Wireless HDL Toolbox™

User's Guide

MATLAB&SIMULINK®

MathWorks®

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

Revision History

September 2017	Online only	New for Version 1.0 (Release 2017b)	
March 2018	Online only	Revised for Version 1.1 (Release 2018a)	
September 2018	Online only	Revised for Version 1.2 (Release 2018b)	
March 2019	Online only	Revised for Version 1.3 (Release 2019a)	
September 2019	Online only	Revised for Version 1.4 (Release 2019b)	
March 2020	Online only	Revised for Version 2.0 (Release 2020a)	
September 2020	Online only	Revised for Version 2.1 (Release 2020b)	
March 2021	Online only	Revised for Version 2.2 (Release 2021a)	
September 2021	Online only	Revised for Version 2.3 (Release 2021b)	
March 2022	Online only	Revised for Version 2.4 (Release 2022a)	
September 2022	Online only	Revised for Version 2.5 (Release 2022b)	

Contents

Model Architecture

1

HDL Code Generation and Verification

2

Reference Page Examples

3

Featured Examples

4

<div style="text-align:right">

Reference Applications

</div>

5

Model Architecture

Streaming Sample Interface

In this section...
"What Is a Streaming Sample Interface?" on page 1-2
"How Does a Streaming Sample Interface Work?" on page 1-2
"Why Use a Streaming Sample Interface?" on page 1-2
"Sample Stream Conversion" on page 1-3
"Timing Diagram of Serial Sample Interface" on page 1-3
"Using the nextFrame Output Signal" on page 1-4

What Is a Streaming Sample Interface?

In hardware, processing an entire frame of data at one time has a high cost in memory and area. To save resources, serial processing is preferable in HDL designs. Wireless HDL Toolbox blocks operate on one sample at a time rather than a frame. The blocks accept and return data as a serial stream of samples and control signals. The control signals indicate the frame boundaries. The protocol mimics the characteristics of a real-world system, including inactive intervals between samples and frames.

How Does a Streaming Sample Interface Work?

The control protocol uses start and end signals to demark each frame, and a valid signal to indicate which samples to process. The Wireless HDL Toolbox streaming sample protocol allows you to configure the number of idle cycles between samples and between frames. Idle cycles model the bursty character of real-world systems.

This protocol allows for frames of different sizes, such as if runt or partial frames enter the system due to synchronization changes.

Why Use a Streaming Sample Interface?

Format Independence

The blocks that use this interface do not need a configuration option for an exact frame size or inactive intervals. In addition, if you change the input data timing for your design, you do not need to update each block. Instead, update the stream configuration once at the serialization step. Some blocks still require a maximum frame size parameter to allocate memory resources.

Error Tolerance

By using a streaming sample interface with control signals, each Wireless HDL Toolbox block starts computation on a fresh set of samples at the start-of-frame signal. Computations on the new frame occur whether or not the block receives the end signal for the previous frame.

The protocol tolerates minor timing errors. If the number of valid and invalid cycles between start and end signals varies, the blocks continue to operate correctly. This protocol makes the system resilient to runt frames and synchronization changes.

The Wireless HDL Toolbox encoder blocks require minimum between-frame spacing to accommodate insertion of codewords. The turbo and convolutional decoder blocks require that the previous frame

is decoded (has asserted the frame end signal) before the next frame arrives. The polar, LPDC, and RS encoder and decoder blocks provide a signal to indicate when the block is ready to receive the start of a new frame.

Sample Stream Conversion

Use the Frame To Samples block to convert framed data to a stream of samples and control signals that conform to this protocol. The control signals are grouped in a bus data type called `samplecontrol`.

The Frame To Samples block can serialize fixed-size frames. If your frames vary in size, use the `whdlFramesToSamples` function to convert framed data to vectors of samples and control signals in MATLAB®. Then import the vectors to Simulink®. Use the Sample Control Bus Creator block to create a `samplecontrol` bus in your model.

If your data is already in a serial format, design your own logic to generate these control signals from your existing serial control scheme.

Supported Sample Data Types

Wireless HDL Toolbox blocks have an input and output port, `sample`, for the streaming sample data. The blocks capture one sample at a time from the input, and produce one sample at a time for output. The samples can be one of these supported data types.

Port	Description	Data Type
sample	Scalar integer value that represents one sample. The protocol also allows for a vector of integer values that represent a single sample, such as for turbo-encoded samples.	Supported data types include: • Boolean • uint or int • ufix or sfix double and single are supported for simulation but not for HDL code generation.

Streaming Sample Control Signals

Wireless HDL Toolbox blocks have an input and output port, `ctrl`, for the frame control signals relating to each sample. These three control signals indicate the validity of a sample and the boundaries of the frame. The control signal port is a nonvirtual bus data type called `samplecontrol`. For details of the bus data type, see "Sample Control Bus" on page 1-7.

Timing Diagram of Serial Sample Interface

The timing diagram illustrates the streaming sample protocol. It shows a six-sample input frame and the equivalent sequence of control and data signals.
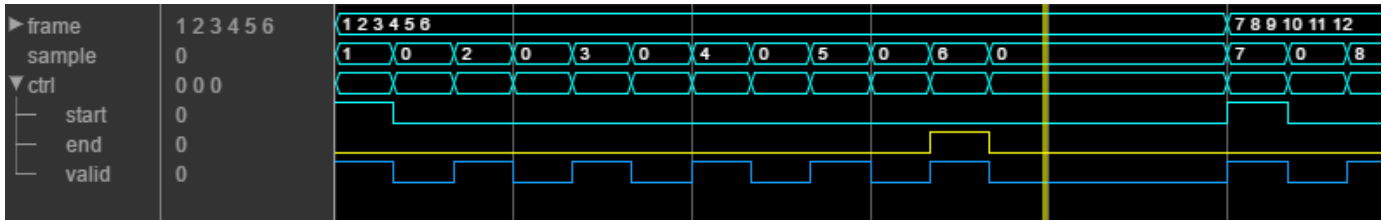
The input frame is ([1 2 3 4 5 6])', and the serializer is configured to insert idle cycles around the valid samples:

- One idle cycle between samples
- Three idle cycles between frames
- One value representing each sample (default output size)

You can specify these parameters by using either the Frame To Samples block or the whdlFramesToSamples function.

The control signals start and end are 1 for the first and last valid samples of the frame, respectively. The valid signal is 1 for each valid input sample. The valid signal is 0 for the idle cycles inserted between the samples and between the frames. The six-sample frame is now represented by streaming data over 15 cycles.

