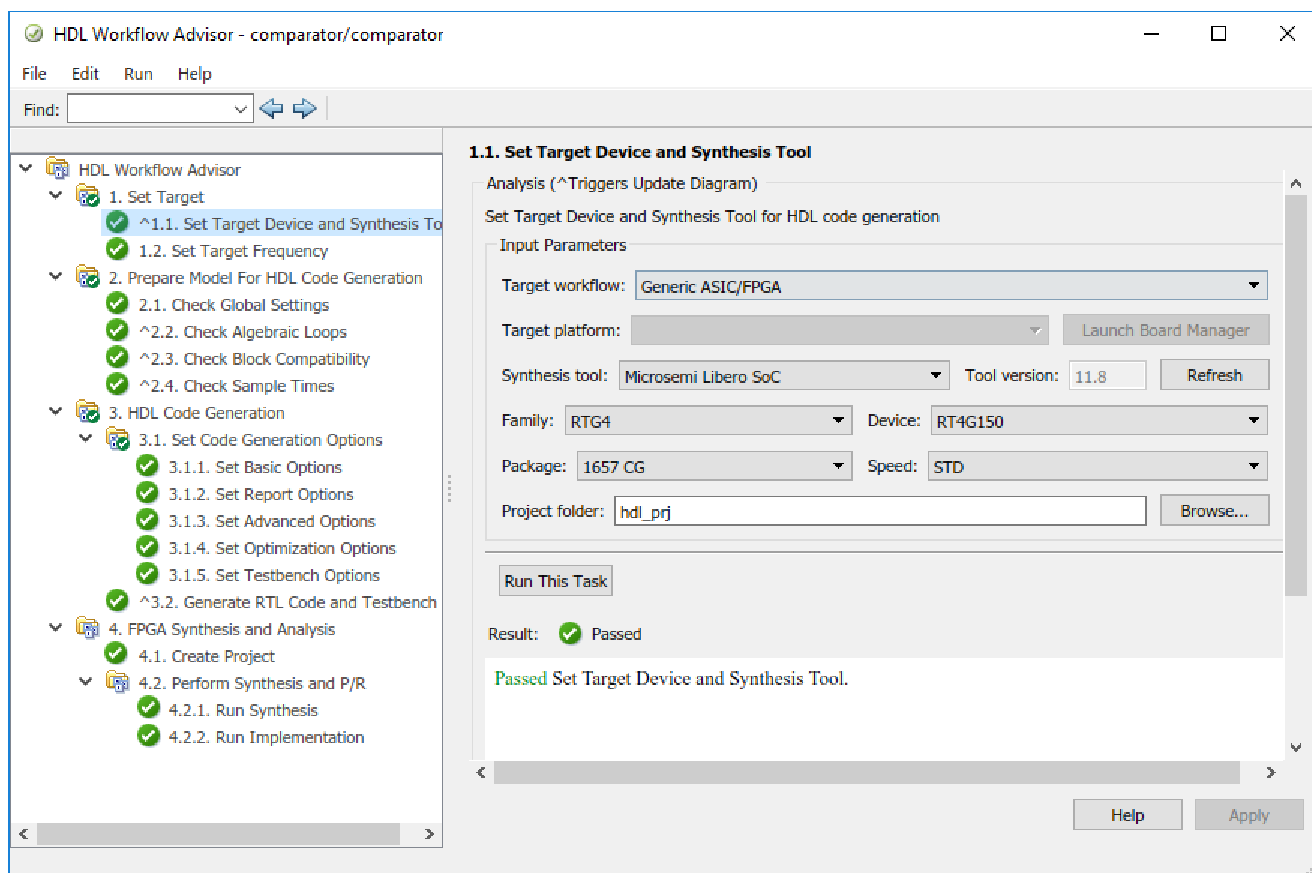
For more information, see Model Design for AXI4 Slave Interface Generation.

Microsemi Libero SoC Targeting: Synthesize and implement generated code on Microsemi FPGAs by using HDL Workflow Advisor

If you specify `Microsemi Libero SoC` as the Synthesis tool and `Generic ASIC/FPGA` as the target workflow, you can now synthesize and implement the generated HDL code on Microsemi FPGA devices.

In R2018b, HDL Coder supports these family of devices:

- `RTG4`
- `SmartFusion2`
- `IGLOO2`



Before you specify Microsemi Libero SoC as the Synthesis tool, set up the tool path by using the hdlsetuptoolpath function. Make sure that you have already installed Microsemi Libero SoC.

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

See also Tool Setup and Supported Third-Party Tools and Hardware.

Speedgoat IO Modules IO321 and IO321-5 being replaced

HDL Coder no longer supports the Speedgoat IO modules Speedgoat IO321 and its variant Speedgoat IO321-5 that use the Xilinx Virtex-4 FPGA with the Simulink Real-Time FPGA I/O workflow.

Compatibility Considerations

If you load a pre-R2018b model that was saved with the target platform Speedgoat IO321 or Speedgoat IO321-5, and then open the HDL Workflow Advisor, HDL Coder generates a warning. To avoid this warning, when you run the Simulink Real-Time FPGA I/O workflow, use the Speedgoat IO331 or a later Speedgoat board.

See also Supported Third-Party Tools and Hardware and Xilinx HDL Support with Speedgoat IO Modules.

Updates to supported software

HDL Coder has been tested with:

- Xilinx Vivado Design Suite 2017.4
- Intel Quartus Prime Standard Edition 17.1
- Microsemi Libero SoC 11.8

See Supported Third-Party Tools and Hardware.

Simscape Hardware-in-the-Loop Workflow

Hardware Acceleration of Plant Models: Generate HDL code from Simscape Electrical switched linear models

In R2018b, you can generate HDL code for your plant model that you developed by using Simscape blocks, and then deploy the generated code to a target FPGA device. Previously, before you could generate HDL code, you had to convert the Simscape model to an equivalent Simulink model.

By deploying the plant model to an FPGA, you can:

- Simulate the HDL implementation in real time with smaller time steps and increased accuracy by using hardware-in-the-loop (HIL) simulations.
- Model complex physical systems that previously took a long time to model by using Simulink blocks.
- Use the reconfigurability and parallelism capabilities of the FPGA for improved area and timing.

For more information, see Simscape .

R2018a

Version: 3.12

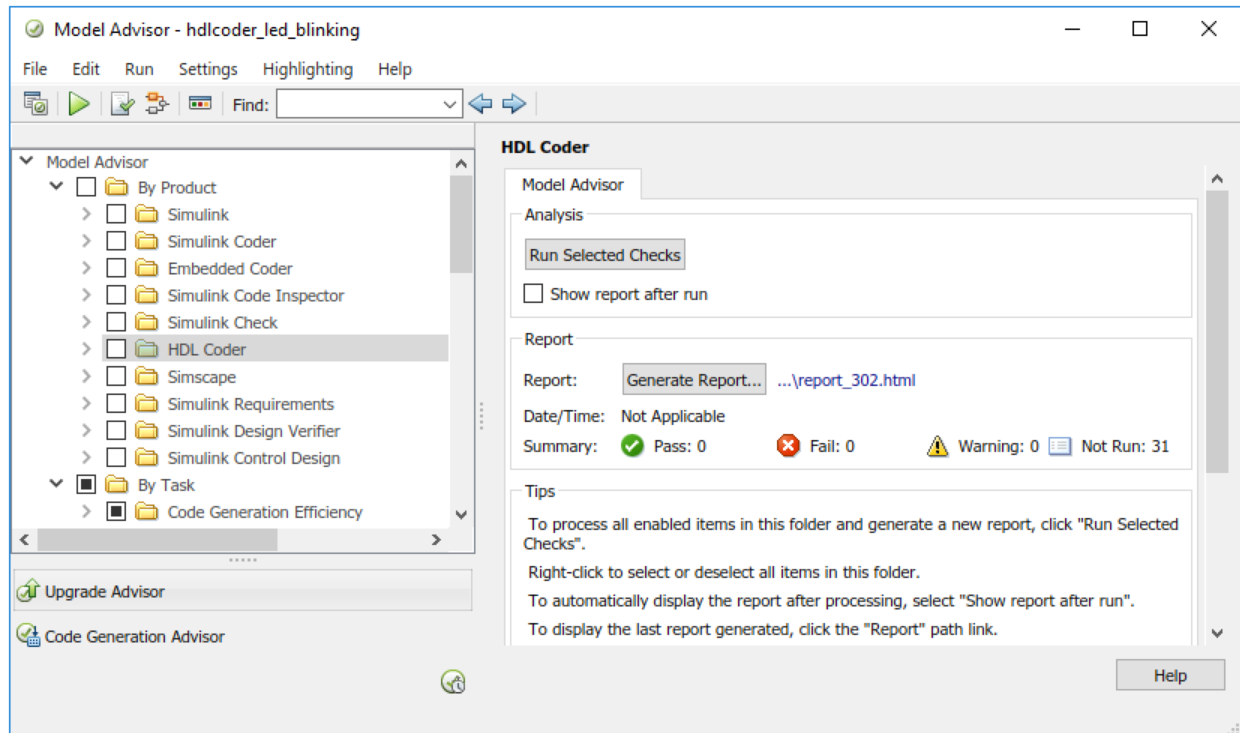
New Features

Compatibility Considerations

Model and Architecture Design

HDL Model Checker integrated with Model Advisor

HDL Coder has now integrated the checks in the HDL Model Checker into the Simulink Model Advisor. When you open the Model Advisor in Simulink, you see the checks in the **HDL Coder** subfolder of the **By Product** folder.



For more information, see Run Model Advisor Checks for HDL Coder.

Updates to model checks in HDL Coder

Checks Added

For native floating-point support, the code generator has added these checks.

- **Check for HDL Reciprocal block usage**: Checks whether your model contains HDL Reciprocal blocks that use floating-point types. This check then recommends replacing these blocks with Math Reciprocal blocks to save area and improve accuracy of your design.
- **Check for Relational Operator block usage**: Checks whether Relational Operator blocks that use floating-point types have `Boolean` outputs.
- **Check for large matrix operations**: Checks whether your model contains matrix inputs with more than two dimensions or contains large matrix operations that result in a matrix output with more than ten elements.

Checks Updated

The code generator has updated these checks:

- The **Check for infinite sample time sources** has been updated to the **Check for infinite and continuous sample time sources**.
- The **Check for Data Type Conversion blocks with incompatible settings** has been updated to check whether the blocks use the `Stored Integer (SI)` conversion mode while converting between floating-point and fixed-point data types. Previously, this check detected whether the blocks used the `Stored Integer (SI)` conversion mode and whether the **Integer rounding mode** was set to `Nearest`.

For more information, see Model Checks in HDL Coder.

Enhanced Radix-4 algorithm for Divide and Reciprocal blocks in Native Floating Point mode

The code generator now supports a `Radix-4` mode for Divide and Reciprocal blocks with `single` data types in the `Native Floating Point` mode.

The previous `Radix-2` mode, which is currently the default, offers a trade-off between latency and frequency. In R2018a, by using the `Radix-4` mode, you can trade-off your design between latency and resource usage.

To learn more, see DivisionAlgorithm.

Improved shift-and-add algorithm for exponential and hyperbolic functions in Native Floating Point mode

The code generator now uses an improved shift-and-add algorithm for the exponential and hyperbolic functions in `Native Floating Point` mode. This algorithm has a smaller ULP error of `1`, uses less resources on the target device, and achieves a higher clock frequency maximum with less latency.

The improved shift-and-add algorithm supplements the previous shift-and-add algorithm with the Taylor polynomial approximation when the absolute error is small. This algorithm reduces the number of iterations of shift-and-add operations that are required to achieve the desired accuracy.

To see the latency and ULP of floating-point operators:

- ULP of Native Floating-Point Operators
- Minimum and Maximum Latency of Floating Point Operators

HDL code generation support for all rounding modes of Data Type Conversion block in Native Floating Point mode

The code generator now supports all **Integer rounding mode** options for the Data Type Conversion block with `single` data types in `Native Floating Point` mode. Using the various supported rounding modes, you can easily convert between `single` and fixed-point data types in your design and use the same block settings for the blocks in floating-point and fixed-point domains.

Previously, the code generator supported the `Nearest` rounding mode for floating-point operations. If you used other rounding modes, you ran the **Check for Data Type Conversion blocks with incompatible settings** in the HDL Model Checker to convert the rounding mode to `Nearest`.

Floating-point control for Multiport Switch and Selector blocks

The code generator now supports single-precision floating-point data types as control input to the Multiport Switch and Selector blocks in the `Native Floating Point` mode.

Block Enhancements

Matrix Support: Generate HDL code directly from two-dimensional matrix data types and operations

In R2018a, you can use two-dimensional `matrix` data types and operations in your Simulink model for HDL code generation. With HDL Coder support, you can:

Model complicated math operations by using `Matrix` types easily.

- Use `matrix` types with all supported data types that include fixed-point, single-precision native floating-point, complex, and bus types.
- Use `matrix` types with all optimizations, particularly resource sharing.
- Generate HDL code, verify the generated code, and deploy the code onto a target platform when you use `matrix` data types for a subset of Simulink blocks in your model.

To learn about the block subset and how to use matrix types, see Signal and Data Type Support. See also Matrix Multiply.

Additional blocks and block modes supported for HDL code generation

HDL Coder now supports these blocks and block modes:

- `PartMultiplierPartAddShift` mode of mantissa multiplication for the `sin` and `cos` **Function** modes of the Trigonometric Function block. You can specify this mode by using the **Mantissa multiply strategy** setting in the Configuration Parameters dialog box or in the **Native Floating Point** tab of the HDL Block Properties for the `sin` and `cos` functions.
- Denormal handling support for the Gain block by power of two.
- External reset port with mode set to `none`, `rising`, or `falling` for the Discrete-Time Integrator block with fixed-point and `single` data types.

Bit-Natural FFT Output: Directly access the bit-natural output from the frame-based FFT/IFFT (Requires DSP System Toolbox)

You can now select bit-natural output order, with any input order, when using the frame-based mode of the HDL-optimized FFT and IFFT. Before R2018a, input and output data had to be in opposite order. The order of the input and output data are no longer restricted for these blocks and System objects:

- FFT HDL Optimized
- IFFT HDL Optimized
- `dsp.HDLFFT`
- `dsp.HDLIFFT`

Compatibility Considerations

Before R2018a, the output order of the Channelizer HDL Optimized block was bit-reversed when you set **Output vector size** to `Same as input size`. The output order is now bit-natural for both output sizes. This change also affects the `dsp.HDLChannelizer` System object.

LTE OFDM demodulation and Gold sequence generation blocks (Requires LTE HDL Toolbox)

LTE HDL Toolbox introduces two new HDL-supported blocks for LTE system design:

- Gold Sequence Generator — Generate LTE-standard Gold sequences for channel estimation and descrambling.
- OFDM Demodulator — Demodulate orthogonal frequency-division multiplexing symbols according to the LTE standard.

Additional pipelining of HDL-optimized Complex to Magnitude-Angle (Requires DSP System Toolbox)

To improve synthesized clock frequency and make better use of DSP blocks on FPGAs, the Complex to Magnitude-Angle HDL Optimized block has additional pipelining. This change also affects the `dsp.HDLComplexToMagnitudeAngle` System object.

Compatibility Considerations

The latency of the block and System object is three cycles longer than in previous releases. You must adjust the delay balancing of parallel data paths. The latency is displayed on the block.

5G filtered-OFDM modulation reference application (Requires LTE HDL Toolbox)

This example implements an F-OFDM transmitter suitable for 5G transmitter designs and verifies the design by using the 5G Library for LTE System Toolbox®. The example supports HDL code generation. It shows how to convert from double to fixed-point types and how to minimize the resource use of the design on an FPGA.

Code Generation and Verification

Line-Level Traceability: Navigate directly between Simulink blocks and corresponding lines of generated HDL code

In R2018a, when you generate the traceability report with the HDL code, the code generator provides line-level control of model-to-code and code-to-model traceability.

Previously, code-to-model and model-to-code navigation depended on block comments in the generated HDL code. In R2018a, HDL Coder does not generate block comments and provides more precise code-to-model and model-to-code traceability to lines of HDL code instead of comments. Using the report, you can now easily navigate between the blocks in your Simulink model and the generated HDL code.

For more information, see Navigate Between Simulink Model and HDL Code by Using Traceability.

Microsemi FPGA Support: Specify Microsemi Libero SoC as Synthesis Tool and generate HDL code

The code generator now supports `Microsemi Libero SoC` as a **Synthesis Tool** that you can specify in the **HDL Code Generation** > **Target and Optimizations** pane of the Configuration Parameters dialog box. When you use the `Native Floating Point` support in HDL Coder and generate code, the default `Auto` mode for the **Mantissa Multiply Strategy** setting maps your design efficiently to the multiply-accumulate (MAC) units on the Microsemi Libero FPGAs.

To specify `Microsemi Libero SoC` as a **Synthesis Tool** in the HDL Workflow Advisor, set up the path to your **Synthesis Tool** by using the `hdlsetuptoolpath` function. After you specify the tool, you can:

- Generate HDL code and verify with HDL cosimulation when you run the `Generic ASIC/FPGA` workflow.
- Run the `FPGA-in-the-Loop` workflow for your target device.

Concise summary of synthesis results displayed in HDL Workflow Advisor

The HDL Workflow Advisor now displays more concise information about the area usage and timing of your design. For additional information, the Advisor provides easily navigable links to the relevant synthesis files.

To obtain the concise report, in the **Set Target Device and Synthesis Tool** task, **Synthesis Tool** must be `Xilinx Vivado` or `Altera Quartus II`, and **Target workflow** must be `Generic ASIC/FPGA`.

For example, this figure shows the resource report when you run the **Perform Mapping** task in the HDL Workflow Advisor.

Passed Mapping.

Parsed resource summary file: FOC_Current_Control_quartus.map.rpt

Resource summary	
Resource	**Usage**
Combinational ALUTs	13405
Dedicated logic registers	48246
DSP block 18-bit elements	108

Parsed timing summary file: FOC_Current_Control_preroute.tqr

Timing summary	
	Value (ns)
Data Delay	3.584
Slack	-2.657
Timing constraints	not met

```
Task "Perform Mapping" successful.
Generated logfile: hdl_prj\hdlsrc\hdlcoderFocCurrentFloatHdl\workflow_task_PerformMappin
Info: ********************************************************************
```

New Code Generation Report: View more information and navigate through code generation results more easily

In R2018a, the code generation report has a new user interface, more information, and improved navigation.

REPORT

Back Forward Go To Find | Trace Code | Edit In MATLAB | Package Code

NAVIGATE | TRACE | EDIT | SHARE

MATLAB SOURCE

Function List | Call Tree

mlhdlc_sfir.m

fx mlhdlc_sfir

GENERATED CODE

Generated HDL Files

mlhdlc_sfir.vhd

mlhdlc_sfir.vhd | Entry-Point Summary | Code Generation Config

```
38 LIBRARY IEEE;
39 USE IEEE.std_logic_1164.ALL;
40 USE IEEE.numeric_std.ALL;
41
42 ENTITY mlhdlc_sfir IS
43   PORT( clk                 :   IN    std_logic;
44         reset               :   IN    std_logic;
45         clk_enable          :   IN    std_logic;
46         x_in                :   IN    real;  -- double
47         h_in1               :   IN    real;  -- double
48         h_in2               :   IN    real;  -- double
49         h_in3               :   IN    real;  -- double
50         h_in4               :   IN    real;  -- double
51         ce_out              :   OUT   std_logic;
52         y_out               :   OUT   real;  -- double
53         delayed_xout        :   OUT   real  -- double
54         );
55 END mlhdlc_sfir;
56
57
```

SUMMARY | ALL MESSAGES (0) | CODE INSIGHTS (0) | VARIABLES

Code generation successful

Generated on: 02-Nov-2017 15:41:59

Build type:

Output file: C:\Users\ggnanase\AppData\Local\Temp\mlhdlc_sfir\codegen\mlhdlc_sfir\hdlsrc

Processor: Generic->MATLAB Host Computer

Version: HDL Coder 3.11 (R2018a Prerelease), MATLAB Coder 4.0 (R2018a Prerelease)

Notices: –

Details: Entry Points | Settings

Reports: Conformance Report | Resource Report

You can now:

- View the HDL code with syntax highlighting.
- Find more information on the **Summary** tab, including code generation settings, entry points, and links to the Conformance Report, Resource Report, and Compliance Report.
- Navigate from the MATLAB code to context-sensitive information. For example, if you double-click a variable in the MATLAB code, you see the variable in the **Variables** tab.

In R2018a, the report is located in the same folder as in previous releases, but has a different file format. In previous releases, the report was saved with an HTML format and consisted of many files. In R2018a, the report is saved as one file with an `.mldatx` file extension.

For more information, see Code Generation Reports.

Compatibility Considerations

If you generate a report in R2018a, you cannot open it in a previous release. In R2018a, you can open reports that you generated in a previous release, but they look and behave as they did in that previous release.

Speed and Area Optimizations

Critical Path Estimation with Native Floating Point: Report critical path for designs with single-precision floating-point operations

You can now estimate the critical path of your single-precision floating-point designs in Simulink without running synthesis. Use critical path estimation to improve timing by quickly iterating through finding and pipelining the critical path in your design.

For more information, see Critical Path Estimation Without Running Synthesis.

Simplification of constant operations and other optimizations for fixed-point and floating-point arithmetic operations

The code generator now evaluates components whose inputs are constants and substitutes the components with other simplified components by propagating the constant value. This optimization avoids redundant computations during simulation and improves area and timing on the target device.

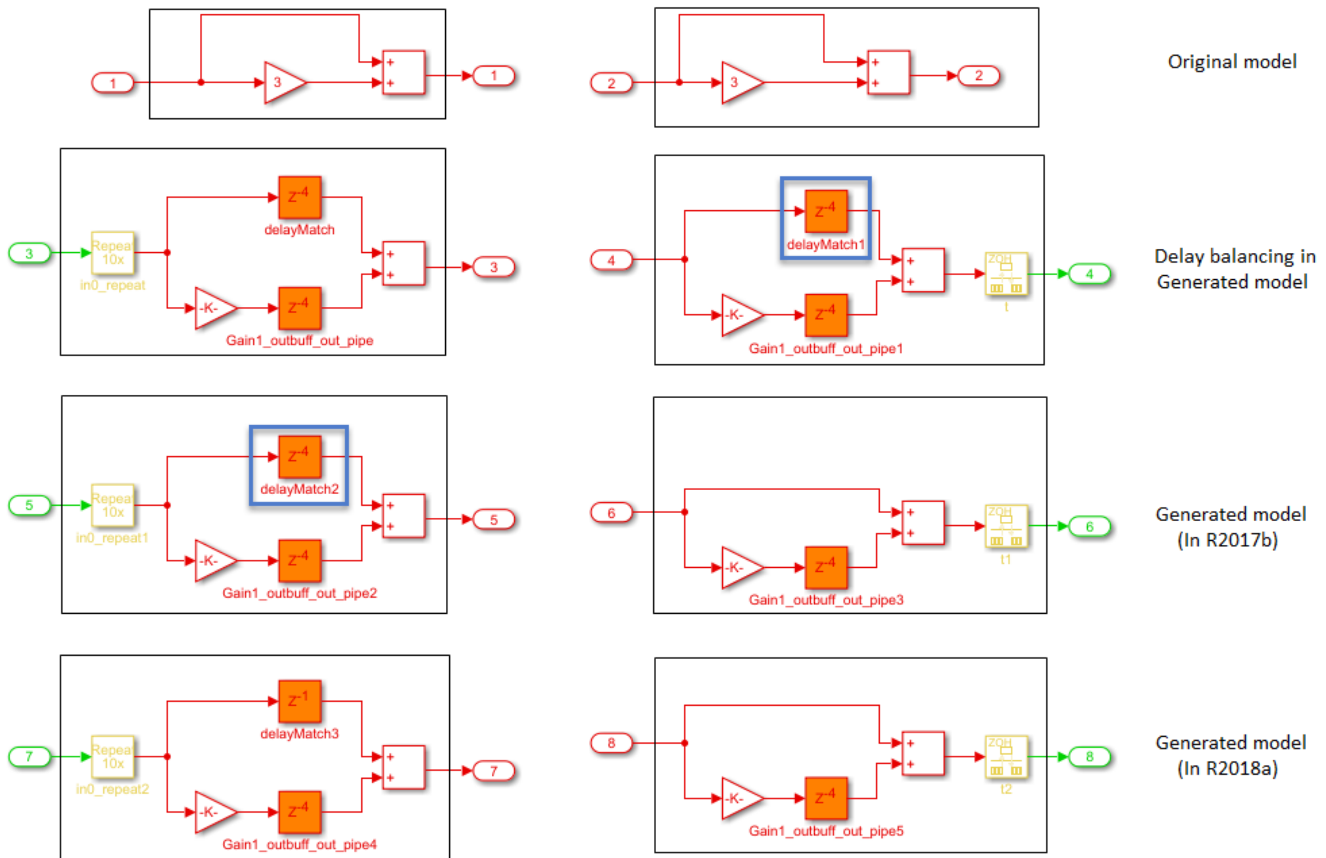
By creating modeling patterns that use a combination of these optimizations, you can significantly improve the performance of your design on the target hardware.

For more information, see Constant Folding and Peephole Optimizations in HDL Coder.

Improvement to reduction of matching delays in clock-rate pipelining regions across hierarchical boundaries

Clock-rate pipelining can introduce a latency in parts of your design that are combinational. The algorithm that the code generator uses attempts to minimize the delays within a clock-rate pipelining region. In R2018a, the code generator can reduce the delays further within a clock-rate pipelining region while preserving the hierarchy of the subsystems in your model.

For example, consider this Simulink model that has back-to-back subsystems inside a feedback loop. Each Subsystem contains a Gain block and an Add block. When you specify an **Oversampling factor** and generate code without flattening the subsystem hierarchy, HDL Coder introduces matching delays to balance the clock-rate pipelines. In R2017b, the code generator reduces the matching delays in the clock-rate pipelining region at the output Subsystem. In R2018a, the code generator can further reduce the latency by traversing the path to identify matching delays in other Subsystem blocks that can be reduced.



See also Clock-Rate Pipelining.

MaxOversampling and MaxComputationLatency parameters being removed

The `MaxOversampling` and `MaxComputationLatency` parameters are being removed. Replace these parameters with **Oversampling factor** and use **Oversampling factor** in conjunction with clock-rate pipelining.

Compatibility Considerations

In R2018a, if you load a pre-R2018a model that has the `MaxOversampling` or `MaxComputationLatency` parameters saved on the model and then generate code, HDL Coder generates a warning and ignores the parameter values during code generation. To avoid this warning, use `hdlset_param` to set `MaxOversampling` and `MaxComputationLatency` to their default values of `Inf` and `1` respectively. Specify the **Oversampling factor**, enable **Clock-rate pipelining**, and then generate HDL code.

See also Optimization with Constrained Overclocking.

IP Core Generation and Hardware Deployment

AXI4-Stream for Intel FPGA: Generate IP cores with the AXI4-Stream interface targeting Intel FPGAs

You can now use the IP Core Generation workflow to generate an HDL IP core with AXI4-Stream interface for targeting Intel FPGAs. Using the AXI4-Stream interface, you can:

- Connect to other IP cores that have AXI4-Stream interface.
- Target high-speed signal processing and video processing applications on Intel FPGAs.

To learn how to model your design, see Model Design for AXI4-Stream Interface Generation.

To generate an IP core with AXI4-Stream interface for Intel devices using the HDL Workflow Advisor:

1. In the **Set Target Device and Synthesis Tool** task, specify `IP Core Generation` as the **Target workflow** and `Generic Altera Platform` as the **Target platform**.
2. In the **Set Target Interface** task, you can assign ports to AXI4-Stream master and slave interfaces in the **Target platform interface table**. Run the workflow to generate the IP core.

You can integrate the generated IP core into your own custom Intel reference design. To learn more, see Define and Register Custom Board and Reference Design for Zynq Workflow.

Intel SoC Reference Design: Target the Intel Arria 10 SoC Development Kit with DDR4 external memory access

You can use a new `Default system with External DDR4 Memory Access` reference design when you specify `Altera Arria10 SoC development kit` as the target platform. To use this reference design, you must have HDL Verifier and the HDL Coder Support Package for Intel SoC Devices.

When you use this reference design, you can access the DDR4 external memory on the Arria 10 SoC development kit with AXI4 Master interface. Using AXI4 Master interface, you can also create custom reference designs for other Intel FPGAs and Intel SoCs for external memory access.

For more information. see Default System with External DDR4 Memory Access Reference Design.

Simulink test point port mapping in IP Core Generation and Simulink Real-Time FPGA I/O workflows

The code generator now supports test point ports in **IP Core Generation** and **Simulink Real-Time FPGA I/O** workflows that use Xilinx Vivado and Intel Quartus Prime as the synthesis tools.

When you enable DUT output port generation for test point signals in the HDL code and use the `IP Core Generation` workflow infrastructure, you can map the test point ports to AXI4, AXI4-Lite, or External Port interfaces in the **Target platform interface table**. The code generator stores this interface mapping information for the test point ports on the DUT that you can reload across subsequent runs of the workflow.

When you run the IP Core Generation workflow and generate the software interface model, you see the test point output port connected to a commented out Scope block for observing and debugging the signals.

To learn more, see Model and Debug Test Point Signals with HDL Coder™.

Audio Reference Design Example on ZYBO Board: Create custom reference design to run audio algorithm on ZYBO board

This HDL Coder example extends the audio reference design example on the ZedBoard™ to the ZYBO™ board. Using this example, you can learn how to create a custom reference design that receives audio input from the ZYBO board, processes the input signal, and transmits the processed audio output.

IP Core Generation of I2C Master Controller Example: Generate IP core for Stateflow-Based I2C Master Controller to configure Audio Codec chip

Using this HDL Coder example, you can learn how to model a generic I2C Master Controller in Simulink by using Stateflow blocks. Use the Master Controller to model an I2C Controller that can configure the Audio Codec chip. Then, run the IP Core Generation workflow to generate an HDL IP core for the I2C Controller. You can use the generated I2C Controller IP core in your custom reference design.

Ethernet programming method being removed

The Ethernet programming method is being removed. Use the Download method to program your target device.

Compatibility Considerations

In R2018a, if you run an HDL Workflow script that uses Ethernet as the programming method, HDL Coder generates an error. If you use Ethernet as the programming method and export the Workflow Advisor settings to a script, you see this code snippet in your script:

```
hWC.ProgrammingMethod = hdlcoder.ProgrammingMethod.Ethernet
```

To avoid this error, in your script, change the programming method to Download. Then, run the script.

```
hWC.ProgrammingMethod = hdlcoder.ProgrammingMethod.Download
```

You can also use JTAG and Custom as the programming methods.

For more information, see Program Target FPGA Boards or SoC Devices.

Updates to supported software

HDL Coder has been tested with:

- Xilinx Vivado Design Suite 2017.2
- Intel Quartus Prime 17.0

See Supported Third-Party Tools and Hardware.

R2017b

Version: 3.11

New Features

Bug Fixes

Model and Architecture Design

Model Advisor Checks: Check and update your Simulink model for HDL code generation compatibility

With the Model Advisor support in HDL Coder, you can now check and update your Simulink model or subsystem for compatiblity with HDL code generation. The Model Advisor checks for model configuration settings, subsystems and block settings, support with native floating point, and conformance to industry standard rules.

You can run all the checks, a certain group of checks, or individual checks in the Model Advisor. To fix warnings or failures that are reported by the checks, use the Model Advisor recommended settings. To open the Model Advisor checks for HDL Coder at the command line, use the `hdlmodelchecker` function.

For more information, see:

- Getting Started with the HDL Model Checker
- Checks In the HDL Model Checker

Simulink Test Points in HDL: Debug internal signals by automatically routing the signals to top-level HDL ports

Test points are signals that you can use to easily debug and observe the simulation results at various points in your Simulink model. In R2017b, with the HDL code generation support for test points, you can now generate code for these signals and further debug the generated code in downstream workflows. See also Test Points.

To see the test point signals in the generated HDL code:

- From the UI, in the Configuration Parameters dialog box, select Enable HDL DUT port generation for test points.
- At the command line, specify `EnableTestpoints` with `hdlset_param` or `makehdl`.

When you generate HDL code, the code generator creates an output port for the test point signal, and then propagates the signal to the DUT as an additional output port. This capability makes debugging your design easier because the code generator can propagate signals marked as test points deep within your Subsystem hierarchy to the DUT output ports.

For an example, see Model and Debug Test Point Signals with HDL Coder™.

Floating-point Support for Simulink Real-Time FPGA I/O: Generate single-precision floating point HDL for communication over the Simulink Real-Time PCIe Interface

When you use the `Simulink Real-Time FPGA I/O` workflow, you can have `Single` data type signals at the subsystem DUT ports and map the signals to PCIe or PCI Interfaces in the **Target platform interface table**. To map `Single` data type signals to PCIe interfaces, in the Configuration Parameters dialog box, on the **HDL Code Generation> Global Settings> Floating Point Target** tab, set the **Floating Point Library** to `Native Floating Point`.

Additional single-precision floating-point operators and block support

HDL Coder now supports these blocks with native floating point:

- Direct Lookup Table (n-D)
- HDL Reciprocal
- Comprehensively supports all **Function** modes of Math Function block.
- Supports all **Function** modes of Trigonometric Function block except `asinh`, `acosh`, and `atanh`.
- Float Typecast

The code generator now supports these block modes:

- Switch block where **Criteria for passing first input** can be `u2 > Threshold`, `u2 >= Threshold`, or `u2 ~= 0`. Previously, to generate HDL code in native floating-point mode, you had to use `u2 ~= 0` as the **Criteria for passing first input**.
- Relational Operator block where **Relational Operator** parameter when you input `single` data type signals can be `isInf`, `isNaN`, or `isFinite`. The modes check and output true for `inf` or `-inf`, `nan`, and so on.

See Also Simulink Blocks Supported with Native Floating-Point.

Improvements to native floating-point operators and algorithms

In R2017b, HDL Coder provides optimized algorithms for these blocks or operators in the native floating-point mode.

- When you use an Add or Subtract block, the implementation is now more optimal and uses fewer resources. The reduction in area usage on the target device is due to the block implementation using a simpler logic to handle `inf` and `nan` inputs and performing the addition or subtraction of the input mantissas using two 28-bit adders instead of one 48-bit adder.
- If you use a Data Type Conversion block that converts from `Single` to a `boolean` data type, the generated model uses Bit Slice blocks to extract the exponent and mantissa, and then compares the result with zero. When you convert from `ufix1` to a `Single` data type, the generated model uses a `Switch` block. These block implementations are more optimal and use fewer hardware resources.
- If you use a Gain block with **Gain** parameter set to `1`, the generated model uses a wire to pass the input to the output. For a **Gain** `parameter` of `-1`, the generated model shows a Unary Minus block that inverts the polarity of the input signal. These block optimizations use zero latency and reduces the resource usage on the target platform.

Input Range Reduction setting for Trigonometric Function blocks in native floating-point mode

If you have `Single` data type inputs to the Trigonometric Function block, you can use the InputRangeReduction setting in the **Native Floating Point** tab.

By default, this setting is enabled for the block, and it assumes that your input range is unbounded. If your input to the block is bounded in the range `[-pi, pi]`, your design does not require the logic to reduce the input range. In that case, you can disable this setting, and the block implementation incurs a lower latency and uses fewer resources on the target device.

Block-level latency customization for Discrete Transfer Function and Discrete Time Integrator blocks with native floating-point

For the Discrete Transfer Fcn and Discrete-Time Integrator blocks, you can now specify native floating point settings at the block level which includes HandleDenormals, LatencyStrategy, and MantissaMultiplyStrategy settings.

Block Enhancements

Minimum Resource FFT/IFFT: Reduce resource usage with the Burst Radix 2 architecture of the HDL-Optimized FFT (requires DSP System Toolbox)

You can now choose a minimum resource architecture for the HDL-optimized FFT blocks and System objects. To use this feature, select the `Burst Radix 2` architecture in these blocks and System objects:

- FFT HDL Optimized
- IFFT HDL Optimized
- `dsp.HDLFFT`
- `dsp.HDLIFFT`

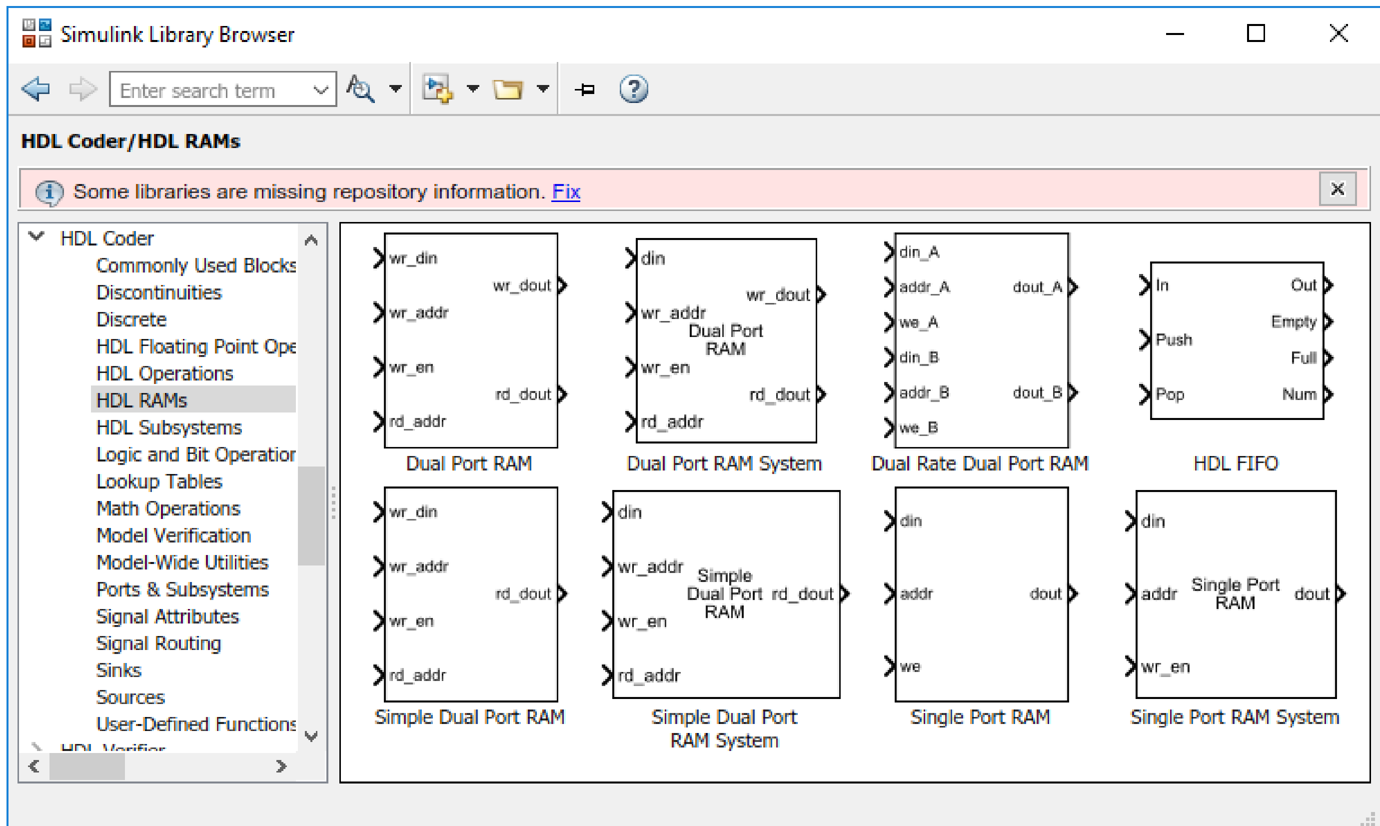
Support for scalar addressing mode with vector data input to hdl.RAM System Object

You can now use a scalar address mode with vector data input to the `hdl.RAM` System Object. With a vector data input, the write enable and address inputs can be scalar, and the system object applies the same operation to each RAM bank.

Previously, the inputs to the system object had to be all scalars or all vectors.

New HDL RAMs Block Library and hdl.RAM System Object based blocks

The `HDL Coder` block library in Simulink now has an `HDL RAMs` block library that consists of all RAM blocks and new MATLAB System blocks that are based on the `hdl.RAM` System object. These blocks are the Dual Port RAM System, Simple Dual Port RAM System, and Single Port RAM System. Previously, all RAM blocks were part of the `HDL Operations` block library.

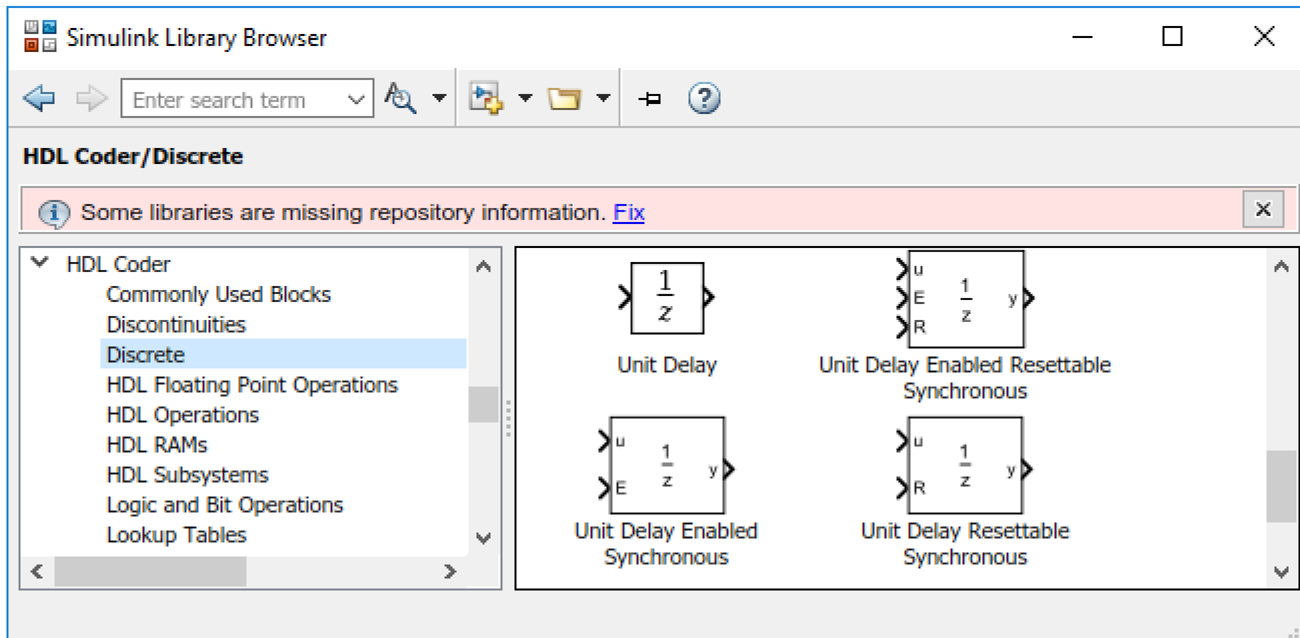


With the hdl.RAM System Object based blocks, you can:

- Specify an initial value for the RAM. Double-click the block to open the Block Parameters dialog box, and then enter a value for **Specify the RAM initial value**.
- Obtain faster simulation results when you use these blocks in your Simulink model.
- Create parallel RAM banks when you use vector data by leveraging the hdl.RAM System object functionality.
- Obtain higher performance and support for large data memories.

Synchronous versions of Unit Delay blocks with reset and enable ports in Discrete block library

In R2017b, the code generator introduces synchronous versions of the Unit Delay block with reset and enable ports in the Discrete block library in the HDL Coder library. The blocks correspond to Unit Delay Enabled Synchronous, Unit Delay Resettable Synchronous, and Unit Delay Enabled Resettable Synchronous.



The blocks use the Enabled Delay, Resettable Delay, and the Enabled Resettable Delay block with a **Delay length** of 1 in combination with the State Control block in **Synchronous** mode. The synchronous behavior of the State Control block generates cleaner HDL code and uses fewer hardware resources.

Bilateral filter, bird's-eye-view transform, and line buffer for vision applications

Vision HDL Toolbox introduces three blocks that support HDL code generation for streaming video processing designs:

- Bilateral Filter — Perform Gaussian filtering with edge preservation
- Birds-Eye View— Transform forward-facing video to a top-down perspective
- Line Buffer — Store a sliding window of pixels as part of developing custom filter algorithms

HDL code generation support for Bus Element port blocks

In R2017b, you can generate HDL code for Simulink models that use the In Bus Element and Out Bus Element blocks. These bus element port blocks provide a simple and flexible way to use bus signals as inputs and outputs to subsystems.

See also Simplify Subsystem Bus Interfaces (Simulink).

One-hot and two-hot encoding schemes for enumeration types

You can now use one-hot, two-hot, and binary encoding schemes to represent enumerated types in the generated HDL code. By default, the code generator uses a decimal encoding in Verilog and VHDL-native enumerated types in VHDL. To choose a different encoding scheme:

- From the UI, open the Configuration Parameters dialog box, and in the **HDL Code GenerationGlobal SettingsCoding Style** tab, specify **Enumerated Type Encoding Scheme**.

  At the command line, use `EnumEncodingScheme`.

This table shows the generated Verilog code from various encoding schemes for a Stateflow Chart that has four states.

Encoding Schemes

Default	Binary	One-Hot	Two-Hot
parameter is_Chart_IN_s_idle = 2'd0, is_Chart_IN_s_rx = 2'd1, is_Chart_IN_s_wait_0 = 2'd2, is_Chart_IN_s_wait_tb = 2'd3;	parameter is_Chart_IN_s_idle = 2'b00, is_Chart_IN_s_rx = 2'b01, is_Chart_IN_s_wait_0 = 2'b10, is_Chart_IN_s_wait_tb = 2'b11;	parameter is_Chart_IN_s_idle = 4'b0001, is_Chart_IN_s_rx = 4'b0010, is_Chart_IN_s_wait_0 = 4'b0100, is_Chart_IN_s_wait_tb = 4'b1000;	parameter is_Chart_IN_s_idle = 4'b00 is_Chart_IN_s_rx = 4'b0101 is_Chart_IN_s_wait_0 = 4'b is_Chart_IN_s_wait_tb = 4'

Custom header and footer comments in generated HDL code

In R2017b, you can specify custom header and footer comments for the generated HDL code. Using these custom comments, you can create templates for the header and footer comments that you can reuse across multiple designs. For example, you can specify arguments such as title, author, modified date, and so on.

```
// =======================================================
// Title           : <%Title%>
// Project         : <%Project%>
// Author          : <%Author%>
//
// Revision           : $Revision$
// Date Modified      : $Date$
// =======================================================
```

- From the UI, open the Configuration Parameters dialog box, and in the **HDL Code Generation > Global Settings > Coding Style** tab, specify **File Comment Customization** and **Custom File Header Comment**.

  At the command line, use `CustomFileFooterComment` and `CustomFileHeaderComment`.

Code Generation and Verification

Changes to HDL Code Generation Panel in Configuration Parameters Dialog Box

Parameters Added

- `Enable HDL DUT port generation for test points` in the **Global Settings** > **Ports** tab
- In the **Global Settings** > **Coding style** tab:
  - `Enumerated Type Encoding Scheme`
  - `Custom File Header Comment`
  - `Custom File Footer Comment`
- `Enable-based constraints` in the **Target and Optimizations** > **General** tab
- `Check for presence of reals in generated HDL code` in the **Global Settings** > **Diagnostics** tab

Parameters Moved

- These parameters moved from the **Target and Optimizations** > **General** tab to the **Global Settings** > **Coding Style** tab:
  - `Optimize timing controller` check box
  - `Timing controller architecture`
- The `Generate multicycle path information` check box in the **EDA Tool Scripts** tab moved to the **Target and Optimizations** > **General** tab and is now called **Register-to-register path info**.

Speed and Area Optimizations

Vector Input Multiply-Accumulate (MAC) Block: Map arithmetic operations efficiently to FPGA DSP slices

The code generator now supports a Multiply-Accumulate block that performs a multiply-accumulate operation on the input vectors and efficiently maps the generated HDL code to DSP units on the target FPGA device. Using the block, you can:

- Perform matrix multiplication operations. For example, if you have two matrix inputs with dimensions `N-by-M` and `M-by-P`, you can compute the result by using `N-by-P` multiply-accumulate operations in parallel. By combining these operations with optimizations such as resource sharing and streaming, you can improve the hardware performance by efficiently mapping the generated HDL code to DSP units on the FPGA.
- Replace a sequence of multiplication and addition operations, such as in filter blocks, and improve the performance on hardware by mapping to DSP slices on the FPGA.

The Multiply-Accumulate block is available in the **HDL Operations** sublibrary in the HDL Coder block library. The block has `Auto`, `Parallel`, and `Serial` **HDL Architecture** implementations that you can choose from.

Hierarchical Clock Rate Pipelining: Apply clock rate pipelining across hierarchical boundaries

You can now use clock rate pipelining more widely across subsystem boundaries without having to flatten the hierarchy. Preserving the subsystem hierarchy:

- Improves the modularity of your design and makes navigation through the generated model easier especially in large designs with complex hierarchies.
- Improves readability of the generated HDL code by creating multiple Verilog or VHDL files for the various Subsystem blocks in your design.

To use this optimization, disable **FlattenHierarchy** on the top-level DUT Subsystem. See also Clock-Rate Pipelining.

Support for enable-based multicycle path constraints

In multirate designs with a single clock signal, use **Enable based constraints** to meet the timing requirements for data paths operating at a rate slower than the base rate. HDL Coder generates a constraints file that specifies the enabled-based multicycle path constraints.

Previously, to generate the multicycle path information, you used the `MulticyclePathInfo` setting. This setting corresponds to the **Register-to-register path info** in the **Target and Optimizations** > **General** tab. Now, to generate the multicycle constraints file:

- In the Configuration Parameters dialog box, on the **Target and Optimizations** pane, select the **Enable based constraints** check box.
- At the command line, use `MulticyclePathConstraints`.

When you use **Enable based constraints**:

- The generated constraints are more robust to name changes in synthesis tools.
- HDL code generation is faster than when you use the **Register-to-register path info** setting.

For more information, see Meet Timing Requirements Using Enable-Based Multicycle Path Constraints.

Clock-rate pipelining enhancements

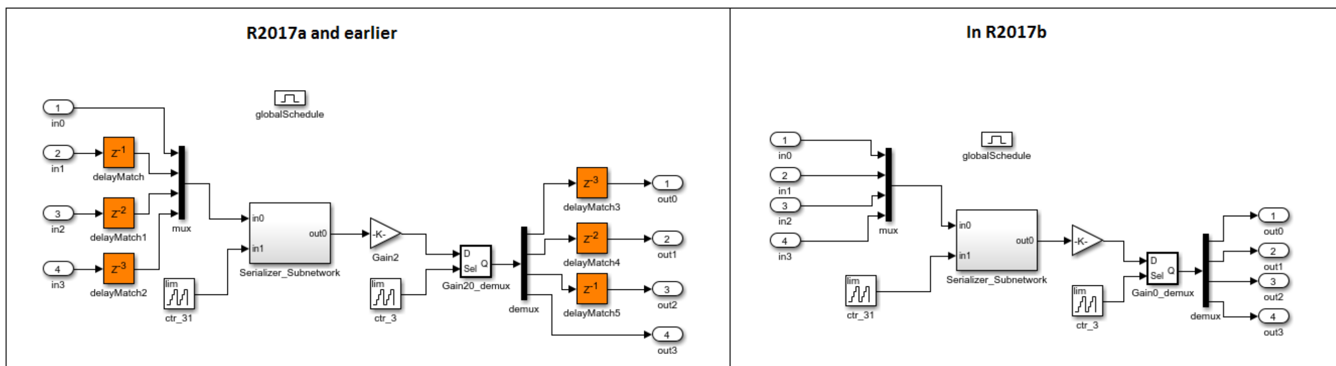
Latency reduction in the presence of design delays

The code generator can now absorb design delays that have a **delay length** greater than 1 inside a clock rate pipelining region. This optimization avoids the additional latency by accommodating the slower design delays as part of the faster clock-rate pipeline registers. Based on the length of the design delay and the **Oversampling factor** that you specify, HDL Coder inserts a certain number of pipeline registers that is equal to the **Oversampling factor** times the delay length.

Resource sharing improvements with clock rate pipelining

The code generator now does not have to add matching delays at the input and output ports when the resource sharing optimization is applied within a clock rate pipelining region. This optimization reduces the latency and resource usage significantly, especially for large values of **Sharing factor**.

This figure shows the generated model when resource sharing is applied in a clock rate pipelining region in R2017b and prior releases.



IP Core Generation and Hardware Deployment

AXI4 Master Interface: Facilitate communication between your design and external memory by using the AXI4 Master protocol for more flexible data access

When your design uses algorithms that require accessing large data sets from an external memory, you can generate an HDL IP core with AXI4 Master interface that can communicate between your design and the external memory controller IP by using the AXI4 Master protocol. Use the AXI4 Master interface when your:

- Design targets multiframe video processing applications. You can store the image data in external memory, such as a DDR3 memory on board, and then read or write the images to your design in a burst fashion for high-speed processing.
- Algorithm must access memory data in a nonstreaming arbitrary pattern.
- DUT IP core must control other IPs with the AXI4 slave interface in the system. This capability is especially useful in standalone FPGA devices.

If you use `Xilinx Zynq ZC706 evaluation kit` as the **Target platform**, you can integrate the generated HDL IP core into the `Default system with External DDR3 Memory Access` reference design. Optionally, you can integrate the HDL IP core with AXI4 Master Interface into your own custom reference design by using the `addAXI4MasterInterface` method of the `hdlcoder.ReferenceDesign` class.

To learn more, see Model Design for AXI4 Master Interface Generation.

IP Core Generation Support for Xilinx System Generator: Generate an HDL IP core for DUT containing System Generator blocks

When you use the `IP Core Generation` workflow or workflows that use the `IP Core Generation` workflow infrastructure such as `Simulink Real-Time FPGA I/O`, you can have Xilinx System Generator Subsystem blocks inside the DUT.

To learn how to generate HDL code from your DUT containing Xilinx System Generator blocks, see Using Xilinx System Generator for DSP with HDL Coder.

INOUT port type support for External Port interface in IP Core Generation workflow

With the `IP Core Generation` workflow, you can now specify `INOUT` port types on a blackbox subsystem and then map the corresponding DUT ports to `External Port` interfaces in the **Target platform interface table** when you run the workflow.

To learn how to specify `INOUT` port types, see Specify Bidirectional Ports.

Faster Simulink Real-Time FPGA I/O model build time with version register in generated IP core

For the newer Speedgoat boards that use the `IP Core Generation` workflow infrastructure, when you use the `Simulink Real-Time FPGA I/O` workflow, the generated IP core now contains a

unique timestamp. The IP Core Generation report shows an `IPCore_Timestamp` register that contains information about when the IP core was created up to the minute time value. The code generator appends this information to the bitstream file name, and then copies the file to the current working directory for easier access.

By using the timestamp, you can match the packaged HDL IP core to the FPGA bitstream that gets downloaded to the board. When you run the workflow, HDL Coder generates a `Simulink Real-Time FPGA I/O` interface model that contains the timestamp information as part of the Setup block. When you build this model, the code generator compares the timestamp with the FPGA bitstream file name that is used to program the FPGA. If there is a match in the timestamp name, the FPGA no longer needs to be reprogrammed, which significantly reduces the model build time.

Default system with External DDR3 Memory Access reference design

You can use a new `Default system with External DDR3 Memory Access` reference design when you specify `Xilinx Zynq ZC706 evaluation kit` as the target platform.

You must have HDL Verifier and the HDL Coder Support Package for Xilinx Zynq Platform.

Updates to supported software

HDL Coder has been tested with:

- Xilinx Vivado Design Suite 2016.4
- Intel Quartus Prime 16.1

See Supported Third-Party Tools and Hardware.

HDL Coder support packages renamed

- The HDL Coder Support Package for Altera FPGA Boards has been renamed to the HDL Coder Support Package for Intel FPGA Boards.
- The HDL Coder Support Package for Altera SoC Platform has been renamed to the HDL Coder Support Package for Intel SoC Devices.
- The HDL Coder Support Package for Xilinx Zynq-7000 Platform has been renamed to the HDL Coder Support Package for Xilinx Zynq Platform.