Using the nextFrame Output Signal

The NR Polar Encoder, NR Polar Decoder, NR LDPC Encoder, NR LDPC Decoder, and RS Decoder blocks each provide an output signal to indicate when the block is ready to receive the start of a new frame. This signal is necessary because these blocks cannot accept a new frame at certain stages of internal computations, and the latency of those stages can vary with the values of input ports.

Port	Description	Data Type
nextFrame	Boolean scalar that indicates when the block can accept the start of a new frame	Boolean

This waveform shows the NR Polar Encoder block processing several frames. The **nextFrame** output signal is 0 when the block is processing data, and 1 when the block is ready to receive the start of a new frame. The cursors show the latency varying with the values of the input **K** and **E** port values. For the first frame with given **K** and **E** values, the block must determine the message length and information bit mapping for those values. This configuration stage means the block needs some time before it is ready to accept the next input frame. For subsequent frames with the same values for **K** and **E**, the block is ready sooner because it does not need to recompute the configuration.
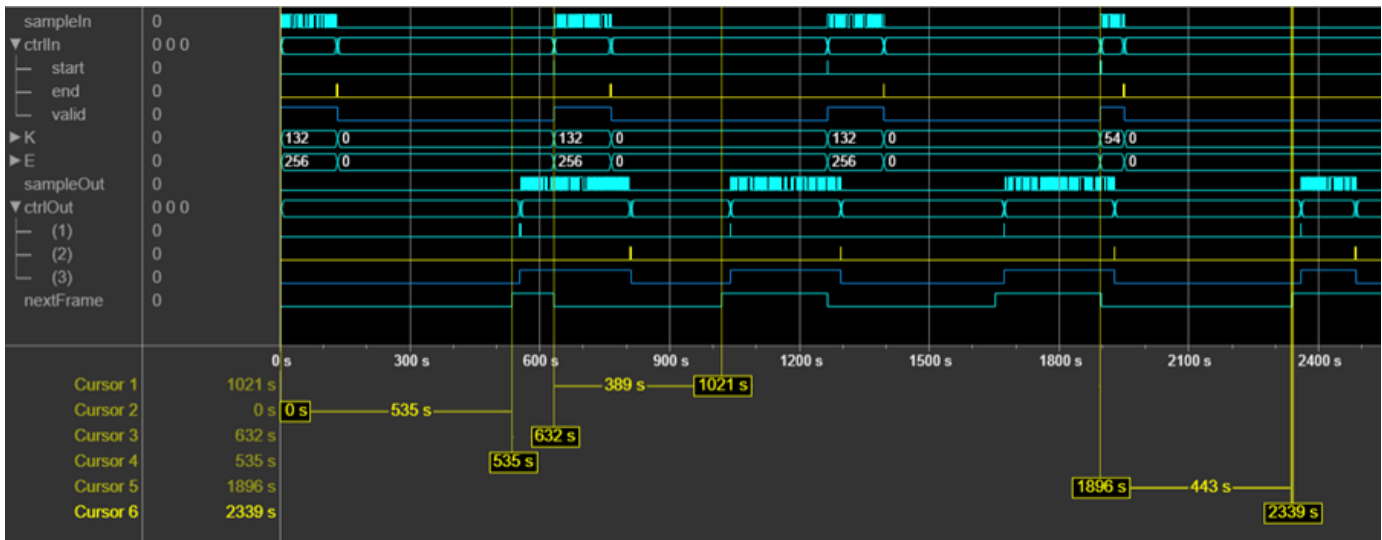
If the block receives an input **start** signal while **nextFrame** is 0, the block discards the frame in progress and begins processing the new data. This waveform shows an NR Polar Encoder input frame (3) applied when **nextFrame** is 0. The block discards the frame in progress (2) and processes the new frame (3) as normal.
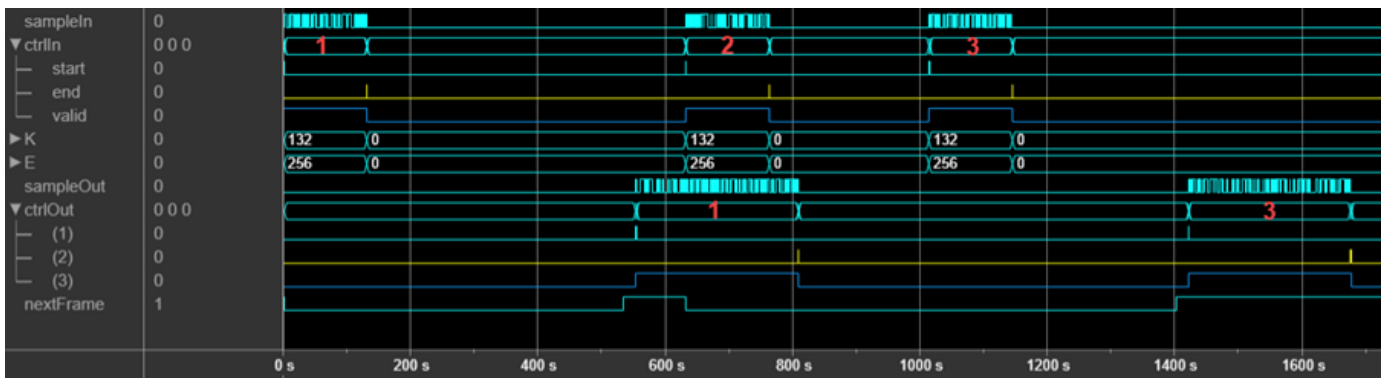
If the block receives an invalid input frame, for example, if the frame size is not within the supported range, then the block sets **nextFrame** to 1 one cycle after the input **end** signal. This behavior indicates that the input frame is discarded. This waveform shows an NR Polar Encoder input frame (1) that does not have the correct number of samples expected for the accompanying **K** and **E** values. The waveform shows the **nextFrame** signal set to 1 immediately after the input **end** signal from frame 1. The block discards the frame in progress (1) and processes the new frame (2) as normal.
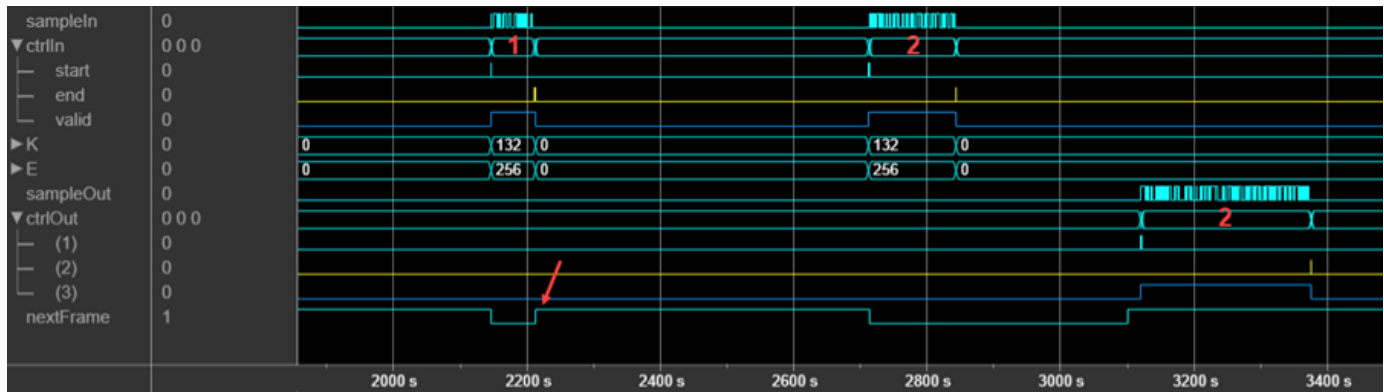
See Also

Blocks
Frame To Samples | Samples To Frame

Functions
whdlFramesToSamples | whdlSamplesToFrames

Related Examples

- "Verify Turbo Decoder with Streaming Data from MATLAB"
- "Verify Turbo Decoder with Framed Data from MATLAB"

Sample Control Bus

Wireless HDL Toolbox blocks use a nonvirtual bus data type, `samplecontrol`, for control signals associated with serial data. The bus contains three `boolean` signals indicating the validity of a sample and the boundaries of the frame. You can easily connect one block to another, because all Wireless HDL Toolbox blocks use this bus for input and output. To convert frames into a sample stream and a `samplecontrol` bus, use the Frame To Samples block. This block serializes fixed-size frames. If your frames vary in size, use the `whdlFramesToSamples` function to convert the frames to a data vector in MATLAB, and then import the data into Simulink.

Signal	Description	Data Type
start	`true` for the first sample in the frame	Boolean
end	`true` for the last sample in the frame	Boolean
valid	`true` for any valid sample	Boolean

Troubleshooting:

When you generate HDL code from a Simulink model that uses this bus, you may need to declare an instance of `samplecontrol` bus in the base workspace. If you encounter the error `Cannot resolve variable 'samplecontrol'` when you generate HDL code in Simulink, use the `samplecontrolbus` function to create an instance of the bus type. Then try generating HDL code again.

To avoid this issue, the Wireless HDL Toolbox model template includes this line in the `InitFcn` callback.

```
evalin('base','samplecontrolbus')
```

You can also call this command from the MATLAB command line.

See Also

Blocks
Frame To Samples | Samples To Frame

More About

- "Streaming Sample Interface" on page 1-2

Configure the Simulink Environment for Hardware Design

About Simulink Model Templates

Simulink model templates provide common configuration settings and best practices for new models. Instead of using the default canvas of a new model, select a template model to help you get started.

For more information on Simulink model templates, see "Build and Edit a Model Interactively".

Create Model Using Wireless HDL Toolbox Model Template

1   Click the Simulink button, , or type `simulink` at the MATLAB command prompt.
2   On the Simulink start page, find the Wireless HDL Toolbox section, and click the **Streaming Data from MATLAB** or **Framed Data from MATLAB** template.
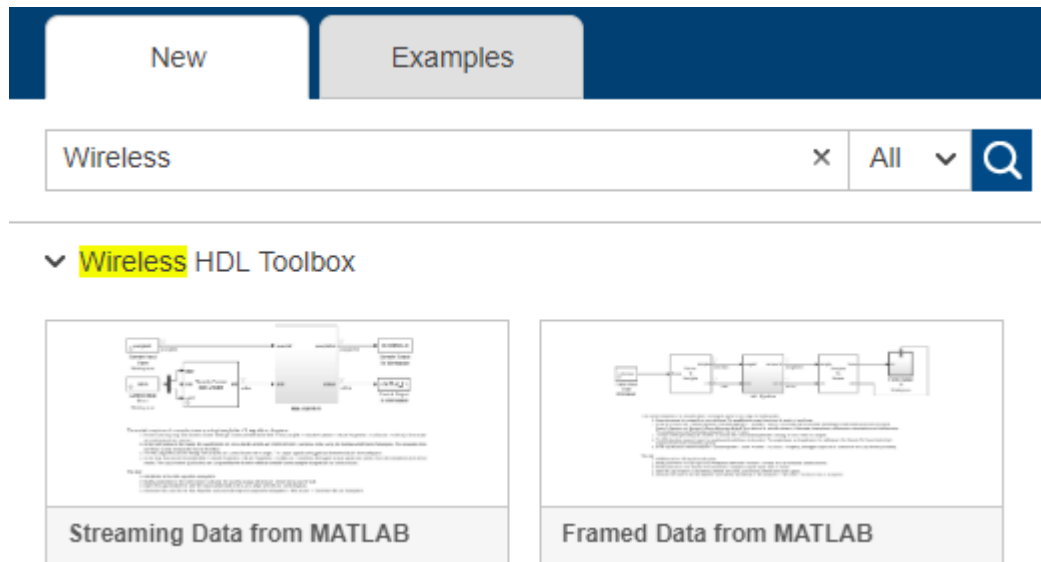
A new model, with the template contents and settings, opens in the Simulink Editor. Select **Save** to save the model.

Alternatively, you can create a new model from the template on the command line. For example:

```
new_system my_whdl_Fmodel FromTemplate whdl_framed_data.sltx
open_system my_whdl_Fmodel
```

Or:

```
new_system my_whdl_Smodel FromTemplate whdl_streaming_data.sltx
open_system my_whdl_Smodel
```

Wireless HDL Toolbox Model Templates

Both Wireless HDL Toolbox model templates include an empty subsystem, HDL Algorithm. This subsystem accepts and returns streaming data and accompanying control signals using the `samplecontrolbus`. You can design an HDL-targeted algorithm within this subsystem.

The templates also configure the model for HDL code generation. Both templates:

- Configure solver settings equivalent to calling `hdlsetup`
- Display data rates and data types in the Model Editor
- Create an instance of `samplecontrolbus` in the workspace (in `InitFcn`)

The simulation time, input data, and block parameters are defined in the callback function, `InitFcn`. To view or edit this function, on the **Modeling** tab, expand **Model Settings** and click **Model Properties**, and then on the **Callbacks** tab, click `InitFcn*`.

Framed Data Template

The **Framed Data from MATLAB** template imports framed data from the MATLAB workspace, assuming all frames are the same size. Then, it converts the data to a sample stream by using the Frame To Samples block.

The output of the HDL Algorithm subsystem is connected to a Samples To Frame block. This block converts the output back to framed data for export to the MATLAB workspace.

The `InitFcn` defines placeholder input frames and settings for the Frame Input From Workspace, Frame To Samples, and Samples To Frame blocks.