See also HDL Coder Supported Hardware.

R2017a

Version: 3.10

New Features

Bug Fixes

Compatibility Considerations

Model and Architecture Design

HDL Floating Point Operations Library: Easily find additional and existing single-precision floating-point blocks supported for HDL code generation

HDL Coder provides a **HDL Floating Point Operations** block library that consists of Simulink blocks configured for HDL code generation with the native floating-point support.

In R2017a, HDL Coder added native floating-point support for these blocks in the **HDL Floating Point Operations** Library.

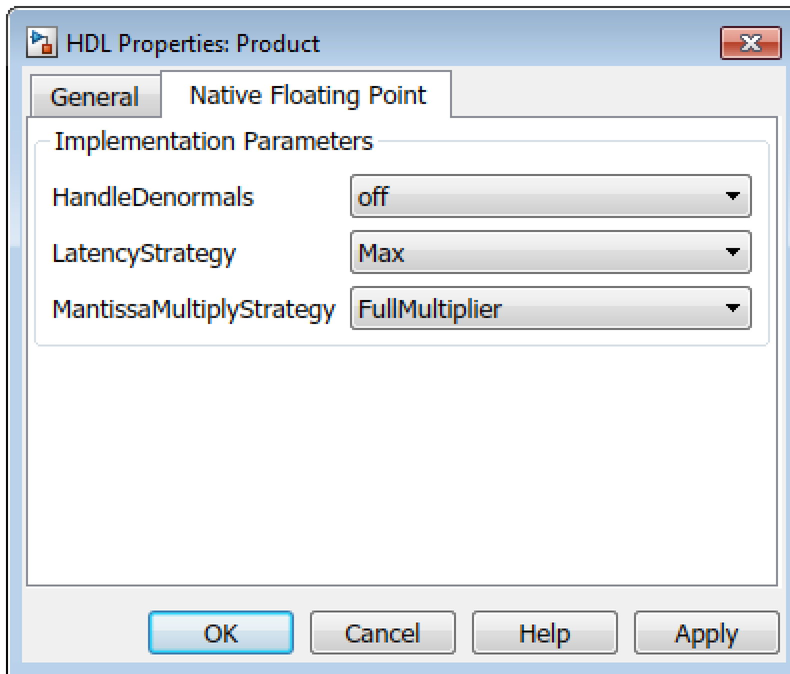
- Discrete FIR Filter with `Fully Parallel` as **HDL Architecture**
- Discrete-Time Integrator
- Discrete PID Controller
- Rounding Function
- Trigonometric Function with **Function** as `cos + jsin`
- Sign
- Math Function block with **Function** as:
  - `mod`
  - `rem`

See also HDL Floating Point Operations.

Floating-Point Latency Customization at Block-Level

For blocks that support code generation in native floating-point mode, you can now specify custom block-level settings. By default, the blocks in your design inherit the native floating-point settings that you specify in the Configuration Parameters dialog box. To specify custom settings for these blocks, right-click the block and open **HDL Code > HDL Block Properties**, and then select the **Native Floating Point** tab. Using custom settings for the blocks, you can optimize your design implementation on the target FPGA device for area and speed.

For most blocks, the **Native Floating Point** tab contains the **HandleDenormals** and **LatencyStrategy** settings. If there are multipliers in your design, you can specify how you want HDL Coder to implement the mantissa multiplication operation for individual blocks by using the **MantissaMultiplyStrategy** setting. This figure shows the HDL Block Properties dialog box for a Product block.



Additional Block and System Object Support with Native Floating Point

HDL Coder now supports these blocks and system objects with native floating point.

- All RAM blocks, which include:
  - Single Port RAM
  - Dual Port RAM
  - Simple Dual Port RAM
  - Dual Rate Dual port RAM
- Serializer1D and Deserializer1D
- `hdl.RAM`

See Also Operators and Simulink Blocks Supported for Native Floating-Point.

Custom reference model prefix specification

For module names or files that are generated for a model reference, you can now specify a custom reference model prefix. Previously, HDL Coder prefixed the referenced model with `modelname_`.

To add a prefix for the referenced model, in the HDL Block Properties dialog box, for **ReferenceModelPrefix**, specify the prefix as a text. When generating code, HDL Coder applies this prefix to the names of generated HDL files for submodels, package names, and HDL identifiers. By default, the prefix is the name of the top-level subsystem.

See also Model.

GenerateWebview parameter name changed to HDLGenerateWebview

To include a model Web view in the HDL Code Generation report programmatically, you now use the `HDLGenerateWebview` parameter. This parameter corresponds to the **Generate model Web view** setting on the **HDL Code Generation** pane in the Configuration Parameters dialog box. Previously, you used the `GenerateWebview` parameter.

HDL Coder now distinguishes the `HDLGenerateWebview` parameter from the `GenerateWebview` parameter that Embedded Coder® uses.

Compatibility Considerations

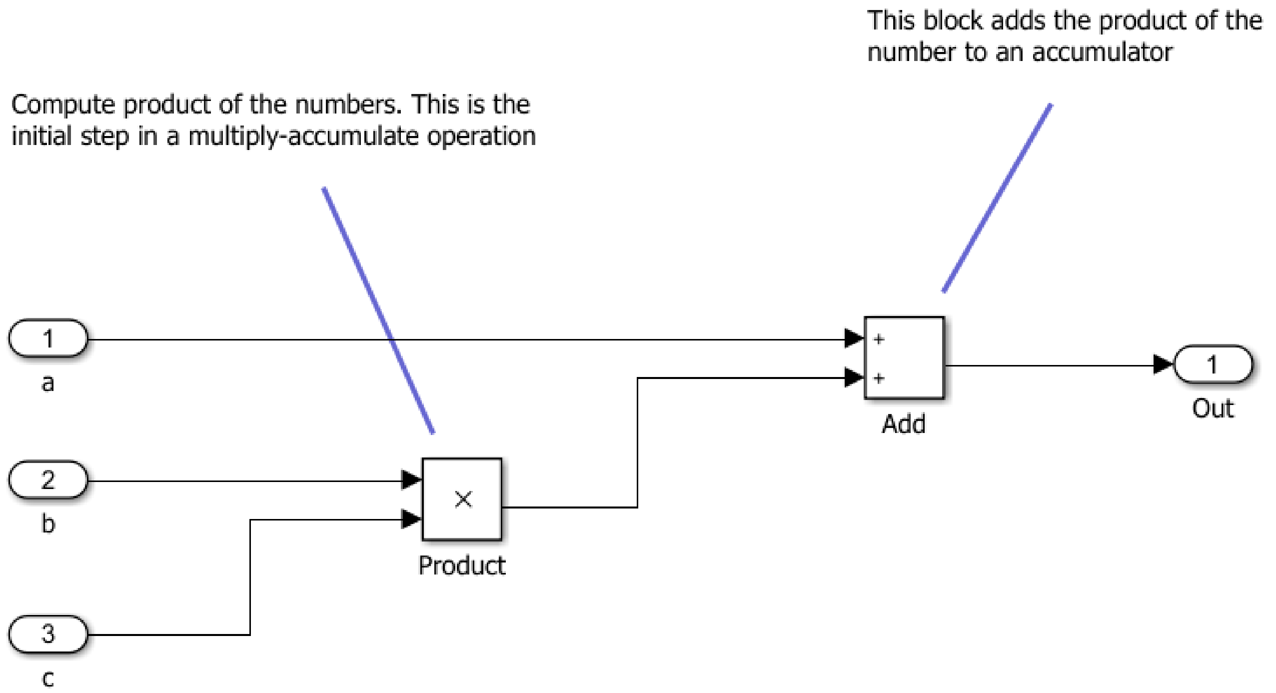
In R2017a, if you run a previously saved MATLAB script that used `hdlset_param` or `makehdl` with the `GenerateWebview` parameter, HDL Coder generates an error. To fix the error, change the parameter name to `HDLGenerateWebview`, and then run the script.

If you load a pre-R2017a model that was saved with the `GenerateWebview` parameter enabled, HDL Coder ignores the parameter setting. To generate the model Web view, enable the `HDLGenerateWebview` parameter.

Comments in HDL code for Simulink blocks with text annotations

If you add text annotations with connecting lines to Simulink blocks, HDL Coder generates comments in the HDL code for the blocks. The comments make it easier to map your algorithm in Simulink to the generated code.

Previously, the comments were grouped together on top of the process declaration statement in Verilog or the entity declaration statement in VHDL. For example, consider this Simulink model that performs a multiply-accumulate operation.



This table shows the generated Verilog code from the model for R2017a and earlier releases.

<table>
<tr><th>R2017a</th><th>Releases before R2017a</th></tr>
<tr><td>
<pre>
module MAC
          (a,
           b,
           c,
           Out);

  input   [15:0] a; // uint16
  input   [15:0] b; // uint16
  input   [15:0] c; // uint16
  output [31:0] Out; // uint32

  wire [31:0] Product_out1; // uint32
  wire [31:0] Add_1; // ufix32
  wire [31:0] Add_out1; // uint32

  // Initial product computation
  assign Product_out1 = b * c;

  // Adds product of the numbers to an accumulator
  assign Add_1 = {16'b0, a};
  assign Add_out1 = Add_1 + Product_out1;

  assign Out = Add_out1;

endmodule // MAC
</pre>
</td><td>
<pre>
module MAC
          (
           a,
           b,
           c,
           Out
          );

  input   [15:0] a; // uint16
  input   [15:0] b; // uint16
  input   [15:0] c; // uint16
  output [31:0] Out; // uint32

  wire [31:0] Product_out1; // uint32
  wire [31:0] Add_1; // ufix32
  wire [31:0] Add_out1; // uint32

  // Adds the product of the number to an
  //
  // Initial product computation

  assign Product_out1 = b * c;

  assign Add_1 = {16'b0, a};
  assign Add_out1 = Add_1 + Product_out1;

  assign Out = Add_out1;

endmodule // MAC
</pre>
</td></tr>
</table>

See also Generate Code with Annotations or Comments.

Block Enhancements

For Each Subsystems: Reduce block replication and improve code reuse in HDL-targeted designs

HDL Coder supports the For Each Subsystem block for code generation. The block:

- Supports vector processing that enables you to process individual elements or subarrays of an input signal simultaneously. You no longer need to split the signals or create and connect replicas of a subsystem to model the same algorithm.
- Improves readability of code by using a `for-generate` loop in the generated HDL code that iterates through each element of the input signal. The elements can be scalars or subarrays of the input signal. The for-generate loop is cleaner and reduces the number of lines of code, which can otherwise result in hundreds of lines of code for large vector signals.
- Supports HDL code generation for all data types, Simulink blocks, and predefined and user-defined system objects.
- Supports optimizations on and inside the block, such as resource sharing and pipelining. The parallel processing capability of the For Each Subsystem block combined with the optimizations that you specify provides high performance on the target FPGA device.

The For Each Subsystem block is available as part of the **Ports & Subsystems** block library in HDL Coder.

For an example that shows how to generate HDL code for the For Each Subsystem, see Generate HDL Code for Blocks Inside For Each Subsystem.

HDL Optimized Filters: Model and generate optimized hardware implementations for FIR filters (requires DSP System Toolbox)

This release introduces the Discrete FIR Filter HDL Optimized block and `dsp.HDLFIRFilter` System object, which model FIR filter structures optimized for HDL code generation. The filter is sample-based. Control signals are provided for flow control. Resource sharing options allow tradeoffs between throughput and resource utilization. The block and object provide cycle-accurate models of the generated HDL code, including clock rates and latency.

HDL Channelizer Block and System Object: Isolate narrowband channels from a wideband signal and generate HDL with efficient multiplier usage (requires DSP System Toolbox)

This release introduces the Channelizer HDL Optimized block and `dsp.HDLChannelizer` System object, which model a polyphase filter bank and fast Fourier transform and support HDL code generation. The algorithm provides an efficient hardware implementation and hardware-friendly control signals. You can achieve giga-sample-per-second (GSPS) throughput with vector input.

Gigasample per Second (GSPS) Signal Processing: Increase throughput of FIR decimation algorithms by using frame input

You can now generate HDL code from the FIR Decimation block when the block uses frame input. The block accepts a column vector of input data. Each element of the vector represents a sample in time.

The coder implements a parallel HDL architecture for the filter. This capability increases throughput in hardware designs. To configure the block for frame input:

1. Connect a column vector signal to the FIR Decimation block input port.
2. Specify **Input processing** as `Columns as channels (frame based)`.
3. Set **Rate options** to `Enforce single-rate processing`.
4. Right-click the block and open **HDL Code > HDL Block Properties**. Set the **Architecture** to `Frame Based`. The block implements a parallel HDL architecture. See Frame-Based Architecture.

Enhancements to MATLAB Function block support in synchronous subsystems

For a MATLAB Function block inside a synchronous subsystem, you can now use the combinational and sequential logic portions in one MATLAB function. Previously, you created two separate MATLAB Function blocks, one for the combinational logic, and the other for the sequential logic.

To use the combinational and sequential logic portions inside one MATLAB Function block, in the Ports and Data Manager dialog box, select the **Allow direct feedthrough** check box. The output function can then depend on inputs and persistent variables. For example, you can now use this MATLAB function that has two outputs, with one output depending on the input, and the other output depending on a persistent variable.

```
function [y1, y2] = fcn(u, v)

persistent p;
if isempty(p)
    p = uint8(0);
end

y1 = p;
y2 = v;

p = u;
```

Using the MATLAB Function block inside a synchronous subsystem generates cleaner HDL code and uses fewer hardware resources. See also State Control.

HDL Coder support for blocks that support bus signal treated as vector

HDL Coder can now generate code for blocks that support the **Bus signal treated as vector** setting. These blocks are not bus-capable, but they can accept a vector signal. Previously, to generate HDL code for these blocks, you used a Bus to Vector block to convert the bus signals to vectors for input to the block.

When you use these blocks with buses, ensure that:

- Input to the blocks is a virtual bus.
- Constituent signals of the bus have the same attributes.

- **Bus signal treated as vector** is set to `none` or `warning`. This setting is available in the **Diagnostics > Connectivity** pane in the Configuration Parameters dialog box. See also Bus signal treated as vector (Simulink).

HDL code generation support for Bus Assignment block with nonvirtual bus

In your Simulink model, you can now use Bus Assignment blocks with nonvirtual bus signals for HDL code generation. Previously, when you used Bus Assignment blocks, to generate HDL code, you converted the nonvirtual buses to virtual buses.

Additional HDL Coder bus support

HDL Coder now supports the:

- Ground block with bus input
- Constant block with a value of `0` and `bus` as **Output data type**

HDL code generation support for System Objects with enumeration types

You can now generate HDL code for System objects with enumeration types:

- From MATLAB, by using the **HDL Coder** app or MATLAB to HDL Workflow Advisor
- From Simulink, for System objects that are used with MATLAB System or MATLAB Function blocks

You can also use array of enumeration types and enumeration types as fields in Simulink bus data types for HDL code generation.

Code Generation and Verification

Native Floating-Point Testbench: Generate SystemVerilog DPI, cosimulation, and FPGA-in-the-loop test benches with single-precision data types (requires HDL Verifier)

Before generating code, if you enable the native floating-point support, you can now verify the HDL implementation of your design by using any of these testbenches:

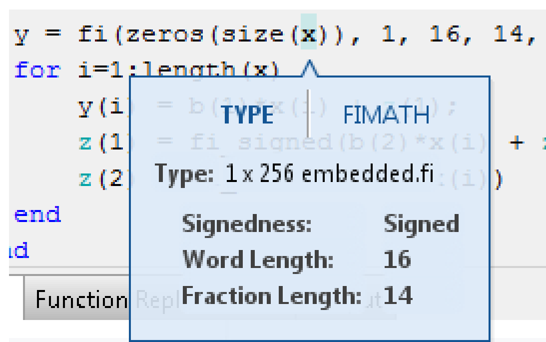
- SystemVerilog DPI test bench
- Cosimulation
- FPGA-in-the-loop

See also Verify the Generated Code from Native Floating-Point.

More fixed-size variable information in Fixed-Point Conversion step of HDL Coder App

In R2017a, when you convert floating-point MATLAB code to fixed-point MATLAB code, the target interface step provides information about the converted fixed-point code. Previously, in the target interface step, the app displayed the original floating-point MATLAB code.

In the **Fixed-Point Conversion** step, after fixed-point conversion, if you place your cursor over a converted variable or expression, the app displays the fixed-point type information.



For a variable with a fixed-point type in the original code, when you place your cursor over the variable before or after conversion, the app displays the fixed-point type information.

Comments in generated HDL code for MATLAB System blocks

In R2017a, if you have comments in your MATLAB System block, HDL Coder passes these comments to the generated HDL code. These comments make it easier to map your algorithm in MATLAB to the generated HDL code.

Global reset signals minimization in generated HDL Code

In R2017a, you can use the **Minimize global resets** setting to minimize or remove global reset signals in the generated HDL code. To specify this setting, in the Configuration Parameters dialog

box, on the **HDL Code Generation** > **Global Settings** > **Ports** tab, select the **Minimize global resets** check box.

See also `MinimizeGlobalResets`

HDL code generation support for DUT subsystem with custom HDL properties

For any DUT subsystem, you can now specify custom HDL block property settings and generate HDL code. To specify these custom settings, right-click the subsystem and select **HDL Code** > **HDL Block Properties**.

Previously, to generate HDL code for any DUT subsystem, the subsystem used the default HDL block property settings.

Changes in HDL Code Generation Panel in Configuration Parameters Dialog Box

Parameters Added

- `Minimize global resets` in the **Global Settings** > **Ports** tab
- In the **Test Bench** pane `Test Bench Generation Output` section:
  - `SystemVerilog DPI test bench`
  - `HDL code coverage` check box

Parameters Moved

These parameters moved from the **Target and Optimizations** > **General** tab to the **Global Settings** > **Ports** tab:

- `Minimize clock enables`
- `Use Trigger as clock`

Parameters Removed

`Cosimulation model for use with:`

Syntax Highlighting of Generated HDL Code in HTML Report

In R2017a, enhancements in syntax highlighting greatly improve the readability of the generated HDL code. To see the syntax highlighting, before generating HDL code, in the Configuration Parameters dialog box, on the **HDL Code Generation** pane, select the **Generate traceability report** check box.

Previously, the Code Generation report used two highlighting colors: blue for code and green for comments. With this change, HDL-specific keywords are highlighted in blue and the rest of the HDL code is in black. Comments in the code are still highlighted in green.

Speed and Area Optimizations

Improvements to delay balancing in multirate regions

If you have local multirate regions in your design or introduce them through clock-rate pipelining, HDL Coder improves delay optimization in these regions by reducing excessive matching delays. This optimization results in area, timing, and power-efficient designs, particularly in cases with significant rate differences.

Functionality Being Removed or Changed

Functionality	Result	Use Instead	Compatibility Considerations
`MaxComputationLatency`	Still runs. The code generator displays a warning.	`Oversampling` with clock-rate pipelining.	Replace all instances of `MaxComputationLatency` with `Oversampling`.
`MaxOversampling`	Still runs. The code generator displays a warning.	`Oversampling` with clock-rate pipelining.	Replace all instances of `MaxOversampling` with `Oversampling`.

IP Core Generation and Hardware Deployment

Data Type Support for AXI4 Slave: Map floating-point signals and vector signals to AXI4 slave interfaces in IP core generation

When using the `IP Core Generation` workflow, in the **Target platform interface table**, you can map single and vector signals at the DUT ports to `AXI4` or `AXI4-Lite` interfaces.

The workflow for vector signals includes an **IP Core Generation** report that displays:

- Address offsets of AXI4 interface accessible registers generated for each input, output, and strobe signal in the **Register Address Mapping** section. The additional strobe register for each input and output vector data maintains the synchronization across multiple, sequential AXI4 read and write operations. For each input and output vector signal, this section displays the size of the vector, and the starting and ending address offsets.
- A **Vector Data Read/Write with Strobe Synchronization** subsection in the **IP Core User's Guide** section that shows how HDL Coder handles vector data and synchronizes read and write operations across the AXI4 interface.

See also Custom IP Core Generation.

When you use signals that have a single data type, specify the floating-point library. In the Configuration Parameters dialog box, on the **HDL Code Generation** > **Global Settings** > **Floating Point Target** tab, for **Library**, choose `Altera Megafunctions (Altera FP Functions)`, `Altera Megafunction (ALTFP)`, `Native Floating Point`, or `Xilinx LogiCORE`.

Incremental Vivado Synthesis: Enable IP caching for faster synthesis of Xilinx Vivado reference designs

When creating a Xilinx Vivado project with the `IP Core Generation` workflow, you can enable IP caching to speed up synthesis of the reference design. To accelerate reference design synthesis when using the workflow for the second time:

1. Before running the **Create Project** task the first time, select the **Enable IP caching** check box. When running this task, the workflow creates an empty IP cache folder.
2. Run the **Build FPGA Bitstream** task to populate the IP cache folder with synthesis logs and design checkpoint files generated for the HDL IP core and other IP blocks in the reference design.
3. To accelerate reference design synthesis, run the **Build FPGA Bitstream** task a second time. Make sure that you use the same `hdl_prj` folder as the first time you ran the workflow.

   The Xilinx Vivado tool then checks and reuses the design checkpoint files in the IP cache, which speeds up reference design synthesis.

If you are using your own custom reference design, you can accelerate reference design synthesis when running through the workflow the first time.

1. In the IP cache folder, delete the IP core files generated for the DUT. Extract the remaining files in this folder into a zip file, name it `ipcache.zip`, and save the file in the reference design folder.

2 Make sure that the IPCacheZipFile property of the hdlcoder.ReferenceDesign class points to the ipcache.zip folder.

In the workflow, HDL Coder uses the files in this IP cache, which speeds up reference design synthesis. For more information, see IP Caching for Faster Reference Design Synthesis.

IP core generation support for Altera Megafunction

When mapping your floating-point algorithm in Simulink to Altera Megafunction floating-point IP, you can now generate an HDL IP core with the IP Core Generation workflow. To use this workflow, map your Simulink model to Altera Megafunctions (Altera FP Functions) or Altera Megafunctions (ALTFP) floating-point libraries.

To specify the floating-point library, in the Configuration Parameters dialog box, on the **HDL Code Generation** > **Global Settings** > **Floating Point Target** tab, for **Library**, choose Altera Megafunctions (Altera FP Functions) or Altera Megafunctions (ALTFP).

Custom IP repository specification

With the IP Core Generation workflow, by using the addIPRepository method of the hdlcoder.ReferenceDesign class, you can add your own custom IP repository to your custom reference design.

Previously, to add IP modules, you used the CustomFiles property of the hdlcoder.ReferenceDesign class. Starting in R2017a, you can still use the CustomFiles property, but it is recommended to use the addIPRepository method instead. Using this method, you can include IP from a shared repository folder, or include multiple repository folders in your reference design.

See also Define and Add IP Repository to Custom Reference Design.

Xilinx Virtex-2 FPGA board support being removed

HDL Coder no longer supports the Xilinx ISE 10.1 synthesis tool and target platforms that use the Xilinx Virtex-2 FPGA board. For example, Speedgoat IO314 and older Speedgoat boards use Xilinx Virtex-2 FPGA, and are no longer supported with the Simulink Real-Time FPGA I/O workflow.

Compatibility Considerations

If you load a pre-R2017a model that was saved with a target platform that used the Xilinx Virtex-2 FPGA, and then open the HDL Workflow Advisor, HDL Coder generates a warning. To avoid this warning, use a newer FPGA board and synthesis tool. With the Simulink Real-Time FPGA I/O workflow, use the Speedgoat IO321 or a later Speedgoat board.

Updates to supported software

HDL Coder has been tested with:

- Xilinx Vivado Design Suite 2016.2
- Altera Quartus II 16.0

See Supported Third-Party Tools and Hardware.

R2016b

Version: 3.9

New Features

Bug Fixes

Compatibility Considerations

Model and Architecture Design

Native Floating Point: Generate target-independent synthesizable RTL from single-precision floating-point models

In R2016b, if you use single-precision data types in your Simulink model, HDL Coder can generate target-independent HDL code without converting to fixed point. You can deploy the generated code on any generic ASIC or FPGA platform.

In your Simulink model:

- You can have a combination of integer, fixed-point, and floating-point operations. By using Data Type Conversion blocks, you can perform conversions between single-precision and fixed-point data types.
- If your design does not use denormal numbers, you can specify that HDL Coder does not have to add the additional logic to check for denormal numbers, which improves area on the target hardware platform.
- By using the latency strategy setting, customize the latency of the native floating-point library.

The generated code:

- Complies with the IEEE-754 standard of floating-point arithmetic.
- Does not require floating-point processing units or hard floating-point DSP blocks on the target ASIC or FPGA.

When you verify the generated code, use HDL testbench to check for floating-point tolerance based on `relative error` or `ulp error`, and to ensure accuracy of your design with Simulink.

To specify native floating-point support, in the Configuration Parameters dialog box, on the **HDL Code Generation > Global Settings > Floating Point Target** tab, for **Library**, specify **NATIVE FLOATING POINT**.

For more information, see Native Floating Point.

HDL Coder support for tunable parameters in data dictionary

Starting with R2016b, you can manage and define tunable parameters in a Simulink data dictionary for HDL code generation.

To learn more about data dictionary, see What Is a Data Dictionary?.

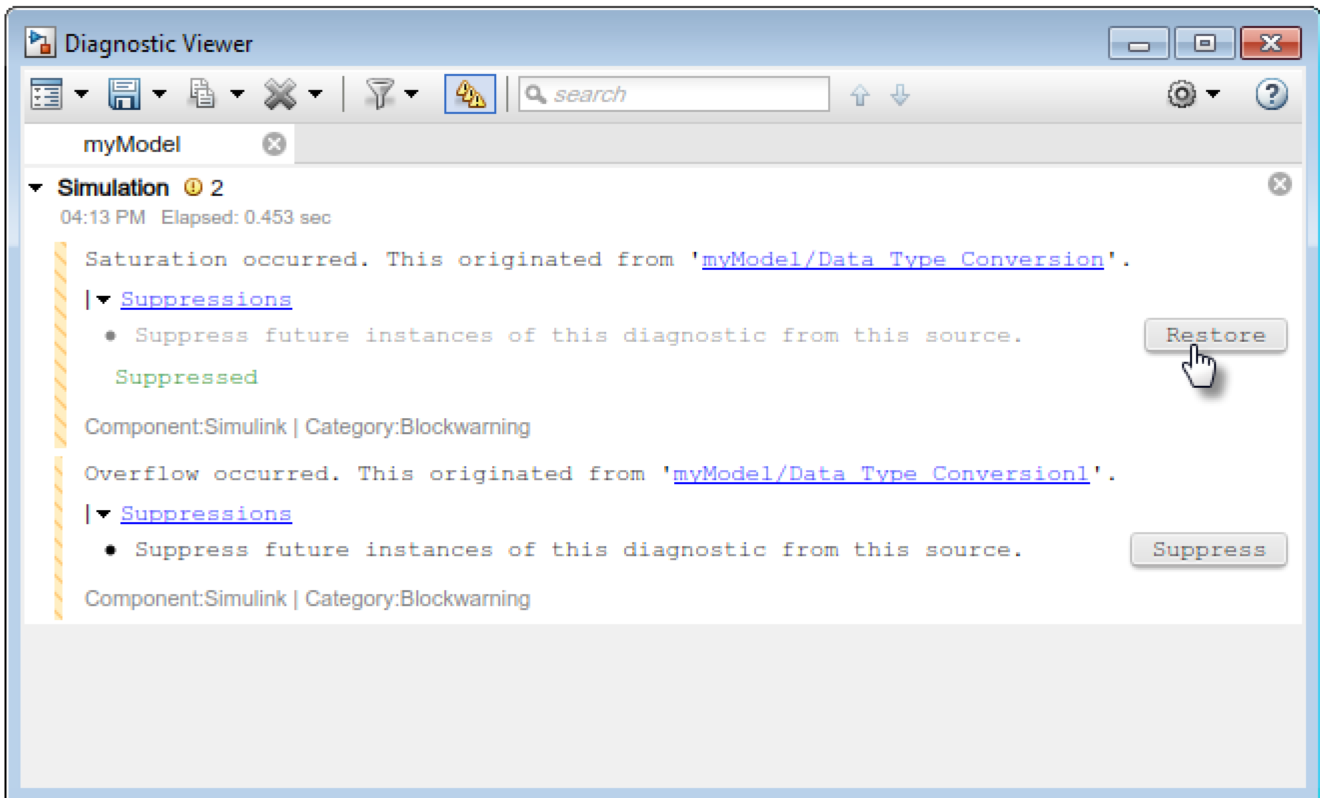
Generic ports for DUT mask parameters

In R2016b, HDL Coder supports mask parameters at the DUT as generic ports for HDL code generation.