The `StopFcn` applies the valid signal to the output data and creates a single variable in the workspace.

The model has one data rate for the framed data and a faster data rate for the sample stream. You can display these rates as different colors in the Simulink model.
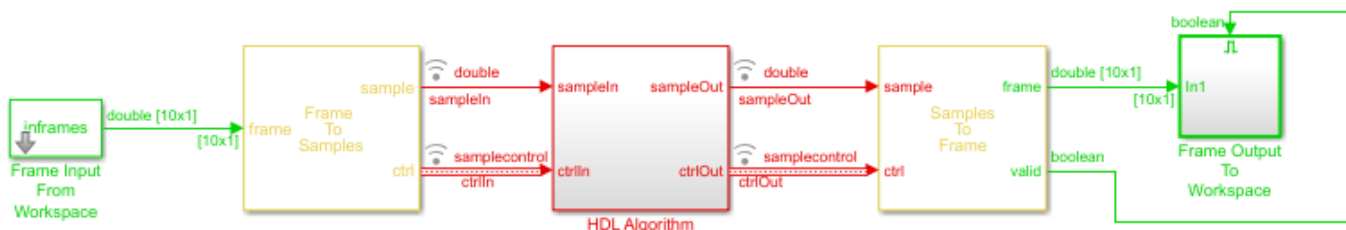
Streaming Data Template

Use the **Streaming Data from MATLAB** template when your data stream has different-sized frames. The `InitFcn` defines placeholder input frames and uses the `whdlFramesToSamples` function to convert framed data to vectors of data and control signals. The From Workspace block imports these variables to the model.

To connect to the HDL Algorithm subsystem and any Wireless HDL Toolbox blocks that you add inside it, the model converts the control signals to the `samplecontrolbus` type, using the Sample Control Bus Creator block.

The model exports the streaming data and control signals back to the MATLAB workspace. The `StopFcn` uses the `whdlSamplesToFrames` function to convert them back to framed data.

The model has a single data rate because all signals in the model represent streaming samples.
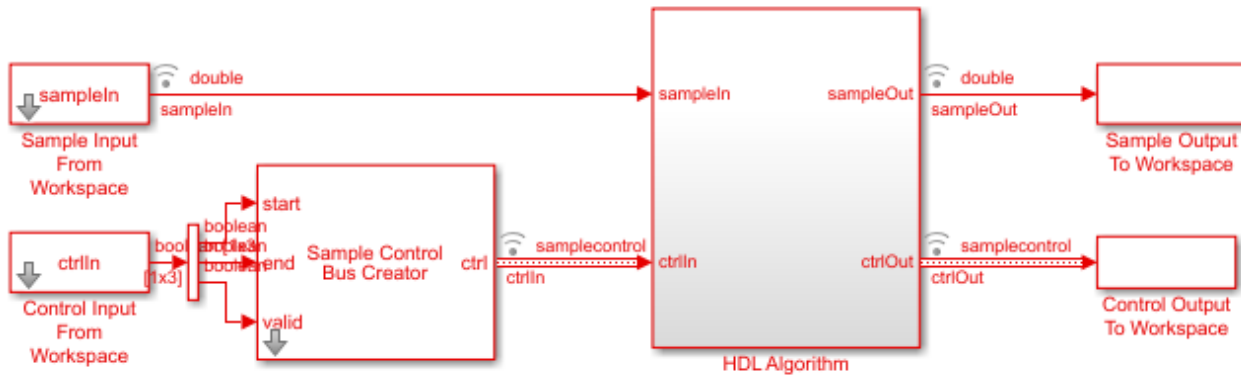
See Also

Blocks
Frame To Samples | Samples To Frame | Sample Control Bus Creator

Functions
whdlFramesToSamples | whdlSamplesToFrames

More About

- "Streaming Sample Interface" on page 1-2

HDL Code Generation and Verification

HDL Code Generation Support

You can use Simulink for rapid prototyping of hardware designs. Wireless HDL Toolbox blocks, when used with HDL Coder™, support HDL code generation. HDL Coder tools generate target-independent synthesizable Verilog® and VHDL® code for FPGA programming or ASIC prototyping and design.

HDL Code Generation Support in Wireless HDL Toolbox

Most blocks in Wireless HDL Toolbox support HDL code generation.

The following blocks are for simulation only and are not supported for HDL code generation:

- Frame To Samples
- Samples To Frame
- FIL Frame To Samples
- FIL Samples To Frame

Other Blocks Supporting HDL Code Generation

Other MathWorks® products also include blocks supported for HDL code generation that you can use to build up your design.

To create a library of HDL-supported blocks from all your installed products, enter `hdllib` at the MATLAB command line. This command requires an HDL Coder license.

You can also view blocks that are supported for HDL code generation in documentation by filtering the block reference list. Click **Blocks** in the blue bar at the top of the Help window, then select the **HDL code generation** check box at the bottom of the left column. The blocks are listed in their respective products. You can use the table of contents in the left column to navigate between products and categories.

Refer to the "Extended Capabilities > HDL Code Generation" section of each block page for block implementations, properties, and restrictions for HDL code generation.

Streaming Sample Interface in HDL

The streaming sample control bus data type used by Wireless HDL Toolbox blocks is flattened into separate signals in HDL.

In VHDL, the interface is declared as:

```
PORT( clk              :   IN    std_logic;
      reset            :   IN    std_logic;
      enb              :   IN    std_logic;
      in0              :   IN    std_logic_vector(7 DOWNTO 0); -- uint8
      in1_start        :   IN    std_logic;
      in1_end          :   IN    std_logic;
      in1_valid        :   IN    std_logic;
      out0             :   OUT   std_logic_vector(7 DOWNTO 0); -- uint8
      out1_start       :   OUT   std_logic;
      out1_end         :   OUT   std_logic;
      out1_valid       :   OUT   std_logic
      );
```

In Verilog, the interface is declared as:

```
input   clk;
input   reset;
input   enb;
input   [7:0] in0;  // uint8
input   in1_start;
input   in1_end;
input   in1_valid;
output  [7:0] out0;  // uint8
output  out1_start;
output  out1_end;
output  out1_valid;
```

See Also

More About

- "Streaming Sample Interface" on page 1-2
- "Generate HDL Code" on page 2-5

Generate HDL Code

You can generate HDL code from subsystems that include blocks supported for HDL code generation, such as the model in "Verify Turbo Decoder with Streaming Data from MATLAB". In that example, you can generate HDL code from the HDL Algorithm subsystem.

To generate HDL code, you must have an HDL Coder license.

Prepare Model

Run `hdlsetup` to configure the model for HDL code generation. If you started your design using the Wireless HDL Toolbox Simulink model template, your model is already configured for HDL code generation.