Simulink diagnostic suppressor option

The Diagnostic Viewer in Simulink now includes an option to suppress certain diagnostics. You can suppress warnings for specific objects in your model. In the Diagnostic Viewer, click the **Suppress**

button next to the warning to suppress the warning from the specified source. You can restore the warning from the source by clicking **Restore**.



You can also control the suppressions from the command line. To view the existing suppressions on the model, use the `Simulink.getSuppressedDiagnostics` function.

```
Suppressed = Simulink.getSuppressedDiagnostics('myModel')

Suppressed =

  SuppressedDiagnostic with properties:

             Source: 'myModel/Data Type Conversion'
                 Id: 'SimulinkFixedPoint:util:Saturationoccurred'
     LastModifiedBy: ''
           Comments: ''
       LastModified: '2016-Apr-26 10:31:22'
```

Suppress diagnostics by using the `Simulink.suppressDiagnostic` function.

```
Simulink.suppressDiagnostic('myModel/Data Type Conversion1', ...
                'SimulinkFixedPoint:util:Overflowoccurred')
Suppressed

Suppressed =

  1×2 SuppressedDiagnostic array with properties:

    Source
    Id
```

```
    LastModifiedBy
    Comments
    LastModified
```

Restore a diagnostic by using the `Simulink.restoreDiagnostic` function.

```
Simulink.restoreDiagnostic('myModel/Data Type Conversion1', ...
'SimulinkFixedPoint:util:Overflowoccurred')
```

Block Enhancements

Gigasample Per Second (GSPS) Signal Processing: Increase throughput of HDL code generated from Discrete FIR Filter and Integer Delay blocks by using frame input

You can now generate HDL code from the Discrete FIR Filter block when using frame input. Set **Input processing** to `Columns as channels (frame based)`. Then, right-click the block and open **HDL Code** > **HDL Block Properties**. Set the **Architecture** to `Frame Based`. The block accepts vector input data, where each element of the vector represents a sample in time. The coder implements a parallel HDL architecture for the filter. See Discrete FIR Filter.

The Delay block also supports HDL code generation with frame input data. Set **Input processing** to `Columns as channels (frame based)`. The block accepts vector input data, where each element of the vector represents a sample in time.

This capability increases throughput in hardware designs.

Bit-reversed input order for HDL-optimized FFT

For vector input data, the HDL-optimized FFT now supports bit-reversed input with natural order output. For scalar input data, you can select any input order with any output order. The default is natural order input with bit-reversed output.

This change affects these blocks and System objects:

- FFT HDL Optimized
- IFFT HDL Optimized
- `dsp.HDLFFT`
- `dsp.HDLIFFT`

High-throughput polyphase filter bank for HDL example

The Generate HDL Code for High Throughput Signal Processing example shows how to design a Polyphase Filter Bank to achieve gigasample per second (GSPS) data rates in the generated HDL implementation. The model uses the FFT HDL Optimized block with vector input.

HDL support for reset port on Discrete FIR Filter

You can now generate HDL code from the Discrete FIR Filter block when you configure the block to have an external reset port.

HDL Coder support for array of buses

In your Simulink model, you can now use an array of buses for HDL code generation.

When your Simulink model uses an array of buses, in the generated code, HDL Coder expands the array of buses into the corresponding scalars. For more information, see Generating HDL Code for Subsystems with Array of Buses.

To learn more about array of buses and supported blocks, see Combine Buses into an Array of Buses.

Synchronous behavior for Resettable Subsystem with State Control block

You can specify synchronous hardware behavior and generate cleaner HDL code for a Resettable Subsystem with the State Control block. If you specify synchronous hardware behavior, the HDL code uses fewer hardware resources, because HDL Coder does not generate bypass registers.

The Resettable Synchronous Subsystem block is now available as part of the HDL Subsystems block library in HDL Coder. The Resettable Synchronous Subsystem block uses the synchronous hardware behavior of the State Control block with the Resettable Subsystem block.

For an example that shows how to use the Resettable Synchronous Subsystem block, see Resettable Subsystem Support in HDL Coder™.

HDL optimized Sine and Cosine blocks

In the Lookup Tables block library in HDL Coder, the Sine HDL Optimized and Cosine HDL Optimized blocks replace the Sine and Cosine blocks respectively. You can still use the Sine and Cosine blocks from the Lookup Tables block library in Simulink for HDL code generation.

The new blocks are optimized for area and speed because you can configure them with Lookup Tables that have an exact power of two as its number of elements. In the generated code, the Lookup Tables precede a register without reset so that they map efficiently to RAM blocks on the target hardware platform.

Simpler method to call System objects

You can now call a System object with arguments, as if it were a function, instead of using the `step` method to perform the operation defined by the object. The `step` method continues to work. This capability improves the readability of scripts and functions that use many different System objects.

For example, if you create a `hdl.RAM` System object named `ramsingle`, and then call the System object as a function with that name:

```
ramsingle = hdl.RAM('RAMType','Single port', ...
    'WriteOutputValue','Old data');
ramsingle(x)
```

The equivalent operation with the `step` method is:

```
ramsingle = hdl.RAM('RAMType','Single port', ...
    'WriteOutputValue','Old data');
step(ramsingle,x)
```

When the `step` method has the System object as its only argument, the function equivalent has no arguments. You must call this function with empty parentheses. For example, `step(sysobj)` and `sysobj()` perform equivalent operations.

Code Generation and Verification

Logic Analyzer: Visualize, measure, and analyze transitions and states over time for Simulink signals

If you have DSP System Toolbox, by using the **Logic Analyzer** visualization tool, you can view the transitions of signals. Use the **Logic Analyzer** to:

- Debug and analyze models.
- Trace and correlate many signals simultaneously.
- Detect and analyze timing violations.
- Trace system execution.

HDL Coder support for creating and attaching configuration sets

Starting in R2016b, HDL Coder supports configuration set management workflow on the Model Explorer, or from the command line. You can create an active configuration set with the preferred HDL Configuration Parameters on a standard Simulink model, and export and copy this configuration set for Simulink models that you create.

Previously, you created a configuration set for each Simulink model, and ensured that the configuration set had similar contents for all your Simulink models.

VHDL Architecture Name available in Configuration Parameters dialog box

You can now specify the **VHDL architecture name** in the Configuration Parameters dialog box.

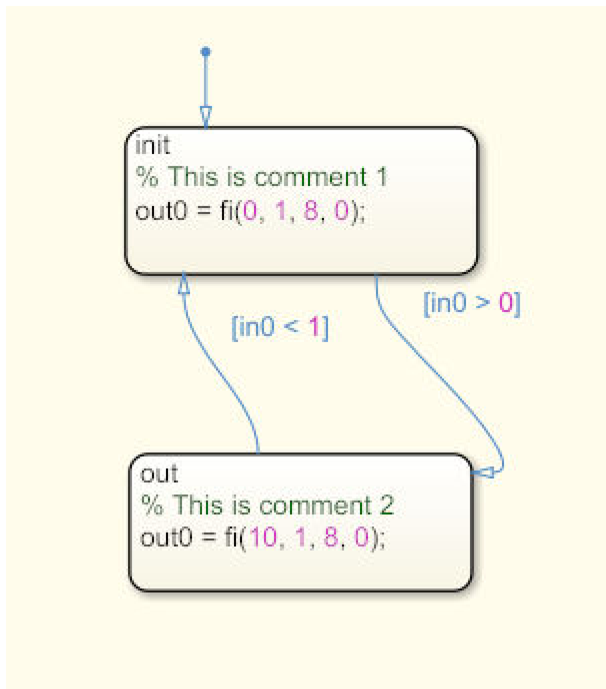
- **Commonly Used Parameters** tab: **HDL Code Generation** > **Global Settings** > **General** tab.
- **All Parameters** tab: Search for `VHDLArchitectureName`.

RAM with generic ports enhancement

Starting in R2016b, when generating code for RAM blocks in your Simulink model, HDL Coder adds parameters in Verilog and generics in VHDL for the RAM address and data widths. This means that HDL Coder generates only one generic RAM file for RAM blocks that differ in address widths, data widths, or both.

Stateflow comments generated as comments in HDL

When your Simulink model contains a Stateflow Chart that uses comments, HDL Coder generates comments in the HDL code. For example, consider this `Moore` Stateflow Chart in your Simulink model.



When you generate Verilog code from the model, HDL Coder displays the comments in the Stateflow Chart inline beside the corresponding Stateflow object.

<table>
<tr><th>R2016b</th><th>R2016a</th></tr>
<tr><td><pre>always @(is_AL_Chart) begin
   out0_1 = 8'sb00000000;
   case (is_AL_Chart)
     is_AL_Chart_IN_init :
       begin
         // This is comment 1
         out0_1 = 8'sb00000000;
       end
     default :
       begin
         // This is comment 2
         out0_1 = 8'sb00001010;
       end
   endcase
 end</pre></td><td><pre>always @(is_AL_Chart) begin
   out0_1 = 8'sb00000000;
   case (is_AL_Chart)
     is_AL_Chart_IN_init :
       begin
         out0_1 = 8'sb00000000;
       end
     default :
       begin
         out0_1 = 8'sb00001010;
       end
   endcase
 end</pre></td></tr>
</table>

Tolerance check for floating-point libraries

When mapping your Simulink model to floating-point libraries, you can specify the tolerance check when you generate the testbench.

For operators such as trigonometric sine and cosine, there can be small rounding errors or numeric differences with the correct rounding range of values that the IEEE-754 standard specifies. To check for numerical accuracy in the generated testbench by using HDL testbench, specify the floating-point tolerance check.

You can perform the floating-point tolerance check based on the `relative error` or `ulp error`.

- relative error: Relative error is the rounding error when approximating a nonzero real number. By default, the tolerance value is 1e-07. You can specify a tolerance value less than or equal to 1e-07.
- ulp error: ulp (unit in the last place) is the gap between two floating-point numbers nearest *x*, even if *x* is one of the numbers. By default, the tolerance value is zero. You can specify a tolerance value greater than or equal to zero.

To check for floating-point tolerance, in the Configuration Parameters dialog box, on the Configuration section of **HDL Code Generation** > **Test Bench** tab, for **Floating point tolerance check based on**, specify relative error or ulp error, and enter the **Tolerance Value**.

For more information, see FPToleranceValue and FPToleranceStrategy.

Code Generation Report enhancements

Delay Balancing Report

HDL Coder now displays the path delay balancing information in the Delay Balancing section of the Optimization Report. Previously, the Optimization Report displayed the delay balancing information in the Path Delay Summary subsection of the Streaming and Sharing report.

See also Optimization Report.

Shift Operators in Resource Report

The High-level Resource Report Summary shows the number of Static Shift operators and Dynamic Shift operators. The Detailed Report shows the number of static left shift, static right shift, dynamic left shift, and dynamic right shift operators.

Comprehensive documentation for HDL coding standard rules

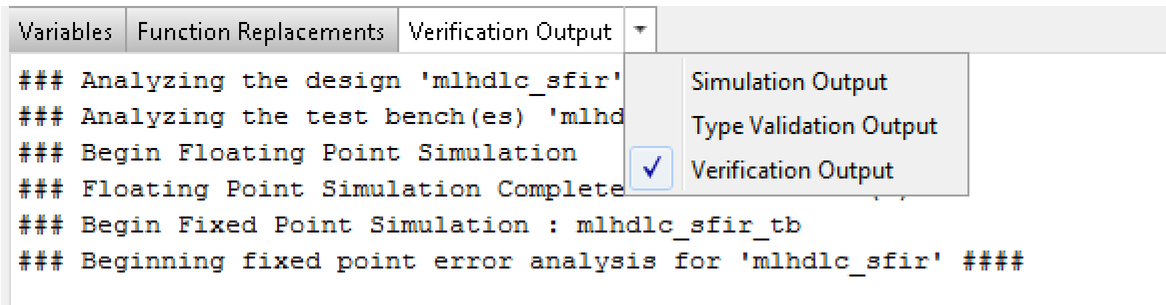
The HDL Coder documentation provides a comprehensive list of coding standard rules with recommendations for each of the rules. The coding standard rules fall under three categories:

- Basic Coding Practices: Checks for conformance of modeling constructs with general naming conventions and basic coding guidelines. See Basic Coding Practices.
- RTL Description Techniques: Checks for conformance with RTL description rules and guidelines. See RTL Description Techniques.
- RTL Design Methodology Guidelines: Includes guidelines for creating and using function libraries, and test facilitation design. See RTL Design Methodology Guidelines.

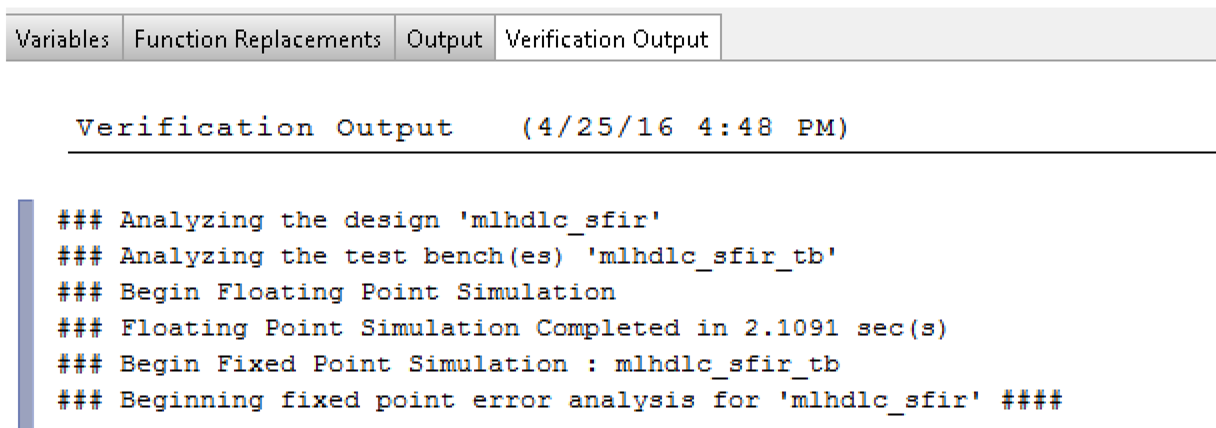
To learn more about HDL coding standards, see HDL Coding Standards and HDL Coding Standard Report.

More discoverable logs and reports for fixed-point conversion in HDL Coder app

Previously, in the HDL Workflow Advisor **Fixed-Point Conversion** task, the **HDL Coder** app displayed logs and report links for range analysis, fixed-point conversion, and verification on separate tabs that were placed on top of each other. To see a hidden tab, you opened a menu and selected the tab.



In R2016b, the app displays logs and report links for range analysis and fixed-point conversion on the **Output** tab. It displays logs and report links for verification on the **Verification Output** tab. These tabs are next to each other so that you can more easily find them.



Enhancements in generated model for Lookup Tables

In R2016b, you can flatten masked subsystems and library blocks that contain Lookup Tables to enable further optimizations and file reduction.

Target and Optimizations pane in HDL Coder Configuration Parameters

In the Configuration Parameters dialog box, on the **HDL Code Generation** pane, HDL Coder has a new **Target and Optimizations** pane where you can specify the target device and optimization settings.

- Parameters that were previously in the **General**, **Pipelining**, and **Resource Sharing** sections of the **Optimization** tab of **Global Settings** pane have moved to a **General** tab, a **Pipelining** tab, and a **Resource Sharing** tab respectively in the **Optimizations** section of **Target and Optimizations** pane.
- In the **Target and Optimizations** pane, you now specify the target device settings in the **Tool and Device** section and the **Target Frequency** in the **Objectives** section respectively.

For more information, see HDL Code Generation Pane: Target and Optimizations.

Link to Code Generation Report after HDL code generation

In the Configuration Parameters dialog box, on the **HDL Code Generation** pane, when you select **Generate resource utilization report** and generate HDL code, HDL Coder displays a link to the Code Generation report. If you happen to close the report after code generation, you can click the link to open the report from the MATLAB Command Window.

Speed and Area Optimizations

Adaptive Pipelining: Specify synthesis tool and target clock frequency for automatic pipeline insertion and balancing

You can now specify adaptive pipelining for your Simulink model, or for an individual subsystem in your Simulink model, to improve area and timing on the target FPGA device. To insert adaptive pipelines, specify the synthesis tool and the target frequency. HDL Coder inserts the required number of pipelines for potential area and timing improvements for these blocks:

- Lookup Table
- Product, Gain, and Multiply-Add
- Rate Transition and Downsample

You can enable adaptive pipelining by using the `AdaptivePipelining` property from the command line, or by using the **AdaptivePipelining** HDL block property for the Subsystem. See also Adaptive Pipelining.

HDL Coder displays a report that shows the adaptive pipelining status and whether HDL Coder successfully inserted pipeline registers. For more information, see Optimization Report.

Clock-rate pipelining enhancements

Subsystem level control of clock-rate pipelining

You can now specify clock-rate pipelining for individual subsystems in your Simulink model. With this optimization, you can selectively apply clock-rate pipelining to subsystems in your model design that are on the critical path, and improve timing.

To disable clock-rate pipelining for an individual subsystem, in HDL Block Properties for the subsystem, set **ClockRatePipelining** to `off`.

To learn more about the HDL block property, see ClockRatePipelining.

Optimization of Downsample block with nonzero offset

In R2016b, when you have a Downsample block with a nonzero **Sample offset** at the boundary of a clock-rate pipelining region, HDL Coder does not introduce the additional latency or generate a Rate Transition block. This optimization improves timing and area.

For more information, see Clock-Rate Pipelining.

Resource sharing enhancements

Sharing of multipliers with different word-lengths

HDL Coder now shares Product blocks and Gain blocks in your Simulink model that have different word-lengths. This optimization shares more multipliers, which saves area on the target platform.

To share multipliers that have different word-lengths, in the Configuration Parameters dialog box, on the **HDL Code Generation > Target and Optimizations > Resource Sharing** tab, specify the

Multiplier promotion threshold. The **Multiplier promotion threshold** is the maximum word-length by which HDL Coder promotes a multiplier for sharing with other multipliers.

Previously, for successful resource sharing, you used Product blocks or Gain blocks with the same word-length.

See also `MultiplierPromotionThreshold` and Resource Sharing.

Sharing of floating-point IP

HDL Coder now shares floating-point IP blocks in the target hardware based on the `SharingFactor` that you specify. This optimization saves area on the target hardware by sharing more floating-point IP blocks.

If you do not want to share floating-point IP blocks, in the Configuration Parameters dialog box, on the **Resource Sharing** tab, clear **Floating-point IPs**.

See also `ShareFloatingPointIP`.

Delay balancing failures reported as errors

Starting in R2016b, if delay balancing is unsuccessful, HDL Coder generates an error. To see the block or subsystem in your Simulink model that caused delay balancing to fail, in the **Delay Balancing** section of the Optimization report, click the link to that block or subsystem.

Compatibility Considerations

Previously, HDL Coder reported delay balancing failures as warnings. Now, if you load a pre-R2016b model for which delay balancing was unsuccessful, HDL Coder generates an error.

To learn more about possible reasons for delay balancing to fail, see Delay Balancing.

Optimization of Delay blocks with nonzero initial condition

In the generated code, HDL Coder now replaces a Delay block that has nonzero initial condition in your Simulink model with a Delay block that has zero initial condition and some additional logic. With this replacement, optimizations such as sharing, distributed pipelining, and clock-rate pipelining can work more effectively, and prevent an assertion from being triggered in the validation model.

To disable this optimization, in the Configuration Parameters dialog box, on the **HDL Code Generation** > **Target and Optimizations** > **General** tab, clear **Transform non zero initial value delay**.

For more information, see `TransformNonZeroInitDelay`.

Initialization script specification for Delay blocks without reset

Starting with R2016b, to initialize the registers, you can use the **no-reset registers initialization** setting to specify `Generate an external script`, `Do not initialize`, or `Generate initialization inside module`. When you select `Generate initialization inside module`, in Verilog, HDL Coder initializes the registers by using an initial block in each module. In VHDL, HDL Coder initializes the registers as part of the signal declaration statements.

Previously, if you had Delay blocks in your Simulink model with `ResetType` set to `None`, HDL Coder generated an external script to initialize the Delay blocks for ModelSim simulation.

The **no-reset registers initialization** setting is available in the Configuration Parameters dialog box, on the **HDL Code Generation** > **Global Settings** > **Coding style** tab.

To learn more, see `NoResetInitializationMode`.

IP Core Generation and Hardware Deployment

AXI4-Stream Video Interface: Generate HDL code with the AXI4-Stream Video interface by using the IP core generation workflow

In R2016b, when your synthesis tool is Xilinx Vivado, HDL Coder can generate an IP core with an AXI4-Stream Video interface for your video algorithm. To generate an IP core, model your video algorithm by using the streaming pixel protocol. Then, in the **Target platform interface table** map the pixel data and pixel control bus ports to the `AXI4-Stream Video Master` or `AXI4-Stream Video Slave` interfaces.

You can integrate the generated IP core into the `Default video system` reference design or your own custom video reference design.

For more information, see Model Design for AXI4-Stream Video Interface Generation.

Customizable FPGA floating-point target configuration

You can customize the floating-point target IP settings by using the floating-point target configuration that you specify for the library. When you customize the IP settings, you can choose from different combinations of IP names and data types, and specify the latency or the target frequency that you want the IP to achieve. You can customize the IP settings from:

- **Floating Point Target** tab in Configuration Parameters dialog box: When you specify an Altera or Xilinx FPGA floating-point library, specify your custom settings in the **IP Settings** section.
- Command-line interface: By using the `hdlcoder.createFloatingPointTargetConfig` class, you can create a floating-point target configuration object for a given FPGA floating-point library or the HDL Coder native floating-point. In the `IPConfig` property of this object, use the `customize` function to customize the IP settings.

For more information, see Customize Floating-Point IP Configuration.

Additional block support for FPGA floating-point target library mapping

For FPGA floating-point target library mapping, HDL Coder supports these Simulink blocks and block modes:

- MinMax block.
- Unary Minus block.
- Add block with `-` ports.

See also HDL Coder Support for FPGA Floating-Point Library Mapping.

Default video system reference design

You can use a new `Default video system (requires HDMI FMC module)` reference design with these target platforms:

- Xilinx Zynq ZC702 evaluation kit
- Xilinx Zynq ZC706 evaluation kit
- ZedBoard

You must have Embedded Coder and the Computer Vision System Toolbox™ Support Package for Xilinx Zynq-Based Hardware.

Custom reference design enhancements

For your custom reference design, by using the hdlcoder.ReferenceDesign class, specify your own custom parameters and custom callback functions. Therefore, you can customize the settings that HDL Coder uses to create the project, generate the software interface model, and build the FPGA bitstream.

The IP Core Generation workflow has these enhancements:

- The **Set Target Interface** task is split into two tasks. One task is the original **Set Target Interface**, and the other task is a new **Set Target Reference Design**. In the **Set Target Reference Design** task, you can specify the parameters and supported tool version for the target reference design.

  The **reference design** setting has moved from the **Set Target Interface** task to the **Set Target Reference Design** task.
- The new task **Set Target Frequency** means that you can specify the **Target Frequency (MHz)** for your design.

To learn more, see Define Custom Parameters and Callback Functions for Custom Reference Design and hdlcoder.ReferenceDesign.

Compatibility Considerations

In the **Generate Software Interface Model** task, the **Add IP core device driver** check box has been removed. To add the device driver, in the task **Program Target Device**, specify a new Download **Programming method**. The Download **Programming method** copies the generated FPGA bistream, device tree, and system initialization scripts to the SD card on the Zynq board, and keeps the bitstream on the SD card persistently.

The reference design names no longer contain a tool version number. If you load a pre-R2016b model that was saved with a reference design containing a version number in its name, and then open the HDL Workflow Advisor, HDL Coder generates a warning. To avoid this warning, select the reference design that does not have the tool version number, and save the model.

IP Core Generation workflow for Xilinx and Altera FPGA devices

You can use the IP Core Generation workflow to generate an HDL IP core for any supported Xilinx or Altera FPGA device. You can integrate the generated IP core into the Default system reference design, or create a custom board and reference design definition for your own FPGA board.

- With a new task **Set Target Frequency**, you can specify the **Target Frequency (MHz)** for your design.
- If the target device does not have an embedded ARM processor, there is no longer the **Generate Software Interface Model** task.

To learn more, see IP Core Generation Workflow for Standalone FPGA Devices.

Additional FPGA board support for IP Core Generation workflow

You can target the following FPGA boards for the IP core generation workflow:

- `Xilinx Kintex-7 KC705 development board`
- `Arrow DECA MAX 10 FPGA evaluation kit`

For examples that show how to target the FPGA boards, see Using IP Core Generation Workflow with Xilinx FPGA Boards: Xilinx Kintex-7 KC705.

Target clock frequency specification

By using the **Target Frequency (MHz)** setting in the **Target and Optimizations** pane in the Configuration Parameters dialog box, you can specify the target frequency for:

- FPGA floating-point target library mapping: Specify the target frequency that you want the IP to achieve when you use `ALTERA MEGAFUNCTION (ALTERA FP FUNCTIONS)`.
- Adaptive pipelining: Specify the target frequency for HDL Coder to insert required number of pipelines to improve area and timing on the target platform.

Previously, you specified the target frequency for floating-point library mapping in the Configuration Parameters dialog box, on the **HDL Code Generation** > **Global Settings** > **Floating Point Target** tab.

By using the new **Set Target Frequency** task in the HDL Workflow Advisor, you can now specify the target frequency for the following workflows:

- `Generic ASIC/FPGA`
- `IP Core Generation`

From the command line, use the `TargetFrequency` property to save the target frequency on the model.

Simulink Real-Time FPGA I/O workflow support for Xilinx Vivado

For the new Speedgoat boards, the `Simulink Real-Time FPGA I/O` workflow supports `Xilinx Vivado` by using the `IP Core Generation` workflow.

See also IP Core Generation Workflow for Speedgoat Boards.

Speedgoat IO333–325K target hardware support

You can target the `Speedgoat IO333–325K` board with `Xilinx Kintex7` for the `Simulink Real-Time FPGA I/O` workflow.

Updates to supported software

HDL Coder has been tested with:

- Xilinx Vivado Design Suite 2015.4
- Altera Quartus II 15.1

For a list of supported third-party tools and hardware, see Supported Third-Party Tools and Hardware.

R2016a

Version: 3.8

New Features

Bug Fixes

Compatibility Considerations

Model and Architecture Design

Gigasample per Second (GSPS) Signal Processing: Increase throughput of HDL-optimized FFT and IFFT algorithms using frame input

You can increase the throughput of the FFT and IFFT calculation by using vector input and output ports. The internal algorithm computes the FFT or IFFT of each vector element in parallel.

The FFT implementation is now a Radix 2^2 architecture which improves performance for vector input. The table compares hardware implementation resources between the old Radix 2 architecture and the new Radix 2^2 architecture.

Architecture	Multipliers	Adders	Memory	Control Logic For Vector Input
Radix 2 Hybrid	$\log_4$(N-1))	3×$\log_4$(N)	17N/16 – 1	Complicated
Radix 2^2 (SDF)	$\log_4$(N-1)	4×$\log_4$(N)	N – 1	Simple

This change affects these blocks and System objects:

- FFT HDL Optimized
- IFFT HDL Optimized
- `dsp.HDLFFT`
- `dsp.HDLIFFT`

Tunable and nontunable parameter enhancements

You can generate HDL code for:

- Stateflow Charts, State Transition Tables, or Truth Tables that use a tunable parameter.
- MATLAB Function blocks that use a tunable or nontunable parameter with vector, array, `struct`, enumeration, or complex data type.
- MATLAB System blocks containing a System object with tunable properties.

See Generate DUT Ports for Tunable Parameters.

Reusable HDL code enhancements for subsystems with tunable mask parameters

In R2016a, to generate one reusable HDL file for Subsystem blocks that contain Gain and Constant blocks for different values of tunable mask parameters, set the `MaskParameterAsGeneric` option.

Previously, in addition to setting the `MaskParameterAsGeneric` option, you used Atomic Subsystem blocks and selected the **tunable** attribute in the Mask Editor **Parameters & Dialog** tab for the Atomic Subsystem block.

See MaskParameterAsGeneric.

HDL Coder support for nondirect feedthrough setting in MATLAB Function blocks

HDL Coder now supports code generation from MATLAB Function blocks that use the nondirect feedthrough setting. With nondirect feedthrough, you can use MATLAB Function blocks in a feedback loop and prevent algebraic loops.

By default, MATLAB Function blocks have direct feedthrough enabled. To disable it, in the Ports and Data Manager pane, clear the **Allow direct feedthrough** check box.

Nondirect feedthrough enables semantics to ensure that outputs rely only on current state. For additional information, see Use Nondirect Feedthrough in a MATLAB Function Block.

Block Enhancements

Synchronous Subsystem Toggle: Specify enable and reset behavior for cleaner HDL code by using State Control block

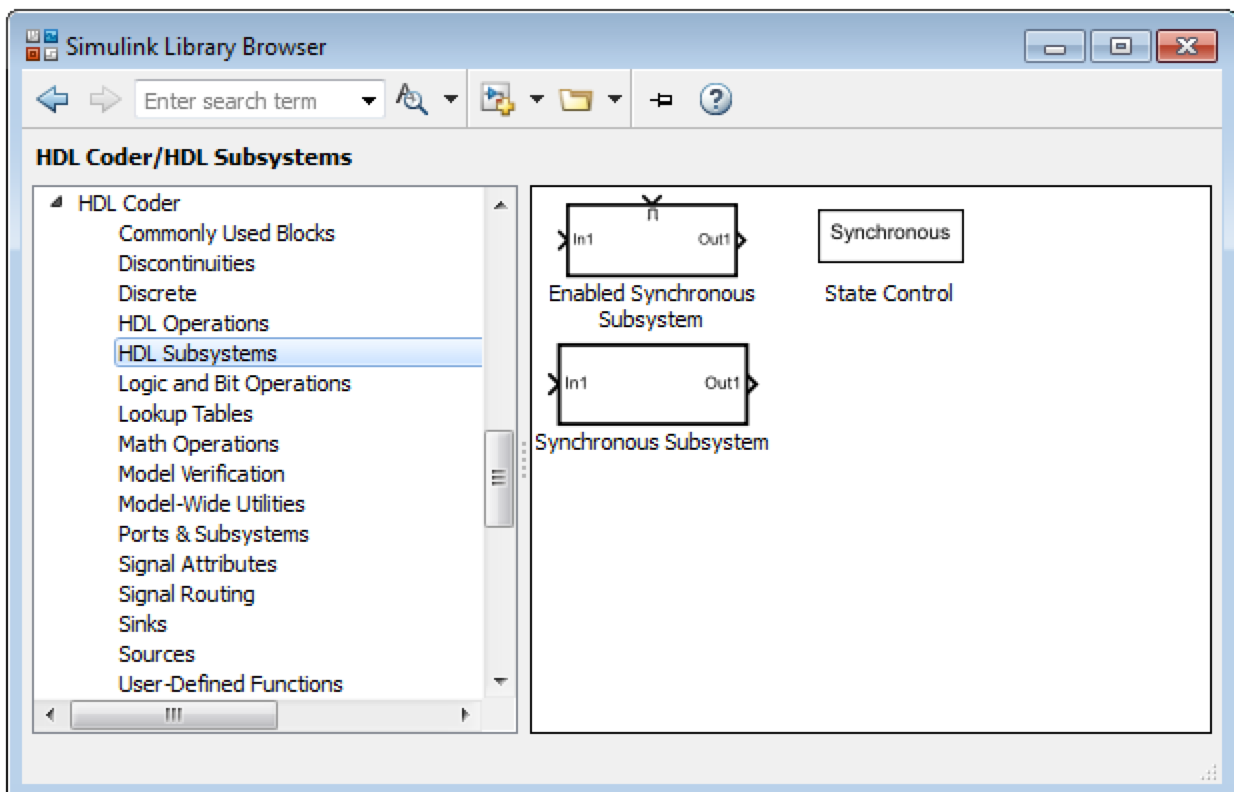
With the State Control block, you can toggle a subsystem between the default Simulink reset and enable behavior and the synchronous hardware reset and enable behavior. How you set the State Control block affects blocks within the subsystem that have state and have reset or enable ports.

If you specify synchronous hardware behavior, the HDL code is cleaner and requires fewer resources because HDL Coder does not generate bypass registers for enabled subsystems or multiplexers for blocks with reset ports.

To toggle a subsystem between synchronous hardware behavior and default Simulink behavior, add a State Control block to the subsystem:

- For synchronous hardware behavior, in the State Control block, set **State control** to `Synchronous`.
- For default Simulink behavior, in the State Control block, set **State control** to `Classic`.

The State Control, Enabled Synchronous Subsystem, and Synchronous Subsystem blocks are available as part of the HDL Subsystems block library in HDL Coder. The Synchronous Subsystem and Enabled Synchronous Subsystem blocks use the synchronous hardware behavior of the State Control block with the Subsystem and Enabled Subsystem blocks respectively.



Use the State Control block with Simulink, DSP System Toolbox, Communications Toolbox™, or Vision HDL Toolbox blocks that support HDL code generation.

For more information, see State Control and Synchronous Subsystem Behavior with the State Control Block.

Region-of-interest selection and grayscale morphology

Vision HDL Toolbox introduces a new block, ROI Selector, that selects a region of interest (ROI) from a video stream. You can specify one or more regions by using input ports or mask parameters. The block returns each new region as streaming pixel data and a corresponding `pixelcontrol` bus.

The `visionhdl.ROISelector` System object provides equivalent MATLAB functionality.

Vision HDL Toolbox includes new blocks and System objects that perform morphology operations on grayscale input data.

Block	System object
Grayscale Closing	`visionhdl.GrayscaleClosing`
Grayscale Dilation	`visionhdl.GrayscaleDilation`
Grayscale Erosion	`visionhdl.GrayscaleErosion`
Grayscale Opening	`visionhdl.GrayscaleOpening`

These blocks and System objects support HDL code generation.

Nested bus support enhancements

In R2016a, HDL Coder supports all nested virtual and nonvirtual buses. For example, you can now generate HDL code for a Delay block with a nested virtual bus signal input in your Simulink model.

Block support enhancements

You can generate HDL code for:

- A Bus to Vector block.
- A Dot Product block with `Tree` architecture when input signals are a mix of row and column vectors.
- A Shift Arithmetic block when **Bits to shift: Number** is a vector of bit shift values or **Bits to shift: Source** is `Input port`.
- Masked Inport and Outport blocks.
- A Matrix Concatenate block with `Multidimensional array` mode.
- A Bus Assignment block with bus signal input containing a nested bus signal.
- A MATLAB System block containing a user-defined System object with bus inputs or outputs.
- An n-D Lookup Table block with the `Breakpoints specification` parameter or a Prelookup block with the `Specification` parameter set to `Explicit values` or `Even spacing`.
- A Subsystem block with `BlackBox` architecture and a cell array variable in the **GenericList** field as input to the Subsystem block.

Code Generation and Verification

Faster Test Bench Generation and HDL Simulation: Generate SystemVerilog DPI test benches for large data sets with HDL Verifier

Reduce test bench generation and simulation time, especially when using large data sets. When you call the `makehdltb` function, set the `GenerateSVDPITestBench` property. The coder generates a DPI component for your entire Simulink model, including your DUT and data sources. Your entire model must support C code generation with Simulink Coder™. The coder generates a SystemVerilog test bench that compares the output of the DPI component with the output of the HDL implementation of your DUT. The tool also generates a build script for your simulator. You can specify '`ModelSim`', '`VCS`', or '`Incisive`'.

```
makehdltb(gcb,'GenerateSVDPITestBench','ModelSim','GenerateHDLTestbench','off')
```

You must have an HDL Verifier license and a Simulink Coder license to use this feature.

Code Generation Report enhancements

The Code Generation Report has these enhancements:

- A new Code Interface Report shows the DUT input and output port names, data types, and bit widths.
- The High-level Resource Report shows the number of 1-bit registers and I/O bits. It includes resource usage for model references.

  See Code Interface Report and High-level Resource Report in Resource Utilization Report.
- For each group of streamed or shared blocks, the Sharing and Streaming report provides more details:
  - For shared blocks, the report shows the resource type, block word length, number of blocks in the group, and a traceability link to the blocks in the original model and generated model. It includes a `Highlight shared resources and diagnostics` link to highlight in the original model and generated model the shared blocks and blocks that are barriers to resource sharing.
  - For streamed blocks, the report links to the group of streamed blocks and shows the streaming factor. It includes a `Highlight streaming groups and diagnostics` link to highlight in the original model and generated model the streamed blocks and blocks that are barriers to streaming.

    See Streaming and Sharing Report in Optimization Report.

Changes to Fixed-Point Conversion Code Coverage

If you use the HDL Coder app to convert your MATLAB code to fixed-point code and propose types based on simulation ranges, the app shows code coverage results. In previous releases, the app showed the coverage as a percentage. In R2016a, the app shows the coverage as a line execution count.

```
11      persistent current_state
12      if isempty( current_state )
13          current_state = S1;                                        1 calls
14      end                                                           51 calls
15
16      % switch to new state based on the value state register
17      switch uint8( current_state )
18          case S1
19              % value of output 'Z' depends both on state and inputs
20              if (A)
21                  Z = true;                                         37 calls
22                  current_state( 1 ) = S1;
23              else                                                   7 calls
24                  Z = false;
25                  current_state( 1 ) = S2;
26              end
27          case S2                                                   51 calls
28              if (A)
29                  Z = false;                                         7 calls
30                  current_state( 1 ) = S1;
31              else                                                   0 calls
32                  Z = true;
33                  current_state( 1 ) = S2;
34              end
35          case S3                                                   51 calls
36              if (A)
37                  Z = false;                                         0 calls
38                  current_state( 1 ) = S2;
39              else
40                  Z = true;
41                  current_state( 1 ) = S3;
42              end
```

See Code Coverage in Automated Fixed-Point Conversion.

Fixed-point conversion requires the Fixed-Point Designer software.

Progress indicator for HDL test bench generation

HDL Coder displays a series of dots to show progress during HDL test bench generation for test benches with a long simulation time.

HDL test bench generation simulates the Simulink model to collect data for every signal in the DUT. This data collection phase can therefore significantly impact HDL test bench generation time. The progress indicator dots help as visual indicators during this long phase of testbench generation.

Test bench generation with updated model stop time

If you generate a test bench, update the stop time in your model, and regenerate the test bench. The generated test bench uses the updated stop time.

Previously, test bench generation used the original stop time even if the stop time was updated.

Performance improvement for MATLAB to HDL test bench generation

In the MATLAB to HDL workflow, HDL Coder uses MEX code for data logging to speed up HDL test bench generation.

Coding standard check for length of control flow statements in a process block

When you enable the **Industry** coding standard, HDL Coder checks for the length of control flow statements, such as if-else, case and loops, which are described separately within a process block (for VHDL code) or an always block (for Verilog code). If the length of control flow statements in your design exceeds the specified limit, the coder displays an error in the HDL coding standard report.

See HDL Coding Standard Rules.

Warnings for non-ASCII characters in generated HDL code

If you have non-ASCII content in model annotations and Model Info blocks, HDL Coder issues warnings during `checkhdl` and `makehdl`. Non-ASCII characters in the generated HDL code can cause RTL simulation and synthesis tools to fail to compile the code.

Japanese translation for resource report

For Japanese versions of HDL Coder, the resource utilization report is in Japanese.

Speed and Area Optimizations

Resource Sharing Enhancements: Share multipliers and gain operations that have different data types

HDL Coder can now share multiply and gain blocks that have the same word length but different fraction lengths and different signs. This optimization reduces the resource utilization by sharing more multipliers and gain operations.

Previously, the word length, fraction length, and signs of the multiply or gain blocks had to be the same.

See Requirements and Limitations for Resource Sharing in Resource Sharing.

Biquad Filter block participates in subsystem HDL optimizations

The Biquad Filter block is now included in subsystem optimizations for speed and area of the generated HDL. To specify resource sharing, streaming, and pipeline options, right-click the subsystem containing the Biquad Filter block and open the **HDL Code** > **HDL Properties** dialog box. To use these optimizations you must set the **Architecture** of the Biquad Filter block to `Fully parallel`.

The optimizations work the same way as the optimizations for the Discrete FIR Filter block. You can share resources between Biquad Filter and Discrete FIR Filter blocks in the same subsystem.

More functions for Multiply-Add block to map to DSP

You can now choose from three different functions for the Multiply-Add block to map to the DSP blocks in Altera and Xilinx FPGA libraries. The three functions are `c+(a.*b)`, `c-(a.*b)`, and `(a.*b)-c`.

For details, see Multiply-Add.

Generation of Multiply-Add blocks for complex multiply operations

If you have Product or Gain blocks with complex input signals in your Simulink model, HDL Coder generates a model with Multiply-Add blocks. These Multiply-Add blocks map efficiently to the DSP blocks in Altera and Xilinx FPGA libraries.

RAM mapping for pipeline and floating-point delays

To optimize for area by mapping pipeline registers to RAM, in the Configuration Parameters dialog box, select the **Map pipeline delays to RAM** check box from the **HDL Code Generation** > **Global Settings** > **Optimization** tab. See MapPipelineDelaysToRAM.

HDL Coder also maps design delays and the pipeline registers in floating-point type to RAM.

Initialization script generated for Delay blocks without reset for ModelSim simulation

If you have delay blocks with in your model with `ResetType` set to `None`, HDL Coder generates a script to initialize these delay blocks for simulation with ModelSim.

Previously, you either modified the generated code or wrote your own script to initialize these delay blocks.

IP Core Generation and Hardware Deployment

Hard Floating-Point IP Targeting: Generate HDL to map to Altera Arria 10 floating-point units at user-specified target frequency

HDL Coder can now map your Simulink model to Altera floating-point IP (ALTERA FP FUNCTIONS) at the target frequency that you specify.

Previously, when mapping to Altera megafunction IP (ALTFP) or Xilinx LogiCORE® IP, you could specify only whether to optimize the Simulink model for minimum or maximum latency and for speed or area.

For more information, see FPGA Floating-Point Library Mapping and `TargetFrequency`.

Compatibility Considerations

Previously, you chose the floating-point target library by selecting the **Set Target Library (for floating-point synthesis support)** check box from the HDL Workflow Advisor.

You now specify the floating-point target library from the **HDL Code Generation** > **Global Settings** > **Floating Point Target** tab in the Model Configuration Parameters dialog box.

End-to-end scripting for Simulink Real-Time FPGA I/O workflow

You can use the HDL Workflow Command Line Interface (CLI) to script the entire Simulink Real-Time FPGA I/O workflow.

To create the script to configure your design using the HDL Workflow Advisor, generate a target hardware bitstream or project from your Simulink model, and then export a script. Run the exported script, which contains HDL Workflow CLI commands, to replicate your HDL Workflow Advisor settings and generate the same target hardware bitstream or project.

SoC device programmed by using Ethernet connection

When using the IP core generation workflow in the HDL Workflow Advisor, you can program the target SoC device by using an Ethernet connection.

You must have Embedded Coder and the Embedded Coder Support Package for Intel SoC Devices.

Custom programming method for IP Core Generation workflow

Using the `CallBackCustomProgrammingMethod` method of the `hdlcoder.ReferenceDesign` class, you can define your own function to program the target device in your custom reference design.

Interface connection name and type for custom reference designs

Using the `AXI4SlaveInterface` method of the `hdlcoder.ReferenceDesign` class, you can specify the type of an AXI4 slave interface in a custom reference design. The type can be AXI4, or AXI4-Lite. You can also name the interface.

Updates to supported software

HDL Coder has been tested with:

- Xilinx Vivado Design Suite 2015.2
- Altera Quartus II 15.0

For a list of supported, third-party tools and hardware, see Supported Third-Party Tools and Hardware.

Automatic generation of FPGA top-level wrapper based on workflow

For the FPGA Turnkey and Simulink Real-Time FPGA I/O workflows, HDL Coder generates a top-level HDL code wrapper and a constraint file that contains pin mapping and clock constraints. In the HDL Workflow Advisor, for the FPGA Turnkey and Simulink Real-Time FPGA I/O workflows, the name of the **Generate RTL Code and Testbench** task is now **Generate RTL Code**.

To run the **Annotate Model with Synthesis Result** task, your target workflow must be `Generic ASIC/FPGA`.

Compatibility Considerations

For all workflows, the **Generate FPGA top level wrapper** check box and `GenerateTopLevelWrapper` of the `hdlcoder.WorkflowConfig` class have been removed.

For the Generic ASIC/FPGA workflow, if you specify the `GenerateTopLevelWrapper` property of the `hdlcoder.WorkflowConfig`, HDL Coder displays a warning. In a future release, specifying this property will result in an error.

For the FPGA Turnkey or Simulink Real-Time FPGA I/O workflow, if you specify the following `hdlcoder.WorkflowConfig` properties, HDL Coder displays a warning. In a future release, specifying these properties will result in an error:

- `RunTaskGenerateRTLCodeAndTestbench`
- `RunTaskVerifyWithHDLCosimulation`
- `RunTaskAnnotateModelWithSynthesisResult`
- `GenerateRTLTestbench`
- `GenerateCosimulationModel`
- `CosimulationModelForUseWith`
- `GenerateValidationModel`
- `GenerateTopLevelWrapper`
- `CriticalPathSource`
- `CriticalPathNumber`
- `ShowAllPaths`
- `ShowDelayData`
- `ShowUniquePaths`
- `ShowEndsOnly`

R2015aSP1

Version: 3.6.1

Bug Fixes

R2015b

Version: 3.7

New Features

Bug Fixes

Compatibility Considerations

Model and Architecture Design

Model Arguments: Parameterize instances of model reference blocks

You can generate VHDL `generic` or Verilog `parameter` syntax for model arguments you use in a masked or unmasked Model block. In the model, you can use the model arguments in Gain or Constant blocks.

For details, see Generate Parameterized Code for Referenced Models.

Integration with Xilinx Vivado System Generator for DSP blocks

You can generate code for subsystems containing Xilinx System Generator for DSP blocks when Xilinx Vivado is your synthesis tool. For setup information, see Xilinx System Generator Setup for ModelSim Simulation.

struct input and output for top-level MATLAB design function

You can generate HDL code for a top-level MATLAB design function that has `struct` inputs or outputs. You can also generate HDL code for a test bench that uses `struct` data. Previously, `struct` data was supported within the design, but not at the top-level inputs or outputs.

For example, in Vision HDL Toolbox, this removes the requirement to flatten the `pixelcontrol` structure into the component signals, as shown here.

```
function [pixOut,hStartOut,hEndOut,vStartOut,vEndOut,validOut] = ...
               HDLTargetedDesign(pixIn,hStartIn,hEndIn,vStartIn,vEndIn,validIn)
```

With HDL code generation support for structures, the arguments can now include the control signal structure.

```
function [pixOut,ctrlOut] = HDLTargetedDesign(pixIn,ctrlIn)
```

The structure is flattened to the individual control signals in the generated Verilog or VHDL code.

Tunable parameters in MATLAB Function block

When you generate code for a MATLAB Function block that uses a tunable parameter, the coder creates a top-level DUT port for the tunable parameter in the generated code.

For details, see Generate DUT Ports for Tunable Parameters.

Output initialization requirement for Stateflow Moore Charts

If you have a model with a Stateflow `Moore` Chart, select the **Initialize Outputs Every Time Chart Wakes Up** chart property. By selecting this property, HDL Coder prevents latching of outputs, generates more readable HDL code, and provides better synthesis results.

Compatibility Considerations

In previous versions, you did not have to set the chart property. Starting in R2015b, If you do not select the **Initialize Outputs Every Time Chart Wakes Up** check box, HDL Coder generates an error.

Enforce ASCII character requirement for model property values

For HDL model properties that require ASCII character strings, HDL Coder now generates an error if you assign a non-ASCII string value.

The following model properties accept non-ASCII character strings:

- BlocksWithNoCharacterizationFile
- CriticalPathEstimationFile
- DateComment
- DistributedPipeliningBarriersFile
- HighlightFeedbackLoopsFile
- SimulationLibPath
- SynthesisProjectAdditionalFiles
- TargetDirectory

Compatibility Considerations

Previously, HDL Coder did not generate an error if you assigned a non-ASCII character string to a model property that required ASCII characters.

To fix the error, at the command prompt, enter: `hdlset_param (gcs, *model_property_name*, '*ASCII_value*')`.

Block Enhancements

Expanded Bus Support: Generate HDL for enabled or triggered subsystems with bus inputs and for black boxes with bus I/O

You can now generate HDL code for the following blocks with bus input or output signals:

- Enabled Subsystem
- Triggered Subsystem
- Subsystem with black box implementation

Library Browser view shows blocks supported for HDL code generation

In the Library Browser, you can enable a filtered view that shows all the blocks that are compatible with HDL code generation. The view shows only those blocks for which you have a license. To use this filtered view, at the command prompt, enter `hdllib`.

After using the `hdllib` command, if you close and reopen the Library browser, the view that shows only those blocks that are compatible for HDL code generation persists. To display all blocks in the Library Browser, enter `hdllib('off')`.

For more information, see Show Blocks Supported for HDL Code Generation.

Compatibility Considerations

Previously, the `hdllib` command created a supported blocks library called `hdlsupported`. The `hdllib` command now opens the Library Browser with a view that shows the supported blocks for HDL code generation, but does not create a block library.

To create the supported blocks library, at the command prompt, enter `hdllib('librarymodel')`.

Trigonometric Function block with sin or cos function can have vector inputs

You can generate code for a Trigonometric Function block with vector inputs when the **Function** is `sin`, `cos`, `cos + jsin`, or `sincos`, and the **Approximation method** is `CORDIC`.

In the HDL Block Properties dialog box, you can also set **UsePipelinedKernel** to `Off` for zero-latency combinatorial HDL code. To avoid delay balancing errors, you can set **UsePipelinedKernel** to `Off` if the block is in a feedback loop.

See Trigonometric Function.

Discrete FIR Filter supports HDL optimizations

You can now optimize speed and area of the generated HDL for the Discrete FIR Filter block. Right-click the subsystem containing the Discrete FIR Filter block, and open the **HDL Code** > **HDL Properties** dialog to specify resource sharing, streaming, and pipeline options. You can use these

optimizations when the **Architecture** is `Fully parallel`. This feature requires an HDL Coder license. See Reduce Critical Path with Distributed Pipelining, Resource Sharing of Multipliers to Reduce Area, and HDL Block Properties.

HDL-optimized FIR Rate Conversion block and System object

FIR Rate Conversion HDL Optimized block in DSP System Toolbox upsamples, filters, and downsamples a signal using an efficient polyphase FIR structure. The block operates on one sample at a time and provides hardware control signals to pace the flow of samples in and out of the block. The `dsp.HDLFIRRateConverter` System object provides equivalent MATLAB functionality. Both the block and System object support HDL code generation.

Code Generation and Verification

HDL Coder Configuration Parameters in list view

The Configuration Parameters list view shows HDL Code Generation parameters. In list view, you can search, filter, sort, and edit parameters. You can also get command-line parameter names for use in scripts. To use the list view, click the **List** button at the top of the Configuration Parameters dialog box.

Some of the HDL Code Generation parameters, such as the lint script, synthesis script, and HDL coding standard parameters, are unavailable in list view. To see all the HDL Code Generation parameters, use the category view.

To learn about Configuration Parameters list view, see Configuration Parameters Dialog Box Overview.

Support for configuration parameter Default parameter behavior

HDL code you generate for models that use the configuration parameter **Default parameter behavior**, which was previously called **Inline parameters**, follows the new behavior.

To learn more about the **Default parameter behavior** name and functionality change, see Configuration parameter Inline parameters name and functionality change.

Compatibility Considerations

Sample time propagation for `Inf` rates in Simulink can differ from previous releases. If you have a model with `Inf` sample times, check the sample time legend to make sure it shows the rates you expect.

To fix code generation issues from such models, fully specify the sample time. For example, specify the sample time for any Constant blocks with `Inf` sample time.

Test bench performance improvements with file I/O

Infrastructure improvements in test bench generation with file I/O have improved test bench performance by reducing:

- Simulation time
- Memory use
- Code generation time

HDL Coder writes the DUT stimulus and reference data from your MATLAB or Simulink simulation to data files (`.dat`). During HDL simulation, the HDL test bench reads the saved stimulus from the `.dat` files.

To learn more about Simulink test bench generation, see Test Bench Generation.

To learn more about MATLAB test bench generation, see Test Bench Generation.

Compatibility Considerations

The HDL test bench code generated for a design in a previous release may differ from the code generated for the same design in the current release.

Previously, by default, DUT stimulus and reference data was generated as constants in the test bench code. To generate test bench data that used file I/O, you had to set the `UseFileIOInTestBench` property to `'on'`.

The `UseFileIOInTestBench` property is now `'on'` by default.

Image processing examples

Five examples of image processing and HDL code generation using Vision HDL Toolbox are added in this release.

- Gamma Correction
- Histogram Equalization
- Edge Detection and Image Overlay
- Edge Detection and Image Overlay with Impaired Frame
- Noise Removal and Image Sharpening

Speed and Area Optimizations

Quality of Results Improvement: Stream and share resources more broadly and efficiently

The streaming optimization works with:

- Subsystems that contain nested subsystems. Previously, the streamed subsystem had to be a leaf subsystem.
- A streaming factor that is divisible by vector width. Previously, the streaming factor had to be a divisor of the vector width.
- Blocks that are not supported for streaming. Instead, the optimization can work around unsupported blocks. Previously, the presence of unsupported blocks caused streaming to fail.

The resource sharing area optimization can implement shared resources at the clock rate within clock-rate pipelining regions, without using oversampling.

The resource sharing optimization can now share a subset of all the identical atomic subsystems in your design. Previously, you had to share all of the atomic subsystems, or none.

The hierarchy flattening optimization can operate on individual atomic subsystems, even if there are other identical atomic subsystems in the design.

Multiply-Add block

A new block, Multiply-Add, that performs a hardware optimized multiplication-addition operation, is available in the HDL Operations block library. Using this block can help your design map to DSPs in hardware. The resource sharing optimization can also share Multiply-Add blocks.

To learn about the Simulink behavior, see Multiply-Add.

To learn about the hardware implementation and HDL block properties, see Multiply-Add.

Hierarchy flattening for masked subsystems and user library blocks

For masked subsystems and user library blocks, you can flatten hierarchy to enable further speed and area optimization.

For details, see Hierarchy Flattening.

Loop optimization improvement

The loop streaming implementation now uses fewer multiplexers and therefore uses less area.

Complex Gain speed optimization

Register retiming for a complex Gain block inserts a register between the multiplier and adder.

Redesigned serializer for streaming and resource sharing

The serializer used in streaming and resource sharing is redesigned as a combinational switch to use less area. It no longer uses registers in its implementation.

Tapped Delay optimization

The Tapped Delay block is supported for streaming, retiming, and floating-point library mapping. It also no longer inhibits clock-rate pipelining.

IP Core Generation and Hardware Deployment

Tunable Parameters: Map to AXI4 interfaces to enable hardware run-time tuning by the embedded software on the ARM processor

For Xilinx Zynq or Altera SoC hardware, you can map tunable parameters in your Simulink model to an AXI4, AXI4-Lite, or external interface in the generated IP core. You can then use the embedded software to dynamically tune the parameter in hardware.

You can also map tunable parameters to the PCI interface in the Simulink Real-Time FPGA I/O workflow.

The HDL Workflow Advisor Target Platform Interface Table shows tunable parameters in the Port Name column. You can map each tunable parameter to a target platform interface as you can with any DUT port.

To make a tunable parameter available for mapping to a target interface, use it in a Gain, Constant, or MATLAB Function block.

End-to-end scripting from design through IP core generation, FPGA Turnkey, and generic ASIC/FPGA workflows

You can use the HDL Workflow Command Line Interface (CLI) to script the entire generic ASIC/FPGA, IP core generation, and FPGA Turnkey workflows.