Generate HDL Code

Right-click the HDL Algorithm subsystem, and select **HDL Code** > **Generate HDL for Subsystem** to generate HDL using the default settings. The output log of this operation is shown in the MATLAB Command Window, along with the location of the generated files.

To change code generation options, use the **HDL Code Generation** panes of the Simulink Configuration Parameters dialog box. For guidance through the HDL code generation process, or to select a target device or synthesis tool, right-click the HDL Algorithm subsystem, and select **HDL Code** > **HDL Workflow Advisor**.

Alternatively, from the MATLAB Command Window, you can call:

```
makehdl([modelname '/HDL Algorithm'])
```

Generate HDL Test Bench

You can select options to generate a test bench in the Simulink Configuration Parameters dialog box or in the HDL Workflow Advisor.

Alternatively, to generate an HDL test bench from the command line, call:

```
makehdltb([modelname '/HDL Algorithm'])
```

See Also

Functions
makehdl | makehdltb

Related Examples

- "HDL Code Generation and FPGA Synthesis from Simulink Model" (HDL Coder)
- "Choose a Test Bench for Generated HDL Code" (HDL Coder)

FPGA-in-the-Loop

FPGA-in-the-loop (FIL) enables you to run a Simulink simulation that is synchronized with an HDL design running on an Intel® or Xilinx® FPGA board. This link between the simulator and the board enables you to verify HDL implementations directly against Simulink or MATLAB algorithms. You can apply real-world data and test scenarios from these algorithms to the HDL design on the FPGA.

When simulating Wireless HDL Toolbox blocks, you must use a streaming sample interface. Streaming sample data, while required for hardware implementations of communications systems, is time-consuming at the FPGA-in-the-loop interface with Simulink.

You can convert from frames to samples and samples to frames either in Simulink or in MATLAB. Depending on your workflow, you can optimize your FPGA-in-the-loop simulation in one of two ways.

One workflow is a Simulink model that imports framed data from MATLAB. This type of model then uses the Frame To Samples and Samples To Frame blocks to convert the data format. For FPGA-in-the-loop, replace these conversion blocks with FIL Frame To Samples and FIL Samples To Frame blocks.

The other workflow is a Simulink model that imports streaming data from MATLAB. This type of model goes with a MATLAB script that uses the `whdlFrameToSamples` and `whdlSamplesToFrames` functions. For FPGA-in-the-loop, modify your script and Simulink model so that they pass vectors of data to the FPGA-in-the-loop interface.

When you generate a programming file for a FIL target in Simulink, the tool creates a model to compare the FIL simulation with your Simulink design. For Wireless HDL Toolbox designs, the FIL block in that model replicates the sample-streaming interface and sends one sample at a time to the FPGA. Both these modifications construct vectors that make more efficient use of the interface between the Simulink model and the FPGA board.

The instructions that follow show how to modify FPGA-in-the-loop models for the "Verify Turbo Decoder with Streaming Data from MATLAB" and "Verify Turbo Decoder with Framed Data from MATLAB" workflow examples.

FIL Workflow: Framed Data from MATLAB

Autogenerated FIL Model

The generated model, including the FIL block that interfaces with the FPGA board, is shown for a model that converts to streaming samples in Simulink. If each sample is represented by multiple values, then the values are flattened into separate ports for FIL.

The blue ToFILSrc subsystem branches the sample-stream input of the HDL Algorithm block to the FromFILSrc subsystem. The blue ToFILSink subsystem branches the sample-stream output of the HDL Algorithm block into the Compare subsystem, where it is compared with the output of the HDL Algorithm_fil block. This setup is slow because the model sends only a single sample, and its associated control signals, in each packet to and from the FPGA board.

Modified FIL Model

To improve the communication bandwidth with the FPGA board, modify the autogenerated model. The modified model uses the FIL Frame To Samples and FIL Samples To Frame blocks to send one frame at a time.
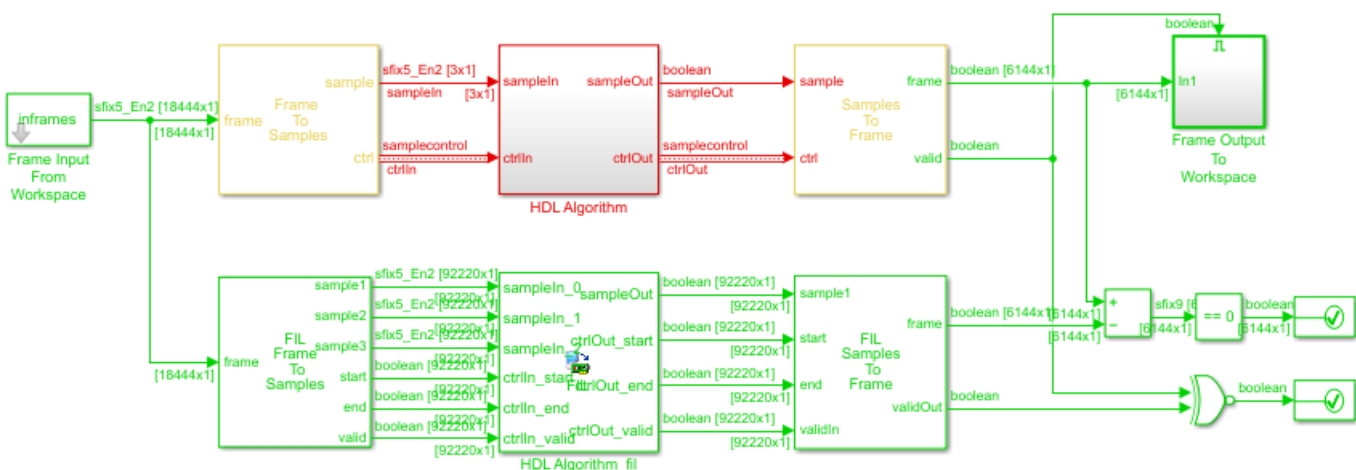
To create this modified FIL model:

1   Remove the blue subsystems, and create a branch at the **frame** input port of the Frame To Samples block.

2 Insert the FIL Frame To Samples block before the HDL Algorithm_fil block. Insert the FIL Samples To Frame block after the HDL Algorithm_fil block.

3 Set the **Output frame size** on the FIL block to the input frame size.

Runtime Options

Overclocking factor:	1
Output frame size:	inframesize

4 In the FIL Frame To Samples and FIL Samples To Frame blocks, set the parameters to match the settings of the Frame To Samples and Samples To Frame blocks.

5 Branch the frame output of the Samples To Frame block for comparison. You can compare the entire frame at once with a Diff block. Compare the `validOut` signals using an XOR block.