The simplest way to create the script is to configure your design using the HDL Workflow Advisor, generate a target hardware bitstream or project from your Simulink model, then export a script. You can run the exported script, which contains HDL Workflow CLI commands, to replicate your HDL Workflow Advisor settings and generate the same target hardware bitstream or project.

For details, see Run HDL Workflow with a Script.

Synthesis objective for synthesis tool target optimization

You can specify a high-level synthesis objective that maps to your third-party synthesis tool can use to optimize the target hardware. The following high-level synthesis objectives are available from the HDL Workflow Advisor and the HDL Workflow Command Line Interface:

- Area Optimized
- Compile Optimized
- Speed Optimized
- None (default)

For details about how the synthesis objective maps to Tcl commands, see Synthesis Objective to Tcl Command Mapping.

AXI4-Stream vector interface

In the hardware-software codesign workflow, for streaming applications, you can use vector ports to connect the hardware DUT to the rest of the model. With an Embedded Coder license, you can then

generate a software interface model that includes the embedded software DMA driver block for the generated IP core.

In the hardware DUT, connect the top-level input and output vector ports to Serializer1D and Deserializer1D blocks. HDL Coder detects this modeling pattern, and generates a software interface model that replaces the Serializer1D, Deserializer1D, and DUT.

For details, see Model Design for AXI4-Stream Interface Generation.

Connect IP core with other IP blocks in custom reference designs

Using the `hdlcoder.ReferenceDesign.addInternalIOInterface` method, you can define a connection between your generated IP core and other IP in a custom reference design. You can use this method for Altera Quartus II, Xilinx Vivado, and Xilinx ISE hardware targets.

Kintex UltraScale and Virtex UltraScale device family support in generic ASIC/FPGA and IP core generation workflows

In the generic ASIC/FPGA workflow and IP core generation workflow, you can target Xilinx Kintex® UltraScale and Virtex UltraScale devices.

R2015a

Version: 3.6

New Features

Bug Fixes

Compatibility Considerations

Model and Architecture Design

Localized control using pragmas for pipelining, loop streaming, and loop unrolling in MATLAB code

You can use pragmas in your MATLAB code to specify pipelining, loop streaming, and loop unrolling optimizations for specific operations.

For a loop statement, you can use `coder.hdl.loopspec` to specify loop unrolling or loop streaming.

For an operation or expression, you can use `coder.hdl.pipeline` to insert one or more pipeline registers.

The following example shows how to use these two pragmas:

```
function y = hdltest(x)
    pv = uint8(1);
    pv = coder.hdl.pipeline(pv + x, 4);

    y = uint8(zeros(1,10));

    coder.hdl.loopspec('stream', 5);
    for i = 1:10
        y(i) = pv + i;
    end
end
```

To learn more, see:

- Pipeline MATLAB Expressions
- Optimize MATLAB Loops

Compatibility Considerations

If you have a model that uses the VariablesToPipeline HDL block property, or a MATLAB design that uses the HDL Workflow Advisor **Pipeline variables** field, the software displays a warning when you generate code.

Replace instances of `Variables ToPipeline` or **Pipeline variables** with `coder.hdl.pipeline`. `VariablesToPipeline` and **Pipeline variables** will be removed in a future release.

Model templates for HDL code generation

Model templates are available for you to use when designing a model for HDL code generation. These model templates show design patterns for using Simulink blocks to model hardware and generate efficient HDL code.

For example, the Simulink Template Gallery contains model templates for ROM, state machines, shift registers, and multipliers that map to DSP48s.

To view the HDL Coder model templates, open the Simulink Library Browser, click the New Model button arrow, and select **From Template**. In the Simulink Template Gallery, browse to the HDL Coder folder.

For more information, see Simulink Templates For HDL Code Generation.

Tunable parameter data type and model reference support enhancements

You can generate a DUT port for tunable parameters that have the following data types:

- Complex
- Vector
- Structure
- Enumeration

You can also generate a DUT port for tunable parameters when your DUT is a model reference.

To learn how to generate code for tunable parameters, see Generate DUT Ports For Tunable Parameters.

Include custom or legacy code using DocBlock

You can integrate custom or legacy HDL code into your design with a black box subsystem that contains DocBlock.

In the DocBlock HDL Block Properties dialog box, set **Architecture** to `HDLText` and **TargetLanguage** to your target HDL language. Specify the interface to your custom code by customizing the black box subsystem interface.

For details, see Integrate Custom HDL Code Using DocBlock.

Single library for VHDL code generated from model references

You can generate VHDL code for model references in your design into a single library. To generate code into a single library, set the `UseSingleLibrary` property to `on` using `makehdl` or `hdlset_param`.

Timing controller architecture and postfix options in Configuration Parameters dialog box and HDL Workflow Advisor

You can specify the timing controller architecture and timing controller postfix in the Configuration Parameters dialog box and HDL Workflow Advisor, from both Simulink and MATLAB.

You can also specify these timing controller options at the command line.

Functionality Being Removed or Changed

Functionality	What Happens When You Use This Functionality	Use This Functionality Instead	Compatibility Considerations
`AlteraBlackBox` architecture for Subsystem block	The software displays an error.	Create an Altera DSP Builder Subsystem	Replace all instances of `AlteraBlackBox` with `Module`, and follow the procedure in Create an Altera DSP Builder Subsystem.
`XilinxBlackBox` architecture for Subsystem block	The software displays an error.	Create a Xilinx System Generator Subsystem	Replace all instances of `XilinxBlackBox` with `Module`, and follow the procedure in Create a Xilinx System Generator Subsystem.

Functionality	What Happens When You Use This Functionality	Use This Functionality Instead	Compatibility Considerations
`VariablesToPipeline` block property or **Pipeline variables** field	Still runs.	`coder.hdl.pipeline`	Replace all instances of `VariablesToPipeline` or **Pipeline variables** with `coder.hdl.pipeline`. See “Localized control using pragmas for pipelining, loop streaming, and loop unrolling in MATLAB code” on page 15-2.

Block Enhancements

Enumeration support at DUT ports

You can use enumerated data at the top-level DUT ports for your design, whether the DUT is a MATLAB design function, or a Simulink subsystem or model reference.

Map to multiple RAM banks

You can map an `hdl.RAM` System object in your MATLAB code to multiple RAM banks.

If you specify vector inputs to the `step` method, the `hdl.RAM` maps to RAM banks. The number of RAM banks is the same as the number of elements in each input vector.

Code generation for bus output from Bus Selector and Constant blocks

You can generate code for:

- Bus Selector with **Output as bus** enabled.
- Constant with **Output data type** set to Bus.

Initial condition for Deserializer1D

For the Deserializer1D block, you can specify the **Initial condition**.

Block support enhancements

Additional Simulink block features are supported for HDL code generation:

- Delay with delay length of 0
- Delay with **Show enable port** enabled
- Dot Product within a delay balancing region
- Model reference block Description field maps to a comment
- Deserializer1D and Serializer1D support enumeration data.

Code generation for predefined System objects in MATLAB System block

You can generate code for the following predefined System objects when you use them in a MATLAB System block:

- `hdl.RAM`
- `comm.HDLCRCDetector`
- `comm.HDLCRCGenerator`
- `comm.HDLRSDecoder`

- `comm.HDLRSEncoder`
- `dsp.DCBlocker`
- `dsp.HDLComplexToMagnitudeAngle`
- `dsp.HDLFFT`
- `dsp.HDLIFFT`
- `dsp.HDLNCO`

Specify filter coefficients using a System object

For Biquad Filter, FIR Decimation, FIR Interpolation, CIC Decimation, and CIC Interpolation blocks, HDL code generation is supported for **Coefficient source** set to **System object**. These blocks and objects are available in DSP System Toolbox.

Libraries for HDL-supported DSP System Toolbox and Communications Toolbox blocks

Find blocks that support HDL code generation, in the 'DSP System Toolbox HDL Support' and 'Communications System Toolbox HDL Support' libraries, in the Simulink library browser. Alternately, you can type `dsphdllib` and `commhdllib` at the MATLAB command prompt to open these libraries.

The blocks in `dsphdllib` and `commhdllib` have their parameters set for HDL code generation.

Support for image processing, video, and computer vision designs in new Vision HDL Toolbox product

Vision HDL Toolbox provides pixel-streaming algorithms for the design and implementation of vision systems on FPGAs and ASICs. It provides a design framework that supports a diverse set of interface types, frame sizes, and frame rates, including high-definition (1080p) video. The image processing, video, and computer vision algorithms in the toolbox use an architecture appropriate for HDL implementations.

The toolbox algorithms are designed to generate readable, synthesizable code in VHDL and Verilog (with HDL Coder). The generated HDL code can process 1080p60 in real time.

Toolbox capabilities are available as MATLAB System objects and Simulink blocks.

See Vision HDL Toolbox

Support for 'inherit via internal rule' data type setting on FIR Decimation and Interpolation blocks

FIR Decimation and FIR Interpolation blocks now support HDL code generation with data types specified by **Inherit via internal rule**. These blocks are available in DSP System Toolbox.

Code Generation and Verification

Coding standard check for X and Z constants

When you enable the **Industry** coding standard, HDL Coder checks for unknown or high-impedance constants in your design. If your design uses these constants, the coder displays a warning.

For VHDL, the coder checks for `X`, `Z`, `U`, `W`, `H`, `L`, and `-`. For Verilog, the coder checks for `X` and `Z`.

Coding style improvements

The generated code has the following coding style improvements:

- Stateflow charts for Moore machines generate code that follows the coding style guidelines from Altera and Xilinx. Open the `hdlcoder_fsm_mealy_moore` model as mentioned in Generate HDL for Mealy and Moore State Machines to see an example of a Moore chart that generates this style of HDL code.
- Comments for a Simulink block appear with the main body of the associated generated code.
- For Unit Delay Resettable, Unit Delay Enabled Resettable, and Delay with **External reset** set to `Level`, the reset signal is applied within the clocked region for better synthesis results.
- Fewer temporary variables for improved multiplier mapping and readability.
- Expressions of the form `(a+1)-1` are reduced to `a`.
- One-line boolean expressions are generated when they can replace if-else statements.
- Verilog code generated for arrays of constants is more compact. The number of lines of generated code is reduced by 50%.

Example HDL implementation of LTE OFDM modulator and detector with LTE Toolbox

The Verification of HDL Implementation of LTE OFDM Modulator and Detector example uses Simulink blocks that support HDL code generation to implement a hardware-friendly LTE Orthogonal Frequency Division Multiplexing (OFDM) modulator and detector. Running this example requires LTE Toolbox™.

Speed and Area Optimizations

Critical path estimation without running synthesis

Critical path estimation helps you to find the timing critical path in your design without running third-party synthesis tools.

If you enable critical path estimation when you generate code, HDL Coder computes the timing critical path and generates a script that highlights the estimated critical path in the generated model.

To find your estimated critical path, in **HDL Workflow Advisor** > **HDL Code Generation** > **Set Code Generation Options** > **Set Basic Options**, select **Generate high-level timing critical path report**.

The estimated critical path is calculated using static timing analysis. In the current release, the timing data for each block is based on Xilinx Virtex-7, speed grade -1 hardware.

If a block in your design does not have timing data, the coder generates a second block highlighting script. To see the uncharacterized blocks in your design, click the script link displayed in the MATLAB command window or HDL Workflow Advisor **Result** pane.

For more information, see Find Estimated Critical Paths Without Synthesis Tools.

Clock-rate pipelining enhancements

The following clock-rate pipelining enhancements are available:

- MATLAB Function blocks that do not have state can be pipelined at the clock rate.
- DUT output ports can be pipelined at the clock rate. In the Simulink HDL Workflow Advisor **Optimizations** tab, enable the **Allow clock-rate pipelining of DUT output ports** option, or set the `ClockRatePipelineOutputPorts` property to `on`.
- HDL Coder generates a MATLAB script that highlights blocks that are inhibiting clock-rate pipelining. You can run the script by clicking the associated link the optimization report.

Partitioning for large multipliers to improve clock frequency and DSP reuse on the FPGA

You can partition large multipliers by specifying a maximum multiplier bit width for your design.

To specify the maximum multiplier bit width, in the HDL Workflow Advisor **Optimization** tab, for **Multiplier partitioning threshold**, enter an integer value greater than or equal to 2. See also `MultiplierPartitioningThreshold`.

Highlighting for blocks in the model that prevent retiming

With distributed pipelining, HDL Coder generates a MATLAB script that highlights blocks that are inhibiting the optimization, and displays messages for highlighted blocks that describe why the block is inhibiting the optimization. The script highlights blocks in your original model and generated model.

To run the highlighting script, click the associated link in optimization report.

Script generation is on by default. You can disable script generation by setting the `DistributedPipeliningBarriers` property to `off` with `makehdl` or `hdlset_param`.

Resource sharing for adders and more control over shareable resources

You can now specify the types of blocks or operations to share in the parts of your design that have resource sharing enabled. You can enable or disable resource sharing for adders, multipliers, atomic subsystems, and MATLAB Function blocks.

You can also specify minimum bit widths for shared adders and multipliers.

Speed and area optimizations for designs that use Unit Delay Enabled, Unit Delay Resettable, and Unit Delay Enabled Resettable

You can use speed and area optimizations in designs that contain Unit Delay Enabled, Unit Delay Resettable, and Unit Delay Enabled Resettable blocks. For example, you can use the following optimizations:

- Resource sharing
- Streaming
- Distributed pipelining or retiming
- Input, output, and constrained output pipelining
- Clock-rate pipelining

Resource sharing for multipliers and adders with input data types in different order

You can share multipliers or adders when their input ports have the same data types, but in a different order. For example, you can share the following two multipliers:

- Multiplier A, with `uint8` data on port X and `uint16` data on port Y.
- Multiplier B, with `uint16` data on port X and `uint8` data on port Y.

Vector streaming for MATLAB code

When the loop streaming optimization is enabled in MATLAB code, HDL Coder applies the streaming optimization to vector operations to minimize multiplexer and register usage.

The following types of vector operations benefit from vector streaming:

- Single vector operations.

  For example:

  ```
  y = u .* v;
  ```
- Chained vector operations.

  For example:

```
t = u .* v;
y = t + w;
```

- Chained vector operations across a persistent variable.

  For example:

  ```
  persistent acc;
  if isempty(acc)
      acc = uint16(zeros(size(u)));
  end

  t = u .* v;
  acc = t + acc;
  y = acc;
  ```

IP Core Generation and Hardware Deployment

Mac OS X platform support

You can install and run HDL Coder to generate code on the 64-bit Mac OS X platform.

AXI4-Stream interface generation for Xilinx Zynq IP core

You can generate an IP core with an AXI4-Stream interface when you target the Xilinx Zynq-7000 platform and your synthesis tool is Xilinx Vivado.

For an example that shows how to generate an HDL IP core with an AXI4-Stream interface, see Getting Started with AXI4-Stream Interface in Zynq Workflow.

Custom reference design and custom SoC board support

You can now define a custom SoC board or a custom reference design.

In the HDL Workflow Advisor, you can generate an IP core for a custom board, insert it into a custom reference design, and generate an FPGA bit stream for the SoC hardware.

To learn more about defining and registering a custom board or custom reference design, see:

- Board and Reference Design Registration System
- Register a Custom Board
- Register a Custom Reference Design

For an example, see Define and Register Custom Board and Reference Design for SoC Workflow.

Automatic iterative optimization for IP core generation and FPGA Turnkey workflows

After you achieve your clock frequency target using automatic iterative optimization, you can generate a custom IP core or use the FPGA Turnkey workflow with the optimized design.

Speedgoat IO331-6 digital I/O interface target

When you target the Speedgoat IO331-6 board, in the HDL Workflow Advisor **Target platform interface table**, you can select `TTL I/O Channel [0:15]` to connect your design interface to digital I/O pins.

IP core settings saved with model

For the IP core generation workflow, when you specify IP core settings, HDL Coder saves the information with your model. The following HDL Workflow Advisor fields are saved with the model as HDL block properties of the DUT block.

HDL Workflow Advisor field	HDL Block Property
IP core name	`IPCoreName`
IP core version	`IPCoreVersion`
Additional source files	`IPCoreAdditionalFiles`
Processor/FPGA synchronization	`ProcessorFPGASynchronization`

For the DUT block, you can set and view **IPCoreName**, **IPCoreVersion**, **IPCoreAdditionalFiles**, and **ProcessorFPGASynchronization** with the HDL Block Properties dialog box or `hdlset_param` and `hdlget_param`. To learn more about the block properties, see Atomic Subsystem or Subsystem

For an example that shows how to configure target hardware settings in your model, see Save Target Hardware Settings in Model.

Updates to supported software

HDL Coder has been tested with:

- Xilinx Vivado Design Suite 2014.2
- Altera Quartus II 14.0

For a list of supported third-party tools and hardware, see Supported Third-Party Tools and Hardware.

R2014b

Version: 3.5

New Features

Bug Fixes

Compatibility Considerations

Model and Architecture Design

Custom or legacy HDL code integration in the MATLAB to HDL workflow

You can use a black box System object, `hdl.BlackBox`, to integrate custom HDL code into your design in the MATLAB to HDL workflow. For example, you can integrate handwritten or legacy HDL code that you previously generated from MATLAB code or a Simulink model.

For an example that shows how to use `hdl.BlackBox`, see Integrate Custom HDL Code Into MATLAB Design.

Model reference as DUT for code generation

You can directly generate code for a model reference, without placing it in a Subsystem block. Previously, the code generation DUT had to be a Subsystem block.

Tunable parameter support for Gain and Constant blocks

The coder generates a top-level DUT port for each tunable parameter in your DUT that you use as the **Gain** parameter in a Gain block, or the **Constant value** parameter in a Constant block.

For details, see Generate Code For Tunable Parameters.

Code generation for Stateflow active state output

If you enable active state output to show child activity or leaf state activity for a Stateflow block, the coder generates code for the active state output. See Active State Output.

Clock enable minimization for code generated from MATLAB designs

You can minimize clock enable logic in your generated code by setting the `MinimizeClockEnables` property of the `coder.HdlConfig` object to `true`, or by enabling the **Minimize clock enables** option in the HDL Workflow Advisor.

For details, see Minimize Clock Enables.

HDL Block Properties dialog box shows only valid architectures

For each block supported for code generation, the HDL Block Properties dialog box **Architecture** drop-down list shows only the architectures that are valid for the block based on mask parameter settings. Previously, all architectures were available for selection regardless of mask parameter settings, and invalid settings caused errors during code generation.

2-D matrix types in HDL generated for MATLAB matrices

When you have matrices in your MATLAB code, you can generate 2-D matrices in HDL code. By default, the software generates HDL vectors with additional index computation logic, which can use more area in the synthesized hardware than HDL matrices.

To generate 2-D matrix types in HDL in the MATLAB to HDL workflow:

- In the HDL Workflow Advisor, in the **HDL Code Generation** > **Coding Style** tab, select **Use matrix types in HDL code**.
- At the command line, set the `UseMatrixTypesInHDL` property of the `coder.HdlConfig` object to `true`.

Previously, 2-D matrix types could be generated in HDL for the Simulink MATLAB Function block, but not in the MATLAB to HDL workflow.

Block Enhancements

Code generation for HDL optimized FFT/IFFT System object and HDL optimized Complex to Magnitude-Angle System object and block

You can generate code for the `dsp.HDLFFT` and `dsp.HDLIFFT` System objects, Complex to Magnitude-Angle HDL Optimized block, and `dsp.ComplexToMagnitudeAngle` System object, which are available in the DSP System Toolbox.

Added features to HDL optimized FFT/IFFT blocks, including reduced latency

For details of the updates to the FFT HDL Optimized and IFFT HDL Optimized blocks, see the DSP System Toolbox release notes.

Compatibility Considerations

The FFT HDL Optimized and IFFT HDL Optimized blocks take fewer cycles to compute one frame of output than in previous releases. For instance, for the default 1024-point FFT, the latency in R2014a was 1589 cycles whereas in R2014b the latency is 1148. The latency is displayed on the block icon.

If you have manually matched latency paths in models using the R2014a version of the FFT HDL Optimized and IFFT HDL Optimized block, adjust the delay on those paths to accommodate the lower FFT latency.

HDL Reciprocal block with Newton-Raphson Implementation

The HDL Reciprocal block is available with Simulink. Use this block to implement division operations in models intended for HDL code generation. HDL Reciprocal has two Newton-Raphson HDL implementations, `ReciprocalNewton` and `ReciprocalNewtonSingleRate`. The new implementations use fewer hardware resources and can achieve higher clock frequency than Divide or Math Function HDL block implementations.

For the Divide and Math Function blocks, the names of the Newton-Raphson HDL block implementations have changed:

- `RecipNewton` is now `ReciprocalRsqrtBasedNewton`.
- `RecipNewtonSingleRate` is now `ReciprocalRsqrtBasedNewtonSingleRate`.

If you open a model from a previous release, HDL Coder automatically maps the `RecipNewton` and `RecipNewtonSingleRate` implementation names to `ReciprocalRsqrtBasedNewton` and `ReciprocalRsqrtBasedNewtonSingleRate`, respectively.

To learn about the HDL Reciprocal block, see HDL Reciprocal.

To learn about the `ReciprocalNewton` and `ReciprocalNewtonSingleRate` implementations, see HDL Reciprocal.

Serializer1D and Deserializer1D blocks

The following new blocks are available from the HDL Operations library for simulation and code generation:

- Serializer1D
- Deserializer1D

Additional blocks supported for code generation

The following blocks are now supported for code generation:

- Backlash
- Bus assignment
- Coulomb and Viscous Friction
- DC Blocker
- Dead Zone/Dead Zone Dynamic
- Discrete PID Controller
- Hit Crossing
- HDL Reciprocal
- Serializer1D/Deserializer1D
- Wrap to Zero

Composite user-defined System object support

You can generate code for user-defined System objects that contain child user-defined System objects.

System object output and update method support

You can generate code for the output and update methods in user-defined System objects that inherit from the `matlab.system.mixin.Nondirect` class.

hdlram renamed to hdl.RAM

The `hdlram` System object has been renamed to `hdl.RAM` and is now available with MATLAB. Previously, `hdlram` required a Fixed-Point Designer license.

Compatibility Considerations

If you open a design that uses `hdlram`, the software displays a warning. For continued compatibility with future releases, replace instances of `hdlram` with `hdl.RAM`.

Functionality Being Removed or Changed

Functionality	What Happens When You Use This Functionality	Use This Functionality Instead	Compatibility Considerations
HDL Streaming FFT	Still runs. The block is forwarded from hdldemolib to dsp.obselete.	FFT HDL Optimized block, which is available in the DSP System Toolbox.	This block will be removed in a future release.
HDL FFT	Still runs. This block is renamed “HDL Minimum Resource FFT”. It is forwarded from hdldemolib dsp.obselete.	FFT HDL Optimized block, which is available in the DSP System Toolbox.	This block will be removed in a future release.

Code Generation and Verification

Coding standards customization

If you enable HDL coding standard rule checking, you can enable or disable specific rules. You can specify rule parameters. For example, you can specify the maximum nesting depth for if-else statements. See HDL Coding Standard Customization.

HDL Designer script generation

You can now generate a lint tool script for Mentor Graphics HDL Designer.

To learn about HDL lint script generation for your Simulink design, see Generate an HDL Lint Tool Script.

To learn about HDL lint script generation for your MATLAB design, see Generate an HDL Lint Tool Script.

Traceable names for RAM blocks and port signals

When you generate code for a RAM block from the HDL Operations library, the name in the generated code reflects the name in the model. Similarly, port signal names in the generated code are inherited from your model.

For each RAM block of a particular size, the coder generates an HDL module. The module file name reflects the name and size of the RAM block or persistent variable in your design.

For example, suppose `DPRAM_foo` is the name of a Dual Port RAM block in your model. The generated code for the instance is:

```
u_DPRAM_foo : DualPortRAM_Wrapper_256x8b
```

The RAM module name and wrapper name also match the name of the Simulink block:

```
DualPortRAM_256x8b.vhd
DualPortRAM_Wrapper_256x8b.vhd
```

for-generate statements in generated VHDL code

When you generate VHDL code for block architectures that use replicated structures, the coder generates `for-generate` statements for better readability. For example, VHDL code generated for the Add and Product blocks uses `for-generate` statements.

Validation model generation regardless of delay balancing results

When you enable the **Generate validation model** option, HDL Coder generates the validation model even if delay balancing is unsuccessful. In previous releases, if delay balancing was unsuccessful, the coder did not generate the validation model.

Speed and Area Optimizations

Clock-rate pipelining to optimize timing in multi-cycle paths

In the Simulink to HDL workflow, for speed optimizations that insert pipeline registers, the coder identifies multi-cycle paths in your design and inserts pipeline registers at the clock rate instead of the data rate. When the optimization is in a slow-rate region or multi-cycle path of the design, clock rate pipelining enables the software to perform optimizations without adding extra latency, or by adding minimal latency. It also enables optimizations such as pipelining and floating-point library mapping inside feedback loops.

Clock-rate pipelining is enabled by default. You can disable clock-rate pipelining in one of the following ways:

- In the HDL Workflow Advisor, in the **HDL Code Generation** > **Set Code Generation Options** > **Set Advanced Options** > **Optimization** tab, select **Clock-rate pipelining**.
- At the command line, use `makehdl` or `hdlset_param` to set the `ClockRatePipelining` parameter to `off`.

For details, see Clock-Rate Pipelining.

RAM mapping for user-defined System object private properties

Private properties in user-defined System objects can map to RAM. For details, see Implement RAM Using a Persistent Array or System object Properties.

Highlighting for feedback loops that inhibit optimizations

You can generate a MATLAB script that highlights feedback loops that may inhibit delay balancing or speed and area optimizations. The script highlights feedback loops in your original model and generated model.

You can also save the highlighting information in a MATLAB script.

For details, see Find Feedback Loops.

Optimizations available for conditional-execution subsystems

The following optimizations are now supported for enabled subsystems and triggered subsystems:

- Resource sharing
- Streaming
- Constrained overclocking
- Floating-point library mapping
- Hierarchy flattening
- Delay balancing
- Automatic iterative optimization

Variable pipelining in conditional MATLAB code

HDL code generation now supports variable pipelining inside conditional MATLAB code for the MATLAB to HDL workflow and the MATLAB Function block in the Simulink to HDL workflow.

Optimizations available with UseMatrixTypesInHDL for MATLAB Function block

When you enable 2-D matrix types in the generated HDL code, for the MATLAB to HDL workflow and Simulink to HDL workflow, speed and area optimizations are available. Previously, for the MATLAB Function block, the `UseMatrixTypesInHDL` parameter was incompatible with speed and area optimizations.

IP Core Generation and Hardware Deployment

Support for Xilinx Vivado

The HDL Coder software is now tested with Xilinx Vivado Design Suite 2013.4. You can:

- Generate a custom IP core for the Zynq-7000 platform and automatically integrate it into a Vivado project for use with IP Integrator.
- Program FPGA hardware supported by Vivado using the HDL Workflow Advisor.
- Perform back-annotation analysis of your design.
- Generate synthesis scripts.

IP core generation for Altera SoC platform

You can generate a custom IP core with an AXI4 interface for the Altera SoC platform.

HDL Coder can also insert your custom IP core into a predefined Qsys project to target the Altera Cyclone V SoC development kit or Arrow SoCKit development board. The coder can connect the IP core to the ARM processor via the AXI interface within the project.

The software provides add-on support for Altera SoC hardware via the HDL Coder Support Package for Intel SoC Devices. For more details, see HDL Coder Support Package for Altera SoC Platform.

Custom HDL code for IP core generation from MATLAB

You can integrate custom HDL code, such as handwritten or legacy HDL code, into your design in the MATLAB to HDL IP core generation workflow. Use one or more `hdl.BlackBox` System objects in your MATLAB design, and add the HDL source files in the **Additional source files** field.

To learn how to use the **Additional source files** field, Generate a Board-Independent IP Core from MATLAB.

Target platform interface mapping information saved with model

For the IP core generation workflow, FPGA turnkey workflow, or Simulink Real-Time FPGA I/O workflow, when you map each of your DUT top-level ports to a platform interface, HDL Coder saves the interface mapping information as port properties in your model. The coder also saves workflow and target platform information with the model.

For DUT Inport and Outport blocks, you can set and view **IOInterface** and **IOInterfaceMapping** with the HDL Block Properties dialog box or `hdlset_param` and `hdlget_param`.

For an example that shows how to configure target platform interface settings, see https://www.mathworks.com/help/releases/R2014b/hdlcoder/examples/save-target-hardware-settings-in-model.html.

Documentation installation with hardware support package

Starting in R2014b, each hardware support package that you install comes with its own documentation. For a list of support packages available for HDL Coder, with links to documentation, see HDL Coder Supported Hardware.

R2014a

Version: 3.4

New Features

Bug Fixes

Compatibility Considerations

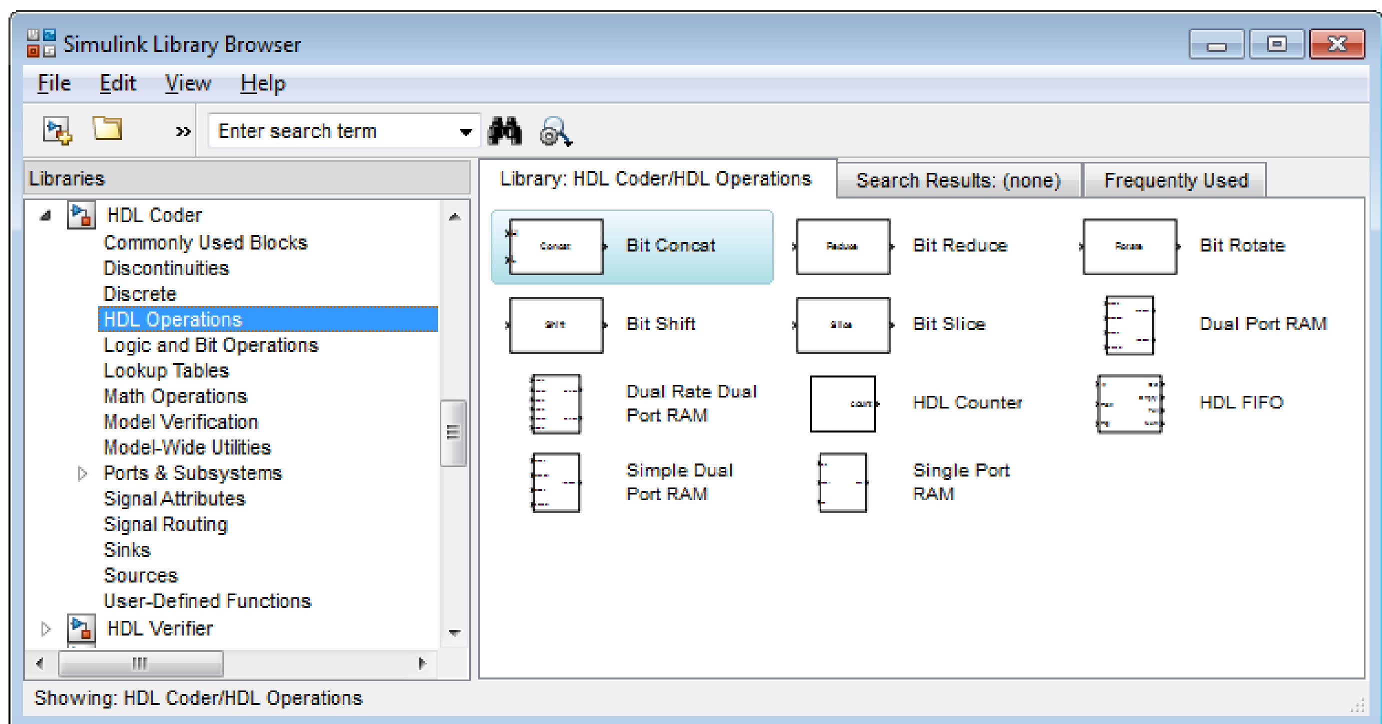
Model and Architecture Design

HDL block library in Simulink

The HDL Coder library, which contains blocks supported for HDL code generation, is available in Simulink. When you create a model using the HDL Coder library, the blocks are preconfigured with settings suitable for code generation.

The HDL Operations library, previously called `hdldemolib`, is available in Simulink as part of the HDL Coder library. Previously, the HDL Operations blocks were available only with an HDL Coder license.

To view the HDL Operations block library from the Simulink Library Browser, open the **HDL Coder** folder and select **HDL Operations**.



Persistent keyword not needed in HDL code generation

If your MATLAB code includes a System object that does not have states, you do not need to include the `persistent` keyword for HDL code generation.

For details, see Limitations of HDL Code Generation for System Objects.

Negative edge clocking

You can clock your design on the falling edge of the clock.

To generate code that clocks your design on the negative edge of the clock, in the Configuration Parameters dialog box, for **HDL Code Generation > Global Settings > Clock Edge**, select **Falling edge**.

Alternatively, at the command line, set the `ClockEdge` property to `'Falling'` using `makehdl` or `hdlset_param`.

For details, see ClockEdge.

Bidirectional port specification

You can specify bidirectional ports for Subsystem blocks that have Architecture set to `BlackBox`. In the FPGA Turnkey workflow, you can use the bidirectional ports to connect to external RAM.

In the generated code, the ports have the Verilog or VHDL `inout` keyword. However, Simulink does not support bidirectional ports, so you cannot simulate the bidirectional behavior in Simulink.

To learn more, see Specify Bidirectional Ports.

Port names in generated code match signal names

You can use the **Icon display** block parameter on Inport and Outport blocks to make your code more readable. When you set the **Icon display** parameter to **Signal name**, **Port number**, or **Port number and signal name**, the port names in the generated code match the display names of the connected signals.

ModelReference default architecture for Model block

The Model block default architecture is `ModelReference`. Previously, the default architecture was `BlackBox`.

Compatibility Considerations

When you open a model created in a previous release, a Model block in that design changes architecture from `BlackBox` to `ModelReference` if all the HDL block properties are set to default settings.

To keep the `BlackBox` architecture for Model blocks, use one of the following workarounds:

- Open the model using the current release, specify the `BlackBox` architecture for the affected Model blocks, and save the model.
- Open the model using a previous release, specify a nondefault setting for each Model block, and save the model.

Reset for timing controller

You can generate a reset port for the timing controller, which generates the clock, clock enable, and reset signals in a multirate DUT.

To generate a reset port for the timing controller, set the `TimingControllerArch` property to `resettable` using `makehdl` or `hdlset_param`.

To learn more, see Generate Reset for Timing Controller.

Reset port optimization

The coder does not generate a top-level reset port when the code generation subsystem does not contain resettable delays or blocks.

To generate code without a top-level reset port:

- Set the `ResetType` HDL block parameter to `none` for all blocks in the DUT with the `ResetType` parameter.
- For MATLAB Function blocks in your DUT, do not enable block level HDL optimizations, which insert resettable registers.

For details, see ResetType.

Functionality Being Removed or Changed

You cannot save a model that uses an attached control file to apply HDL model or block parameters.

Since the R2010a release, if you open a model that uses a control file, the software shows a warning, and updates the model by applying the HDL parameters to your model and removing the control file. For continued compatibility with future releases, save the updated model.

Functionality	What Happens When You Use This Functionality	Use This Instead	Compatibility Considerations
`hdlapplycontrolfile`	Still runs	`hdlset_param`, `hdlget_param`	Do not use control files for model or block configuration. Instead, use `hdlset_param` and `hdlget_param` to configure your model.
`hdlnewblackbox`	Still runs	`hdlset_param`, `hdlget_param`	Do not use control files for model or block configuration. Instead, use `hdlset_param` and `hdlget_param` to configure your model.
`hdlnewcontrol`	Still runs	`hdlset_param`, `hdlget_param`	Do not use control files for model or block configuration. Instead, use `hdlset_param` and `hdlget_param` to configure your model.

Functionality	What Happens When You Use This Functionality	Use This Instead	Compatibility Considerations
`hdlnewcontrolfile`	Still runs	`hdlset_param`, `hdlget_param`	Do not use control files for model or block configuration. Instead, use `hdlset_param` and `hdlget_param` to configure your model.
`hdlnewforeach`	Still runs	`hdlset_param`, `hdlget_param`	Do not use control files for model or block configuration. Instead, use `hdlset_param` and `hdlget_param` to configure your model.

Block Enhancements

Code generation for enumeration data types

You can generate code for Simulink, MATLAB, or Stateflow enumerations within your design. In the current release, you cannot generate code if your design uses enumerations at the top-level DUT ports.

To learn more about code generation support for enumerations in Simulink designs, see Enumerations.

To learn more about code generation support for enumerations in MATLAB designs, see Data Types and Scope.

Code generation for FFT HDL Optimized and IFFT HDL Optimized blocks

You can generate code for the FFT HDL Optimized and IFFT HDL Optimized blocks, which are available in the DSP System Toolbox.

Bus support improvements

You can generate code for designs that contain:

- DUT ports connected to buses.
- Buses that are not defined with a bus object.
- Nonvirtual buses.

To learn more, see Buses.

Variant Subsystem support for configurable models

You can generate code for designs containing Variant Subsystem blocks. Using Variant Subsystem blocks enables you to explore and generate code for different component implementations and design configurations.

Trigger signal can clock triggered subsystems

You can generate code that uses the trigger signals in Triggered Subsystem blocks as clocks. Using triggers as clocks enables you to partition your design into different clock regions in the generated code, but can cause a timing mismatch during testbench simulation.

For details, see Use Trigger As Clock in Triggered Subsystems.

2-D matrix types in code generated for MATLAB Function block

You can now generate 2-D matrices in HDL code when you have MATLAB matrices in a MATLAB Function block. By default, the software generates HDL vectors with additional index computation logic, which can use more area in the synthesized hardware than HDL matrices.

For details, see UseMatrixTypesInHDL.

64-bit data support

You can generate code for `uint64` and `int64` data types in MATLAB code, both in the MATLAB-to-HDL workflow and for the MATLAB Function block in the Simulink-to-HDL workflow.

MATLAB Function block ports must use `sfix64` or `ufix64` types for 64-bit data, because `uint64` and `int64` are not yet supported in Simulink.

HDL code generation from MATLAB System block

The MATLAB System block, which you use to include System objects in Simulink models, now supports HDL code generation.

For details, see MATLAB System.

System object methods in conditional code

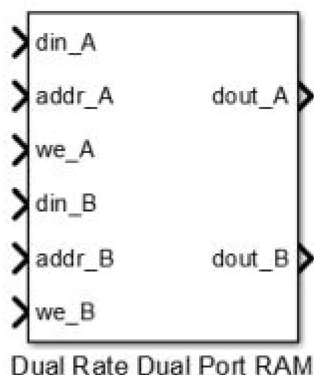
HDL code generation now supports System object `step` method calls inside conditional code regions.

Dual Rate Dual Port RAM block

A new block, Dual Rate Dual Port RAM, is available for simulation and code generation.

The Dual Rate Dual RAM supports two simultaneous read or write accesses at two Simulink rates. When you generate code, the Dual Rate Dual Port RAM block infers a dual-clock dual-port RAM in most FPGAs.

To view the block, open the HDL Operations block library.

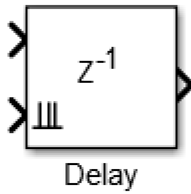


For more information about the block, see Dual Rate Dual Port RAM. For HDL code generation details, see Dual Rate Dual Port RAM.

Additional blocks and block implementations supported for code generation

The following blocks and block implementations are now supported for code generation:

- Sine, Cosine
- Enumerated Constant
- Delay with **External reset** set to `Level`.



- Multiport Switch with enumerated type at control input.

The HDL FIFO block no longer requires a DSP System Toolbox license. The HDL FIFO block is available in the HDL Operations library.

Code Generation and Verification

Errors instead of warnings for blocks not supported for code generation

If your design contains blocks or block architectures that are not supported for HDL code generation, the software shows an error and does not generate code. Previously, the software showed a warning, but still generated code, with black box interfaces for the unsupported blocks or block architectures.

Compatibility Considerations

If you want to generate code for models containing unsupported blocks or block architectures, you must **Comment out** the unsupported blocks in Simulink.

Ascent Lint script generation

You can now generate a lint tool script for Real Intent Ascent Lint.

To learn about HDL lint script generation for your Simulink design, see Generate an HDL Lint Tool Script.

To learn about HDL lint script generation for your MATLAB design, see Generate an HDL Lint Tool Script.

Incremental code generation and synthesis

In the Simulink-to-HDL workflow, and hardware-software codesign workflow, HDL Coder does not rerun code generation or synthesis tasks unless you have changed your model or other hardware-related project settings. You can save time when you want to regenerate HDL code or FPGA programming files without changing your model, code generation options, or hardware target.

Similarly, in the hardware and software codesign workflow, when you modify the embedded software part of your design without changing the hardware part, HDL Coder does not rerun HDL code generation or synthesis tasks.

When the coder skips code generation or synthesis tasks, the HDL Workflow Advisor shows a message. The message contains a link you can click to force the coder to rerun the task.

Automatic C compiler setup

In earlier releases, to set up a compiler to accelerate test bench simulation for MATLAB algorithms, you were required to run `mex -setup`. Now, the code generation software automatically locates and uses a supported installed compiler. You can use `mex -setup` to change the default compiler. See Changing Default Compiler.

Speed and Area Optimizations

RAM mapping scheduler improvements

The RAM mapping scheduling algorithm now minimizes overclocking when your MATLAB code maps to multiple RAMs. In addition, multiple persistent variables with cyclic read-write dependencies can now map to RAM.

Performance-prioritized retiming

When you enable distributed pipelining, you can specify a priority for **Distributed pipelining priority**: **Numerical integrity**, or **Performance**. In the previous release, the distributed pipelining algorithm prioritized numerical integrity.

For details, see `DistributedPipeliningPriority`.

Retiming without moving user-created design delays

You can use the **Preserve design delays** option to prevent distributed pipelining from moving design delays in your Simulink or MATLAB design. If you specify **Preserve design delays**, distributed pipelining does not move the following design delays:

- Persistent variable in MATLAB code, a MATLAB Function block, or a Stateflow Chart
- Unit Delay block
- Integer Delay block
- Memory block
- Delay block from DSP System Toolbox
- `dsp.Delay` System object from DSP System Toolbox

For details, see `PreserveDesignDelays`.

Resource sharing factor can be greater than number of shareable resources

With the resource sharing area optimization, the software shares the maximum number of shareable resources within your overclocking constraints, even if the sharing factor that you specify is not an integer divisor of the number of shareable resources. This capability can increase resource sharing, and therefore reduce area.

For example, if your subsystem has 11 multipliers, and you set **SharingFactor** to 4, the coder can implement your design with 3 multipliers: 2 multipliers shared 4 ways, and 1 multiplier shared 3 ways. In the previous release, the coder implemented the design with 5 multipliers: 2 multipliers shared 4 ways, and 3 unshared multipliers. The resulting implementation requires overclocking by a factor of 4.

To learn more, see Resource Sharing For Area Optimization.

Reduced area with multirate delay balancing

When the coder balances delays in a multirate model, it now inserts a single delay at the transition from a much faster rate to a slow rate, and passes through the data samples aligned with the slow rate. Previously, the coder inserted a large number of delays at the faster rate.

Serializer-deserializer and multiplexer-demultiplexer optimization

The coder removes back-to-back serializer-deserializer and multiplexer-demultiplexer pairs introduced by the implementation of optimizations such as resource sharing and streaming. This results in more area-efficient HDL code.

IP Core Generation and Hardware Deployment

ZC706 target for IP core generation and integration into Xilinx EDK project

You can target the Xilinx Zynq-7000 AP ZC706 Evaluation Board for IP core generation and Xilinx EDK project integration. After you install the HDL Coder Support Package for Xilinx Zynq-7000 Platform, ZC706 hardware support is available.

Automatic iterative clock frequency optimization

You can use the `hdlcoder.optimizeDesign` function to achieve either your target clock frequency or a maximum clock frequency. Based on your clock frequency goal and target device, the software iteratively generates and synthesizes code, retrieves back annotation data, and inserts delays into your Simulink model to break the critical path.

To learn more, see Automatic Iterative Optimization.

Synthesis attributes for multipliers

You can now generate code that includes synthesis attributes to specify multipliers in your design that you want to map to DSPs or logic in hardware. If you specify resource sharing, the software does not share multipliers that have different synthesis attribute settings.

For Xilinx targets, the generated code uses the `use_dsp48` attribute. For Altera targets, the generated code uses the `multstyle` attribute.

For details, see DSPStyle.

Custom HDL code for IP core generation

You can integrate custom HDL code into your design in the Simulink-to-HDL IP core generation workflow. You can integrate handwritten or legacy HDL code into an IP core that you generate from a Simulink model.

To include custom HDL code in your IP core design, use one or more Model or Subsystem blocks with Architecture set to `BlackBox`. Use the **Additional source files** field in the HDL Workflow Advisor to specify corresponding HDL file names.

For details of the IP core generation workflow, see Generate a Custom IP Core from Simulink.

Synthesis and simulation tool addition and detection after opening HDL Workflow Advisor

In the Simulink-to-HDL workflow, you can set up and add a synthesis tool without having to close and reopen the HDL Workflow Advisor. In the HDL Workflow Advisor, in the **Set Target > Set Target Device and Synthesis Tool** task, click **Refresh** to detect and add the new tool.

You can also set up and add a simulation tool after creating a MATLAB-to-HDL project without having to close and reopen the project. In the HDL Workflow Advisor, in the **HDL Verification > Verify with HDL Test Bench** task, click **Refresh list** to detect and add the new tool.

xPC Target is Simulink Real-Time

The **xPC Target FPGA I/O** workflow is now called the **Simulink Real-Time FPGA I/O** workflow. This change reflects the xPC Target™ product name change to Simulink Real-Time. For details about the product name change, see New product that combines the functionality of xPC Target and xPC Target Embedded Option.

Updates to supported software

HDL Coder has been tested with:

- Xilinx ISE 14.6
- Altera Quartus II 13.0 SP1

For a list of supported third-party tools and hardware, see Supported Third-Party Tools and Hardware.

R2013b

Version: 3.3

New Features

Bug Fixes

Compatibility Considerations

Model and Architecture Design

Model reference support and incremental code generation

You can generate HDL code from referenced models using the Model block. To use a referenced model in a subsystem intended for code generation, in the HDL Block Properties dialog box, set **Architecture** to **ModelReference**.

The coder incrementally generates code for referenced models according to the **Configuration Parameters dialog box > Model Referencing pane > Rebuild** options. However, the coder treats **If any changes detected** and **If any changes in known dependencies detected** as the same. For example, if you set **Rebuild** to either **If any changes detected** or **If any changes in known dependencies detected**, the coder regenerates code for referenced models only when the referenced models have changed.

To learn more, see Model Referencing for HDL Code Generation.

Code generation for subsystems containing Altera DSP Builder blocks

You can now generate HDL code for subsystems that include blocks from the Altera DSP Builder Advanced Blockset.

For details, see Create an Altera DSP Builder Subsystem.

To see an example that shows HDL code generation for an Altera DSP Builder subsystem, see Using Altera DSP Builder Advanced Blockset with HDL Coder.

Module or entity generation for local functions in MATLAB Function block

You can now generate instantiable Verilog modules or VHDL entities when you generate code for local functions in a MATLAB Function block, or for functions on your path that are called from within a MATLAB Function block.

To enable this feature, in the HDL Block Properties dialog box, set **InstantiateFunctions** to **on**. For details, see InstantiateFunctions.

Reset port optimization

The coder no longer generates a top level reset port when the `ResetType` HDL block parameter is set to `none` for all RAM blocks in the DUT.

In previous releases, the generated code included a reset port even when the RAM reset logic was suppressed.

Load constants from MAT-files

HDL Coder now generates code for the `coder.load` function, which you can use to load compile-time constants from a MAT-file. You no longer have to manually type in constants that were stored in a MAT-file.

To learn how to use `coder.load` for HDL code generation, see Load constants from a MAT-File.

Block Enhancements

Code generation for user-defined System objects

You can now generate HDL code from user-defined System objects written in MATLAB. System objects enable you to create reusable HDL IP.

The `step` method specifies the HDL implementation behavior. It is the only System object method supported for HDL code generation.

User-defined System objects are not supported for automatic fixed-point conversion.

To learn how to define a custom System object, see Generate Code for User-Defined System Objects.

Bus signal inputs and outputs for MATLAB Function block and Stateflow charts

MATLAB Function blocks and Stateflow charts with bus signal inputs or outputs are now supported for code generation. The bus must be defined with a bus object.

HDL Counter has specifiable start value

You can now specify a start value for the HDL Counter block. When the counter initializes or wraps around, it counts from the specified start value.

Maximum 32-bit address for RAM

For the Single Port RAM block, Simple Dual Port RAM block, Dual Port RAM block, and `hdlram` System object, the maximum address width is now 32 bits. For more information, see:

- `hdlram`
- RAM Blocks

Removing HDL Support for NCO Block

HDL support for the NCO block will be removed in a future release. Use the NCO HDL Optimized block instead.

Compatibility Considerations

In the current release, if you generate HDL code for the NCO block, a warning message appears. In a future release, any attempt to generate HDL code for the NCO block will cause an error.

Code Generation and Verification

Coding style improvements according to industry standard guidelines

The coder now follows these industry standard coding style guidelines when generating HDL code:

- Division by a power of 2 becomes a bit shift operation.
- Constants with double data types in the original design are automatically converted to their canonical fixed-point types as long as there is no loss of precision.
- SystemVerilog keywords are treated as reserved words.
- Intermediate signals and latches are reduced when **HDLCodingStandard** is set to **Industry**.
- Real data types generate warnings, except when you target an FPGA floating-point library.

Coding standard report target language enhancement and text file format

HDL Coder now generates the coding standard report according to target language. Coding standard errors, warnings, and messages that do not pertain to your target language no longer appear in the report.

The coding standard report is generated in text file format, in addition to HTML format, to enable easier comparison between multiple runs.

UI for SpyGlass, Leda, and custom lint tool script generation

You can now use the UI to generate Atrenta SpyGlass, Synopsys® Leda, or custom lint scripts in the Simulink-to-HDL and MATLAB-to-HDL workflows.

To learn about HDL lint script generation for your Simulink design, see Generate an HDL Lint Tool Script.

To learn about HDL lint script generation for your MATLAB design, see Generate an HDL Lint Tool Script.

File I/O to read test bench data in VHDL and Verilog

You can now specify the generated VHDL or Verilog test bench to use file I/O to read input stimulus and output response data during simulation, instead of including data constants in the test bench code. Doing so improves scalability for designs requiring long simulations and large test vectors.

This feature is available for Simulink-to-HDL and MATLAB-to-HDL code generation.

To learn about test bench generation with file I/O in the Simulink-to-HDL workflow, see Generate Test Bench With File I/O.

To learn about test bench generation with file I/O in the MATLAB-to-HDL workflow, see Generate Test Bench With File I/O.

Floating point for FIL and HDL cosimulation test bench generation

With the R2013b release, HDL Coder HDL workflow advisor for Simulink supports double and single data types on the DUT interface for test bench generation using HDL Verifier.

Fixed-point file name change

The suffix for generated fixed-point files is now `_fixpt`. Previously, the suffix was `_FixPt`.

Compatibility Considerations

If you have MATLAB-to-HDL projects from previous releases that depend on the generated fixed-point file name, you can use the `FixPtFileNameSuffix` property to set the suffix to `_FixPt`.

Speed and Area Optimizations

RAM inference in conditional MATLAB code

The coder now infers RAM from persistent array variables accessed within conditional statements, such as if-else or switch-case statements, for both MATLAB designs and MATLAB Function blocks in Simulink.

If you have nested conditional statements, the persistent array variables can map to RAM if accessed in the topmost conditional statement, but cannot map to RAM if accessed in a lower level nested conditional statement.

Coding style for improved ROM mapping

The coder now automatically inserts a no-reset register at the output of a constant matrix access. Many synthesis tools infer a ROM from this code pattern. For details, see Map Matrices to ROM.

Pipeline registers between adder or multiplier and rounding or saturation logic

The coder now places a pipeline register between an adder or multiplier and associated rounding or saturation logic when distributing pipelining registers. This register placement can significantly improve clock frequency.

Distributed pipelining improvements with loop unrolling in MATLAB Function block

When you enable distributed pipelining for a MATLAB Function block without persistent variables, set the **Loop Optimization** option to `Unrolling` for better timing results.

IP Core Generation and Hardware Deployment

IP core integration into Xilinx EDK project for ZC702 and ZedBoard

When you generate an IP core from your MATLAB design or Simulink model, HDL Coder can automatically insert the IP core into a predefined Xilinx ZC702 or ZedBoard EDK project for the Zynq-7000 platform. The coder automatically connects the IP core to the AXI interface and ARM processor in the EDK project.

For an overview of the hardware and software codesign workflow, see Hardware and Software Codesign Workflow.

For an example that shows how to deploy your MATLAB design in hardware and software on the Zynq-7000 platform, see Getting Started with HW/SW Co-design Workflow for Xilinx Zynq Platform.

For an example that shows how to deploy your Simulink model in hardware and software on the Zynq-7000 platform, see Getting Started with HW/SW Co-design Workflow for Xilinx Zynq Platform.

FPGA Turnkey and IP Core generation in MATLAB to HDL workflow

You can now generate a custom IP core with an AXI4-Lite or AXI4-Stream Video interface from a MATLAB design. You can integrate the generated IP core into a larger design in your Xilinx EDK project.

You can also automatically program an Altera or Xilinx FPGA development board with code generated from your MATLAB design, using the HDL Workflow Advisor FPGA Turnkey workflow. To learn how to use this workflow, see Program Standalone FPGA with FPGA Turnkey Workflow and Getting Started with FPGA Turnkey Workflow.

Previously, IP core generation and FPGA Turnkey were available only for the Simulink to HDL workflow.

Synthesis tool addition and detection after MATLAB-to-HDL project creation

You can now set up and add a synthesis tool after creating a MATLAB-to-HDL project without having to close and reopen the project. In the HDL Workflow Advisor, in the **Set Code Generation Target** task, click **Refresh list** to detect and add the new tool. For details, see Add Synthesis Tool for Current MATLAB Session.

Synthesis script generation for Microsemi Libero and other synthesis tools

You can now generate a Microsemi Libero or custom synthesis tool script during Simulink-to-HDL and MATLAB-to-HDL code generation.

In the MATLAB-to-HDL workflow, you can now generate synthesis tool scripts customized for Xilinx ISE, Microsemi Libero, Mentor Graphics Precision, Altera Quartus II, and Synopsys Synplify Pro®. The coder populates the scripts with default options, but you can further customize the scripts as needed. In previous releases, you had to enter the synthesis tool commands manually. For details, see Generate Synthesis Scripts.

Floating-point library mapping for mixed floating-point and fixed-point designs

When you enable FPGA target-specific floating-point library mapping, you can now generate code from a design containing both floating-point and fixed-point components. The coder determines whether to map to a floating-point IP block based on the data types in your model.

xPC Target FPGA I/O workflow separate from FPGA Turnkey workflow

The HDL Workflow Advisor target workflow that programs Speedgoat boards to run with xPC Target is now called the **xPC Target FPGA I/O** workflow. This workflow is separate from the FPGA Turnkey workflow for Altera and Xilinx FPGA boards.

For an example that shows how to use the xPC Target FPGA I/O workflow, see Generate Simulink Real-Time Interface for Speedgoat Boards.

AXM-A75 AD/DA module for Speedgoat IO331 FPGA board

The AXM-A75 AD/DA module for Speedgoat IO331 FPGA board is now available as a hardware target for the **xPC Target FPGA I/O** workflow.

Speedgoat IO321 and IO321-5 target hardware support

The xPC Target FPGA I/O workflow now supports the Speedgoat IO321 board and its variant, Speedgoat IO321-5, as separate hardware targets. Previously, the name of the IO321-5 board was IO325.

To learn more about the IO321 and IO321-5 boards, see Speedgoat IO321.

Support package for Xilinx Zynq-7000 platform

Generate a custom IP core for the ZC702 or ZedBoard on the Xilinx Zynq-7000 platform using the IP core generation workflow.