The input size at the FIL block is the frame size from the input data frames. The vector size of the FIL block ports does not modify the generated HDL code. It affects only the packet size of the communication between the simulator and the FPGA board. This modified model sends an entire frame to the FPGA board in each packet, significantly improving the efficiency of the communication link.

FIL Workflow: Streaming Data from MATLAB

Autogenerated FIL Model

The generated model, including the FIL block that interfaces with the FPGA board, is shown for a model that converts to streaming samples in MATLAB. If each sample is represented by multiple values, then the values are flattened into separate ports for FIL.

The blue ToFILSrc subsystem branches the sample-stream input of the HDL Algorithm block to the FromFILSrc subsystem. The blue ToFILSink subsystem branches the sample-stream output of the HDL Algorithm block into the Compare subsystem, where it is compared with the output of the HDL Algorithm_fil block. This setup is slow because the model sends only a single sample, and its associated control signals, in each packet to and from the FPGA board.

Modified FIL Model

To improve the communication bandwidth with the FPGA board, use the generated FIL block in a different model. The alternate model imports and exports vectors of flattened data. The accompanying MATLAB script reshapes the input and output data, and verifies the FIL output against a behavioral model. Reshaping the data in MATLAB is easier and the simulation is faster than reshaping in Simulink.
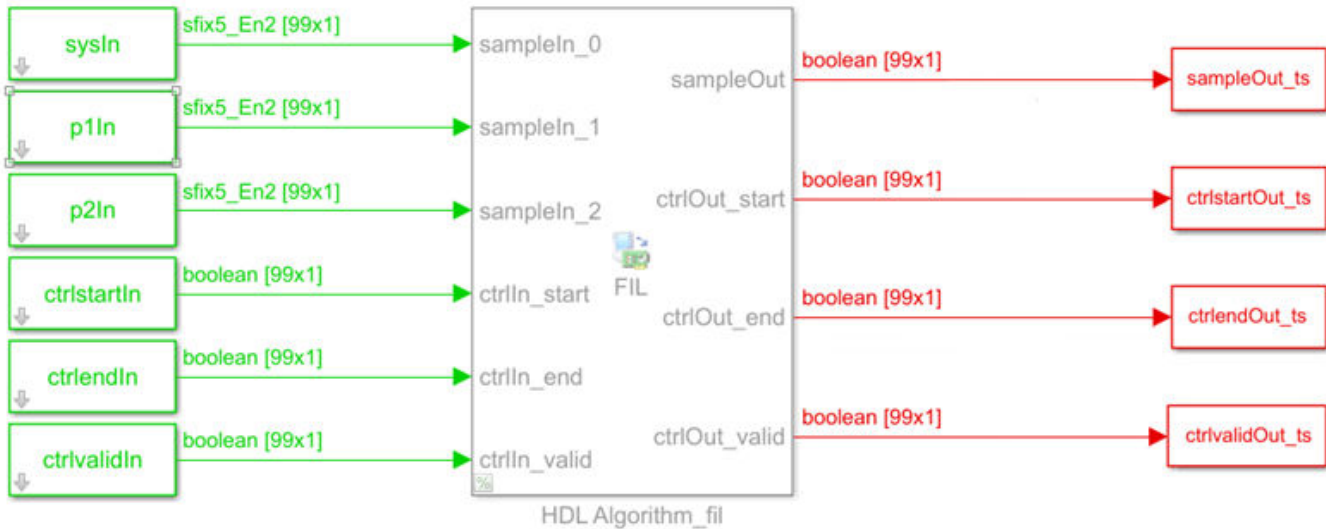
First, modify the accompanying MATLAB script:

1 Pick a frame size for the FIL simulation. This size does not have to match the actual frame sizes in the generated data. It can contain your entire data set. The FIL block divides the data into maximum size packets for communication with the FPGA board.

```
filframesize = 99;
```

2 Combine the cell array of input frames into one matrix.

```
allframes = [inframes{:}];
```

3 Flatten the samples and control signals so there is one vector for each input port on the FIL block. This model includes the LTE Turbo Decoder block, so the input samples consist of three values.

```
sysIn = allframes(1:3:end);
p1In  = allframes(2:3:end);
p2In  = allframes(3:3:end);

ctrlstartIn = ctrlIn(1:3:end);
ctrlendIn   = ctrlIn(2:3:end);
ctrlvalidIn = ctrlIn(3:3:end);
```

4 Call the FIL model.

```
simTime = size(allframes,1);
modelname = 'TurboDecoderStreamingFILVectortoSL';
open_system(modelname);
sim(modelname);
```

5 Reshape the output variables for input to the whdlSamplesToFrames function. Recreate an *N*-by-3 control signal matrix and a vector of sample data. In this example, the output sample is a single value. If the output sample is multiple values, build an *N*-by-*SampleSize* sample matrix.

```
sampleOut = squeeze(sampleOut_ts.Data);
ctrlOut = [squeeze(ctrlstartOut_ts.Data) ...
```

```
        squeeze(ctrlendOut_ts.Data) ...
        squeeze(ctrlvalidOut_ts.Data)];
```

Then, create a Simulink model:

1 Copy the generated FIL block into a new model.

2 Configure and connect a Signal From Workspace block for each input port on the FIL block. Use the variables from your MATLAB script as the parameter values.

3 Set the **Output frame size** on the FIL block to the desired FIL frame size.

4 Configure and connect a To Workspace block for each output port of the FIL block.

The input size at the FIL block is the frame size you specify on the Signal To Workspace blocks. The vector size of the FIL block ports does not modify the generated HDL code. It affects only the packet size of the communication between the simulator and the FPGA board. This modified model sends an entire frame to the FPGA board in each packet, significantly improving the efficiency of the communication link.

See Also

More About

- "Verify Turbo Decoder with Streaming Data from MATLAB"
- "Verify Turbo Decoder with Framed Data from MATLAB"

Verify Viterbi Decoder Using HDL Cosimulation

This example shows how to generate and verify HDL code to implement a fixed-point Viterbi decoder.

To run this example, in addition to the required MATLAB® products, you must install and include on the MATLAB system path either Mentor Graphics® ModelSim®/Questasim® or Cadence® Xcelium®.