To install this support package for MATLAB-to-HDL code generation:

1. In the HDL Workflow Advisor, in the **Select Code Generation Target** task, set **Workflow** to **IP Core Generation**.
2. For **Platform**, select **Get more**.
3. Use Support Package Installer to install the HDL Coder Support Package for Xilinx Zynq-7000 Platform.

To install this support package for Simulink-to-HDL code generation:

1. In the HDL Workflow Advisor, in the **Set Target** > **Set Target Device and Synthesis Tool** task, set **Target workflow** to **IP Core Generation**.
2. For **Target platform**, select **Get more**.
3. Use Support Package Installer to install the HDL Coder Support Package for Xilinx Zynq-7000 Platform.

Support package for Altera FPGA boards

Program Altera FPGA boards with your generated HDL code using the FPGA Turnkey workflow.

To install this support package for MATLAB-to-HDL code generation:

1. In the HDL Workflow Advisor, in the **Select Code Generation Target** task, set **Workflow** to **FPGA Turnkey**.
2. For **Platform**, select **Get more boards**.
3. Use Support Package Installer to install the HDL Coder Support Package for Altera FPGA Boards.

To install this support package for Simulink-to-HDL code generation:

1. In the HDL Workflow Advisor, in the **Set Target > Set Target Device and Synthesis Tool** task, set **Target workflow** to **FPGA Turnkey**.
2. For **Target platform**, select **Get more boards**.
3. Use Support Package Installer to install the HDL Coder Support Package for Altera FPGA Boards.

Compatibility Considerations

Previous versions of HDL Coder had built-in support for Altera FPGA boards in the FPGA Turnkey workflow. The current version of HDL Coder does not have built-in support for Altera FPGA boards. To get support for Altera FPGA boards, install the HDL Coder Support Package for Altera FPGA Boards.

Support package for Xilinx FPGA boards

Program Xilinx FPGA boards with your generated HDL code using the FPGA Turnkey workflow.

To install this support package for MATLAB-to-HDL code generation:

1. In the HDL Workflow Advisor, in the **Select Code Generation Target** task, set **Workflow** to **FPGA Turnkey**.
2. For **Platform**, select **Get more boards**.
3. Use Support Package Installer to install the HDL Coder Support Package for Xilinx FPGA Boards.

To install this support package for Simulink-to-HDL code generation:

1. In the HDL Workflow Advisor, in the **Set Target > Set Target Device and Synthesis Tool** task, set **Target workflow** to **FPGA Turnkey**.
2. For **Target platform**, select **Get more boards**.
3. Use Support Package Installer to install the HDL Coder Support Package for Xilinx FPGA Boards.

Compatibility Considerations

Previous versions of HDL Coder had built-in support for Xilinx FPGA boards in the FPGA Turnkey workflow. The current version of HDL Coder does not have built-in support for Xilinx FPGA boards. To get support for Xilinx FPGA boards, install the HDL Coder Support Package for Xilinx FPGA Boards.

Additional FPGA board support for FIL verification, including Xilinx KC705 and Altera DSP Development Kit, Stratix V edition

Several FPGA boards have been added to the HDL Verifier FPGA board support packages, including Xilinx KC705 and Altera DSP Development Kit, Stratix V edition. You can select these boards for FIL verification using the HDL workflow advisor for Simulink.

R2013a

Version: 3.2

New Features

Bug Fixes

Compatibility Considerations

Model and Architecture Design

Code generation for System objects in a MATLAB Function block

You can now generate code from a MATLAB Function block containing System objects.

For details, see System Objects under MATLAB Language Support, in MATLAB Function Block Usage.

Output folder structure includes model name

When you generate code for a subsystem within a model, the output folder structure now includes the model name.

For example, if you generate code for a subsystem in a model, `Mymodel`, the output folder is `hdlsrc/Mymodel`.

Compatibility Considerations

If you have scripts that depend on a specific output folder structure, you must update them with the new structure.

Prefix for module or entity name

You can now specify a prefix for every module or entity name in the generated HDL code. This feature helps you to avoid name clashes when you want to have multiple instances of the HDL code generated from the same block. For details, see `ModulePrefix`.

Functionality being removed

Property Name	What Happens When You Use This Property?	Use This Property Instead	Compatibility Considerations
`RAMStyle`	Error	`RAMArchitecture`	The new property syntax differs. Replace existing instances of `RAMStyle` with the correct `RAMArchitecture` syntax.
`GainImpls`	Error	ConstMultiplierOptimization	The new property syntax differs. Replace existing instances of `GainImpls` with the correct ConstMultiplierOptimization syntax.

Block Enhancements

Single rate Newton-Raphson architecture for Sqrt, Reciprocal Sqrt

The Sqrt, Reciprocal Sqrt, reciprocal Divide, and reciprocal Math Function blocks now have a single-rate pipelined architecture. The new architecture enables you to use the high-speed Newton-Raphson algorithm without multirate or overclocking.

The following table lists each block with its new block implementation.

Block	Implementation Name	Details
Sqrt	`SqrtNewtonSingleRate`	See Sqrt.
Reciprocal Sqrt	`RecipSqrtNewtonSingleRate`	See Reciprocal Sqrt.
Divide (reciprocal)	`RecipNewtonSingleRate`	See Divide (reciprocal).
Math Function (reciprocal)	`RecipNewtonSingleRate`	See Math Function (reciprocal).

Additional System objects supported for code generation

Effective with this release, the following System objects provide HDL code generation:

- `comm.HDLCRCGenerator`
- `comm.HDLCRCDetector`
- `comm.HDLRSEncoder`
- `comm.HDLRSDecoder`
- `dsp.HDLNCO`

Additional blocks supported for code generation

The following blocks are now supported for HDL code generation:

- NCO HDL Optimized
- Bias
- Relay
- Dot Product
- Sum with more than two inputs with different signs
- MinMax with multiple input data types

Code Generation and Verification

Static range analysis for floating-point to fixed-point conversion

The coder can now use static range analysis to derive fixed-point data types for your floating-point MATLAB code.

The redesigned interface for floating-point to fixed-point conversion enables you to use simulation with multiple test benches, static range analysis, or both, to determine fixed-point data types for your MATLAB variables.

For details, see Automated Fixed-Point Conversion.

Cosimulation and FPGA-in-the-Loop for MATLAB HDL code generation

With the MATLAB HDL Workflow Advisor, the HDL Verification step includes automation for the following workflows:

- Verify with HDL Test Bench: Create a standalone test bench. You can choose to simulate a model using ModelSim or Incisive® with a vector file created by the Workflow Advisor.
- Verify with Cosimulation: Cosimulate the DUT in ModelSim or Incisive with the test bench in MATLAB.
- Verify with FPGA-in-the-Loop: Create the FPGA programming file and test bench, and, optionally, download it to your selected development board.

You must have an HDL Verifier license to use these workflows.

HDL coding standard report and lint tool script generation

You can now generate a report that shows how well your generated HDL code conforms to an industry coding standard. Errors and warnings in the report link to elements in your original design so you can fix problems.

You can also generate third-party lint tool scripts to use to check your generated HDL code. In this release, you can generate Leda, SpyGlass, and generic scripts.

To learn more about the coding standard report, see HDL Coding Standard Report.

To learn how to generate a coding standard report and lint tool script in the Simulink to HDL workflow, see:

- Generate an HDL Coding Standard Report
- Generate an HDL Lint Tool Script

To learn how to generate a coding standard report and lint tool script in the MATLAB to HDL workflow, see:

- Generate an HDL Coding Standard Report
- Generate an HDL Lint Tool Script

File I/O to read test bench data in Verilog

You can now specify the generated HDL test bench to use file I/O to read input stimulus and output response data during simulation, instead of including data constants in the test bench code. Doing so improves scalability for designs needing long simulations.

This feature is available when Verilog is the target language.

For details, see Test Bench Generation with File I/O.

Speed and Area Optimizations

User-specified pipeline insertion for MATLAB variables

You can now specify pipeline register insertion for variables in your MATLAB code. This feature is available in both the MATLAB to HDL workflow and the MATLAB Function block.

To learn how to pipeline variables in the MATLAB to HDL workflow, see Pipeline MATLAB Variables.

To learn how to pipeline variables in the MATLAB Function block, see Pipeline Variables in the MATLAB Function Block.

Resource sharing and streaming without over clocking

You can now constrain the resource sharing and streaming optimizations to prevent or reduce overclocking. The coder optimizes your design based on two parameters that you specify: maximum oversampling ratio, `MaxOversampling`, and maximum computation latency, `MaxComputationLatency`.

For single-rate resource sharing or streaming, you can set `MaxOversampling` to 1.

To learn more about constrained overclocking, maximum oversampling ratio, and maximum computation latency, see:

- Optimization With Constrained Overclocking
- Maximum Oversampling Ratio
- Maximum Computation Latency

Resource sharing for the MATLAB Function block

You can now specify a resource sharing factor for the MATLAB Function block to share multipliers in the MATLAB code.

For details, see Resource Sharing and Specify Resource Sharing.

Finer control for delay balancing

You can now disable delay balancing for a subsystem within your DUT subsystem.

For details, see Balance Delays.

Complex multiplication optimizations in the Product block

You can now share multipliers used in a single complex multiplication in the Product block. Distributed pipelining can also move registers between the multiply and add stages of a complex multiplication.

IP Core Generation and Hardware Deployment

Generation of custom IP core with AXI4 interface

You can now generate custom IP cores with an AXI4-Lite or AXI4-Stream Video interface. You can integrate these custom IP cores with your design in a Xilinx EDK environment for the Xilinx Zynq-7000 Platform.

For more details, see Custom IP Core Generation.

To view an example that shows how to generate a custom IP core, at the command line, enter:

```
hdlcoder_ip_core_led_blinking
```

Coprocessor synchronization in FPGA Turnkey and IP Core Generation workflows

The coder can now automatically synchronize communication and data transfers between your processor and FPGA. You can use the new **Processor/FPGA synchronization mode** in the FPGA Turnkey workflow with xPC Target, or when you generate a custom IP core.

For more details, see Processor and FPGA Synchronization.

Speedgoat IO331 Spartan-6 FPGA board for FPGA Turnkey workflow

You can now use the Speedgoat IO331 Spartan-6 FPGA board in the FPGA Turnkey workflow with xPC Target.

You must have an xPC Target license to use this feature.

R2012b

Version: 3.1

New Features

Input parameter constants and structures in floating-point to fixed-point conversion

Floating-point to fixed-point conversion now supports structures and constant value inputs.

RAM, biquad filter, and demodulator System objects

HDL RAM System object

With release 2012b, you can use the `hdlram` System object for modeling and generating fixed-point code for RAMs in FPGAs and ASICs. The `hdlram` System object provides simulation capability in MATLAB for Dual Port, Simple Dual Port, and Single Port RAM. The System object also generates RTL code that can be inferred as a RAM by most synthesis tools.

To learn how to model and generate RAMs using the `hdlram` System object, see Model and Generate RAM with hdlram.

HDL System object support for biquad filters

HDL support has been added for the following System object:

- `dsp.BiquadFilter`

HDL support with demodulator System objects

HDL support has been added for the following System objects:

- `comm.BPSKDemodulator`
- `comm.QPSKDemodulator`
- `comm.PSKDemodulator`
- `comm.RectangularQAMDemodulator`
- `comm.RectangularQAMModulator`

Generation of MATLAB Function block in the MATLAB to HDL workflow

You can now generate a MATLAB Function block during the MATLAB to HDL workflow. You can use the generated block for further design, simulation, and code generation in Simulink.

For details, see MATLAB Function Block Generation.

HDL code generation for Reed Solomon encoder and decoder, CRC detector, and multichannel Discrete FIR filter

HDL code generation

In R2012b, HDL code generation support has been added for the following blocks:

- General CRC Syndrome Detector HDL Optimized

  For an example of using the HDL-optimized CRC generator and detector blocks, see Using HDL Optimized CRC Library Blocks.

- Integer-Input RS Encoder HDL Optimized
- Integer-Output RS Decoder HDL Optimized

Multichannel Discrete FIR filters

The Discrete FIR Filter block accepts vector input and supports multichannel implementation for better resource utilization.

- With vector input and channel sharing option `on`, the block supports multichannel fully parallel FIR, including direct form FIR, sym/antisym FIR, and FIRT. Support for all implementation parameters, for example: multiplier pipeline, add pipeline registers.
- With vector input and channel sharing option `off`, the block instantiates one filter implementation for each channel. If the input vector size is N, N identical filters are instantiated.

Applies to the fully parallel architecture option for FIR filters only.

Targeting of custom FPGA boards

The FPGA Board Manager and New FPGA Board Wizard allow you to add custom board information so that you can use FIL simulation with an FPGA board that is not one of the pre-registered boards. See FPGA Board Customization.

Optimizations for MATLAB Function blocks and black boxes

The resource sharing optimization now operates on MATLAB Function blocks. For details, see Specify Resource Sharing.

The delay balancing and distributed pipelining optimizations now operate on black box subsystems. To learn how to specify latency and enable distributed pipelining for a black box subsystem, see Customize the Generated Interface.

Generate Xilinx System Generator Black Box block from MATLAB

You can now generate a Xilinx System Generator Black Box block during the MATLAB-to-HDL workflow. You can use the generated block for further design, simulation, and code generation in Simulink.

For details, see Xilinx System Generator Black Box Block Generation.

Save and restore HDL-related model parameters

Two new functions, `hdlsaveparams` and `hdlrestoreparams`, enable you to save and restore nondefault HDL-related model parameters. Using these functions, you can perform multiple iterations on your design to optimize the generated code.

For details, see `hdlsaveparams` and `hdlrestoreparams`.

Command-line interface for MATLAB-to-HDL code generation

You can now convert your MATLAB code from floating-point to fixed-point and generate HDL code using the command-line interface.

To learn how to use the command line interface, open the tutorial:

```
showdemo mlhdlc_tutorial_cli
```

User-specifiable clock enable toggle rate in test bench

You can now specify the clock enable toggle rate in your test bench to match your input data rate or improve test coverage.

To learn how to specify your test bench clock enable toggle rate, see Test Bench Clock Enable Toggle Rate Specification.

RAM mapping for dsp.Delay System object

The `dsp.Delay` System object now maps to RAM if the RAM mapping optimization is enabled and the delay size meets the RAM mapping threshold.

To learn how to map the `dsp.Delay` System object to RAM, see Map Persistent Arrays and dsp.Delay to RAM.

Code generation for Repeat block with multiple clocks

You can now generate code for the DSP System Toolbox Repeat block in a model with multiple clocks.

Automatic verification with cosimulation using HDL Coder

With the HDL Coder HDL Workflow Advisor, you can automatically verify using your Simulink test bench with the new verification step **Run Cosimulation Test Bench**. During verification, the HDL Workflow Advisor and HDL Verifier verify the generated HDL using cosimulation between the HDL Simulator and the Simulink test bench. See Automatic Verification in the HDL Verifier documentation.

ML605 Board Added To Turnkey Workflow

The Xilinx Virtex-6 FPGA ML605 board has been added for Turnkey Workflow in the HDL Workflow Advisor.

R2012a

Version: 3.0

New Features

Compatibility Considerations

Product Name Change and Extended Capability

HDL Coder replaces Simulink HDL Coder and adds the HDL code generation capability directly from MATLAB.

To generate HDL code from MATLAB, you need the following products:

- HDL Coder
- MATLAB Coder
- Fixed-Point Toolbox™
- MATLAB

To generate HDL code from Simulink, you need the following products:

- HDL Coder
- MATLAB Coder
- Fixed-Point Toolbox
- Simulink Fixed Point™
- Simulink
- MATLAB

Code Generation from MATLAB

You can now generate HDL code directly from MATLAB code.

This workflow provides:

- Verilog or VHDL code generation from MATLAB code.
- Test bench generation from MATLAB scripts.
- Automated conversion from floating point code to fixed point code.
- Automated HDL verification through integration with ModelSim and ISim.
- HDL code generation for a subset of System objects from the Communications Toolbox and DSP System Toolbox.
- A traceability report mapping generated HDL code to your original MATLAB code.

The MATLAB to HDL workflow provides the following automated HDL code optimizations:

- Area optimizations: RAM mapping for persistent array variables, loop streaming, resource sharing, and constant multiplier optimization.
- Speed optimizations: input pipelining, output pipelining, and distributed pipelining.

The coder can also generate a resource utilization report, with RAM usage and the number of adders, multipliers, and muxes in your design.

See also HDL Code Generation from MATLAB.

Code Generation from Any Level of Subsystem Hierarchy

You can now generate HDL code from a subsystem at any level of the subsystem hierarchy. In previous releases, you could generate HDL code from the top-level subsystem only.

This feature also enables you to check any level subsystem for code generation compatibility, and to automatically generate a testbench.

Automated Subsystem Hierarchy Flattening

You can now generate code with a flattened subsystem hierarchy, while preserving hierarchy in nested subsystems.

This option enables you to perform more extensive area and speed optimization on the flattened component. It also enables you to reduce the number of HDL output files.

See also Hierarchy Flattening.

Support for Discrete Transfer Fcn Block

You can now generate HDL code from the Discrete Transfer Fcn block.

For details, see Discrete Transfer Fcn Requirements and Restrictions.

User Option to Constrain Registers on Output Ports

A new property, `ConstrainedOutputPipeline`, enables you to specify the number of registers you wish to have on an output port without introducing additional delay on the input to output path. The coder redistributes existing delays within your design to try to meet the constraint. This behavior is different from the `OutputPipeline` property, which introduces additional delay on the input to output path.

If the coder is unable to meet the constraint using existing delays, it reports the difference between the number of desired and actual output registers in the timing report.

Distributed Pipelining for Sum of Elements, Product of Elements, and MinMax Blocks

The Sum of Elements, Product of Elements, and MinMax blocks can now participate in distributed pipelining if their architecture is set to `Tree`.

MATLAB Function Block Enhancements

Multiple Accesses to RAMs Mapped from Persistent Variables

You can now perform multiple reads and writes to a persistent variable, and the persistent variable will still be mapped to RAM. In previous releases, a RAM mapped from a persistent variable could be accessed only once.

Streaming for MATLAB Loops and Vector Operations

You can now perform streaming on MATLAB loops and loops created from vector operations for improved area efficiency.

For details, see Loop Optimization.

Loop Unrolling for MATLAB Loops and Vector Operations

You can now unroll user-written MATLAB loops and loops created from vector operations. This enables the coder to perform area and speed optimizations on the unrolled loops.

For details, see Loop Optimization.

Automated Code Generation from Xilinx System Generator for DSP Blocks

You can now automatically generate HDL code from subsystems containing Xilinx System Generator for DSP blocks.

For details, see Code Generation with Xilinx System Generator Subsystems.

Altera Quartus II 11.0 Support in HDL Workflow Advisor

The HDL Workflow Advisor has now been tested with Altera Quartus II 11.0. In previous releases, the HDL Workflow Advisor was tested with Altera Quartus II 9.1.

Automated Mapping to Xilinx and Altera Floating Point Libraries

The coder can now map Simulink floating point operations to synthesizable floating point Altera Megafunctions and Xilinx LogiCORE IP Floating Point Operator v5.0 blocks. To learn more, see FPGA Target-Specific Floating-Point Library Mapping.

For a list of supported Altera Megafunction blocks, see Supported Altera Floating-Point Library Blocks.

For a list of supported Xilinx LogicCORE IP blocks, see Supported Xilinx Floating-Point Library Blocks.

Vector Data Type for PCI Interface Data Transfers Between xPC Target and FPGA

In the FPGA Turnkey workflow, you can now use vector data types with the **Scalarize Vector Ports** option to automatically generate PCI DMA transfers on the PCI interface between xPC Target and FPGA. You no longer need to manually insert multiplexers, demultiplexers and provide synchronization logic for vector data transfers.

If the **Scalarize Vector Ports** option is disabled when the code generation subsystem has vector ports, the coder displays an error.

New Global Property to Select RAM Architecture

There is a new global property, `RAMArchitecture`, that enables you to generate RAMs either with or without clock enables. This property applies to every RAM in your design, and replaces the block level property, `RAMStyle`. By default, RAMs are generated with clock enables.

To generate RAMs without clock enables, set RAMArchitecture to `'WithoutClockEnable'`. To generate RAMs with clock enables, either use the default, or set `RAMArchitecture` to `'WithClockEnable'`. For more information, see Implement RAMs With or Without Clock Enable.

Compatibility Considerations

The coder now ignores the block level property, `RAMStyle`.

If a block's `RAMStyle` property is set, the coder generates a warning.

Turnkey Workflow for Altera Boards

HDL Workflow Advisor now supports Altera FPGA design software and the following Altera development kits and boards:

- 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

This workflow has been tested with Altera Quartus II 11.0.

HDL Support For Bus Creator and Bus Selector Blocks

Release R2012a provides HDL code generation for the Bus Creator and Bus Selector blocks. You must use these blocks for your buses if you want HDL support.

HDL Support For HDL CRC Generator Block

Release R2012a provides HDL code generation for the new HDL CRC Generator block.

HDL Support for Programmable Filter Coefficients

When using filter blocks to generate HDL code, you can specify coefficients from input port(s). This feature applies to FIR and BiQuad filter blocks only. Fully Parallel and all serial architectures are supported.

Follow these directions to use programmable filters:

1. Select `Input port(s)` as coefficient source from the filter block mask.
2. Connect the coefficient port with a vector signal.
3. Specify the implementation architecture and parameters from the HDL Coder property interface.
4. Generate HDL code.

Notes

- For fully parallel implementations, the coefficients ports are connected to the dedicated MAC directly.
- For serial implementation, the coefficients ports first go to a mux, and then to the MAC. The mux decides the coefficients that used at current time instant
- For Discrete FIR filters, this feature is not supported under the following conditions:
  - Implementations having coefficients specified by dialog parameters (for example, complex input and coefficients with serial architecture)
  - Filters using DA architecture
  - CoeffMultipliers specified as `csd` or `factored-csd`
- For Biquad filters, this feature is not supported when CoeffMultipliers are specified as `csd` or `factored-csd`.

Synchronous Multiclock Code Generation for CIC Decimators and Interpolators

You can specify multiple clocks in one of the following ways:

- Use the model-level parameter `ClockInputs` with the function `makehdl` and specify the value as 'Multiple'.
- In the Clock settings section of the **Global Settings** pane in the HDL Code Generation Configuration Parameters dialog box, set **Clock inputs** to `Multiple`.

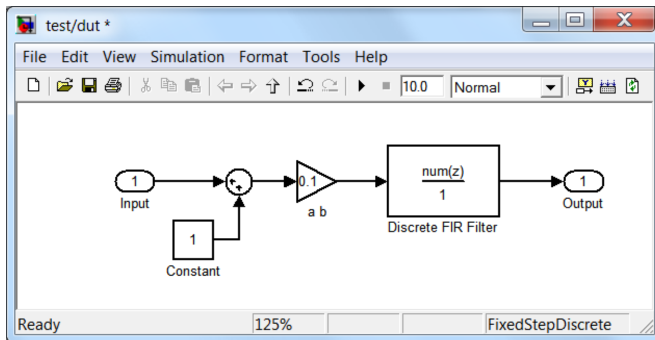
When you use single-clock mode, HDL code generated from multirate models uses a single master clock that corresponds to the base rate of the DUT. When you use multiple-clock mode, HDL code generated from multirate models use one clock input for each rate in the DUT. The number of timing controllers generated in multiple-clock mode depends on the design in the DUT.

The `ClockInputs` parameter supports the values 'Single' and 'Multiple', where the default is 'Single'. In the default single-clock mode, the coder behavior is unchanged from previous releases.

Filter Block Resource Report Participation

Resource reports include the HDL resource usage for filter blocks. The report includes adders, subtractors, multipliers, multiplexers, registers. This feature covers all filter blocks, and all implementations for the block.

You can turn on the report feature using the command line (`ResourceReport`) or GUI (**Generate resource utilization report**). The following illustrations show a report for a model that includes a Discrete FIR Filter block.



High-level Resource Utilization Report for test

File Edit View Go Debug Desktop Window Help

Location: file:///H:/Documents/MATLAB/hdlsrc/html/test/test_bill_of_materials.html

Resource Utilization Report for test

Summary

Multipliers	2
Adders/Subtractors	2
Registers	7
RAMs	0
Multiplexers	3

Detailed Report

[Expand all] [Collapse all]

Report for Subsystem: dut

Multipliers (2)

[-] 12x12-bit Multiply : 1

- a b

[-] 16x16-bit Multiply : 1

- Discrete_FIR_Filter

Adders/Subtractors (2)

[-] 32x32-bit Adder : 1

- Sum

[-] 34x34-bit Adder : 1

- Discrete_FIR_Filter

Registers (7)

32-bit Register : 1
[-] 16-bit Register : 4

- Discrete_FIR_Filter

[-] 33-bit Register : 2

- Discrete_FIR_Filter

Multiplexers (3)

[-] 33-bit 2-to-1 Multiplexer : 1

- Discrete_FIR_Filter

[-] 16-bit 4-to-1 Multiplexer : 2

- Discrete_FIR_Filter

Done

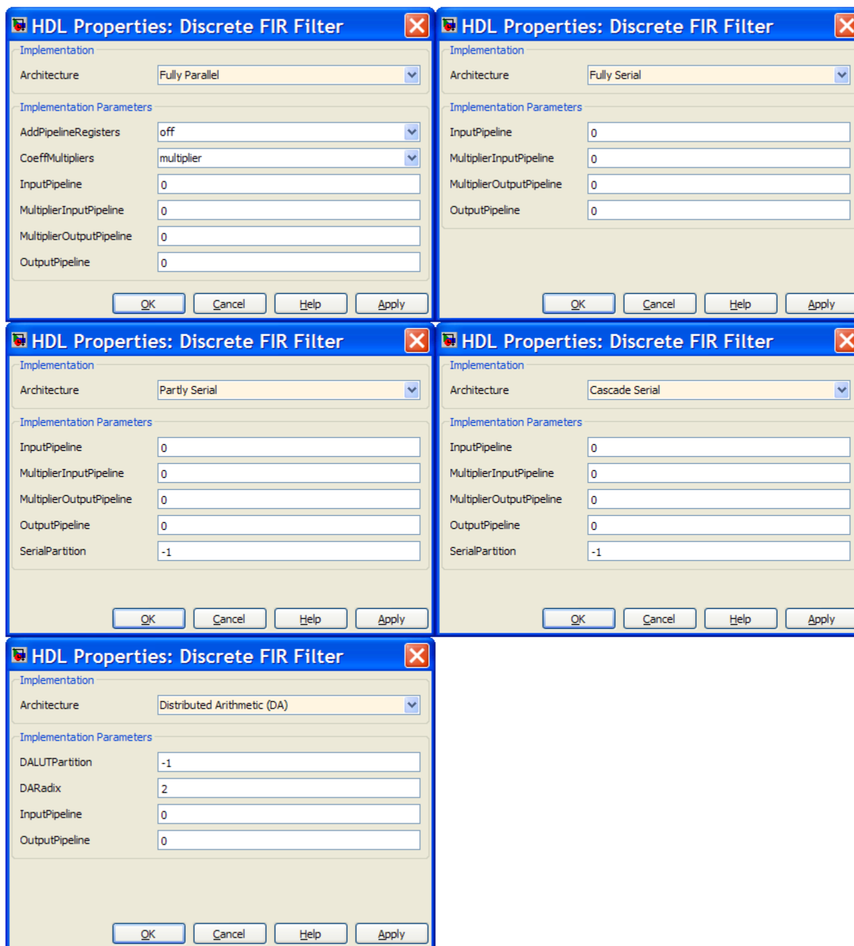
HDL Block Properties Interface Allows Choice of Filter Architecture

You can choose from several filter architectures for FIR Decimation and Discrete FIR Filter blocks. Choices are:

- Fully Parallel
- Distributed Architecture (DA)
- Fully Serial

- Party Serial
- Cascade Serial

The availability of architectures depends on the transfer function type and filter structure of filter blocks. For Partly Serial and DA, specify at least **SerialPartition** and **DALUTPartition**, respectively, so that these architectures are inferred. For example, if you select `Distributed Architecture (DA)`, make sure to also set **DALUTPartition**.



HDL Support for FIR Filters With Serial Architectures and Complex Inputs

HDL support for serial implementations of a FIR block with complex inputs.

HDL Support for External Reset Added for Proportional-Integral-Derivative (PID) and Discrete Time Integrator (DTI) Blocks

External reset support added for level mode.