Overview of Simulink Model

Open the Simulink® model `viterbi_codegen.slx`. This model generates HDL code for a fixed-point Viterbi decoder.
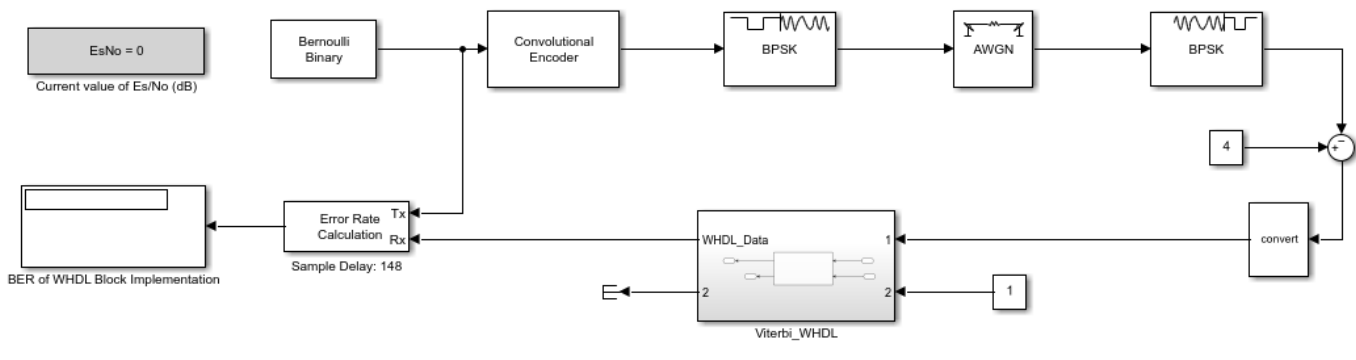
Copyright 2011-2021 The MathWorks, Inc.

The model uses binary phase-shift keying (BPSK) and additive white Gaussian noise (AWGN) blocks to simulate the wireless transmission of data. In the top model, the parameter **EsNo**, which represents the average signal energy to noise ratio, affects the transmission of data. By default, the **EsNo** parameter is set to 0.

After you initiate the data transmission, the test bench feeds the data into the Viterbi Decoder block, which is implemented using the Wireless HDL Toolbox™ product. The Viterbi Decoder block attempts to recover the original data but might have errors in the recovery. To measure how accurate this decoder is, the test bench sends the decoded data to an Error Rate Calculation block along with the original data. Then the Display block displays the results from this calculation.

Generate HDL Code

To open the **HDL Coder**(TM) app, on the **Apps** tab in the Simulink Toolstrip, click the **HDL Coder** app icon. To select the toolchain you want to use for your cosimulation, first click **Settings** to open the Configurations Parameters dialog box. In the left pane, click **HDL Code Generation**, then **Test Bench**. For the **Simulation tool** parameter, select the toolchain. Apply the changes by clicking **OK**.

To generate HDL code for the Viterbi decoder and open a new Simulink model, click **Generate Testbench**, then **HDL Cosimulation**.

Launch HDL Simulator

You can connect and format the new Simulink model to accommodate your test bench. This example includes two prepared models: `viterbi_modelsim.slx` and `viterbi_xcelium.slx`. Choose the

model that fits your toolchain. This example uses the ModelSim/QuestaSim Simulink model, which is shown in the figure.
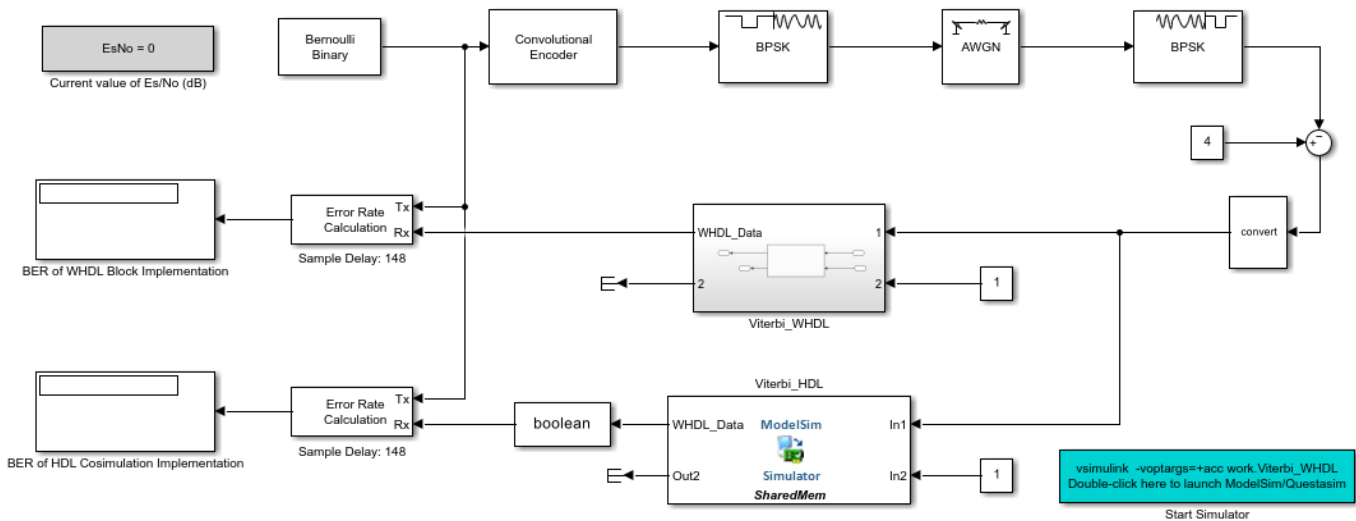
Copyright 2011-2021 The MathWorks, Inc.

To launch the HDL simulator, double-click the Start Simulator block in the model. In addition to launching the HDL simulator, this action inputs the commands to compile the HDL code and prepares for cosimulation with MATLAB and Simulink.

Run Simulation

When the HDL simulator finishes compiling the HDL files and preparing for simulation, the text `Ready for cosimulation ...` appears in the HDL simulator command window. After this text appears, return to the open model in Simulink and run the simulation from there.

When the simulation finishes, the Simulink model displays the results. In this example, the results are displayed as the bit error rate (BER) shown in the two Display blocks. The two displays show the BER results from the Viterbi Decoder block from the Wireless HDL Toolbox Product and the HDL coded block implemented using HDL Coder. Based on the results, the HDL Coder implementation yields the same results as the original block.

Rerun Simulation with New Parameters

The parameter **EsNo** controls the behavior of the transmission. Change this parameter to change the simulation behavior. For example, enter this command at the MATLAB command prompt.

```
EsNo = 5;
```

Changing this parameter does not require new HDL code to be generated, as this change does not affect the Viterbi block. To repeat this example with the new parameter value, run the simulation again from the open Simulink model.

Finish Simulation

After you are finished with simulation, close the HDL simulator session. Then, return to Simulink and close the model.

See Also

Functions

makehdl (HDL Coder) | makehdltb (HDL Coder)

Blocks

Viterbi Decoder

Related Topics

- "Set Up for HDL Cosimulation" (HDL Verifier)
- "Run MATLAB-HDL Cosimulation" (HDL Verifier)
- "Generate HDL Code" on page 2-5
- "Choose a Test Bench for Generated HDL Code" (HDL Coder)

Verify 5G Wireless Applications Using SystemVerilog DPI

This example shows how to use SystemVerilog DPI components to verify 5G wireless applications in an HDL environment.

The system in this example uses various 5G components and a parameterizable 5G waveform generator to validate the behavior of the Synchronization Signal Block (SSB) decoding section of the Master Information Block (MIB) recovery process.

The verification workflow includes these key benefits:

- 3GPP 5G New Radio (NR) standard requires deep domain expertise. Creating a standard-compliant waveform verification model can be challenging. Generating a DPI component from a Wireless HDL Toolbox™ waveform generator simplifies the test bench design process by automatically creating a standard compliant verification IP.
- The parameterizable 5G waveform generator tests the DUT in different scenarios. You can reconfigure the parameters to create a series of test cases to meet coverage.
- The standalone 5G DPI components generated from Simulink® and MATLAB® can be reused and integrated in customized test benches.
- The component-based workflow makes designing a standalone test bench faster. You can splice different modules in the top-level test bench to test different 5G function components.
- Full functional control at the top level test bench enables component manipulation according to the process status changes. This control results in performance gains compared to a test-vector-based HDL test bench.

MIB Recovery Process

MIB recovery requires SSB detection, demodulation, and decoding. This example shows how to validate the HDL code generated by HDL Coder™ for the SSB decoding module.
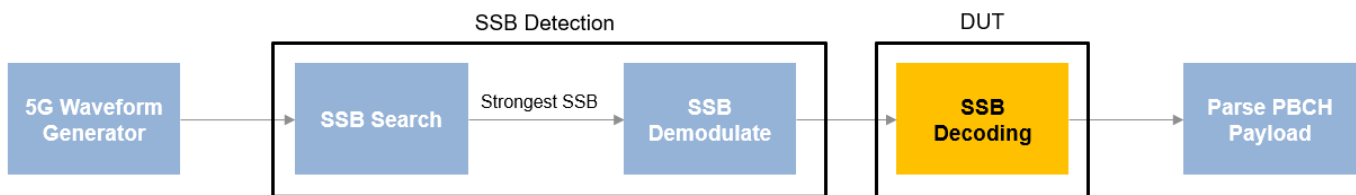
SSB detection performs a primary synchronization sequence (PSS) search, orthogonal frequency division multiplexing (OFDM) demodulation, and a secondary synchronization sequence (SSS) search. SSB detection has two modes of operation: search and demodulation. In search mode, the detection searches for SSBs and returns their parameters. In demodulation mode, the detection recovers a specified SSB, OFDM-demodulates its resource grid, and searches for the SSS within the appropriate resource elements. The details of SSB detection and demodulation are described in the "NR HDL Cell Search" on page 5-88 example.

SSB decoding performs a demodulation reference signal (DMRS) search, channel estimation and phase equalization, and broadcast channel (BCH) decoding steps. The details of SSB decoding are described in the "NR HDL MIB Recovery" on page 5-56 example.

5G waveform generator uses 5G Toolbox™ functions to generate a test waveform, which is then applied to the SSB detection in search mode. After the strongest SSB is determined, the test

waveform is applied to the SSB detection in demodulation mode to recover a specified SSB resource grid and search for the SSS within the appropriate resource element.

After an SSB is detected and demodulated, it needs to be decoded to extract the MIB content. When SSB decoding has the demodulated grid, the SSB decoding module decodes the SSB and output the PBCH payload, which is then parsed to extract the MIB data.

File Structure

This example uses these files.

Simulink models

- `nrhdlSSBDetection.slx`: This Simulink model uses the `nrhdlSSBDetectionFR1Core` model reference to simulate the behavior of the SSB detection part of the MIB recovery process.
- `nrhdlSSBDetectionFR1Core.slx`: This model reference implements the SSB detection algorithm.
- `nrhdlSSBDecoding.slx`: This Simulink model uses the `nrhdlSSBDecodingCore` model reference to simulate the behavior of the SSB decoding part of the MIB recovery process.
- `nrhdlSSBDecodingCore.slx`: This model reference implements the SSB decoding algorithm.

Simulink data dictionary

- `nrhdlReceiverData.sldd`: This Simulink data dictionary contains bus objects that define the buses contained in the example models.

MATLAB code

- `generate5GWaveform.m`: This function is a modified version of the Wireless HDL Toolbox™ 5G waveform generator, which is C code generation compatible.
- `runSSBDetectionModelSearch.m`: This script executes and verifies the `nrhdlSSBDetection` model in search mode.
- `runSSBDecodingModel.m`: This script uses the MATLAB reference to implement the cell search algorithm and then runs the `nrhdlSSBDecoding` Simulink model. The script verifies the operation of the model using 5G Toolbox and the MATLAB reference code.
- `nrsvdpiexamples`: This package contains the MATLAB reference code and utility functions for verifying the implementation models.

Pregenerated HDL test bench components (available for Windows® only)

- `5GNRCellDecodeDPITB`: This folder contains generated DPI components, HDL code of the decoding module, and the top-level test bench with associated build and simulation scripts.

Set Up for HDL Simulation

This section describes the workflow for generating the DPI component for each 5G functional component and HDL code for the SSB decoding component. The provided top-level test bench instantiates all of the generated components to validate the behavior of the HDL code that is generated from the SSB decoding block. To enable reuse of individual SystemVerilog DPI components and use a subset of the components in a test bench, generate SystemVerilog DPI for each 5G component individually.

The `5GNRCellDecodeDPITB` folder contains all of the necessary generated components. if you do not want to regenerate these components, skip this section.

5G Waveform Generator

This function uses 5G Toolbox functions to generate a test waveform. The 5G waveform generator has three input arguments: `ncellid`, `SNR`, and `frequencyOffset`. When you use this waveform generator in a SystemVerilog test bench, you can test different scenarios by providing different values for `SNR`, `frequencyOffset`, and `ncellid` without changing the component code. For this example, use this command to generate the DPI component from the MATLAB function `generate5GWaveform`.

```
dpigen generate5GWaveform -args {0,0,0} -PortsDataType LogicVector
```

The `-args {0,0,0}` parameter indicates that three scalar inputs of type `double`. The `-PortsDataType LogicVector` parameter indicates generating a logic vector type interface for the port.

SSB Detection

This component is introduced in the `nrhdlSSBDetection` Simulink model and the top-level test bench uses this component for the SSB search and SSB demodulation. Use these commands to run a search mode simulation and verify the results in MATLAB.

```
clear all;
runSSBDetectionModelSearch;
```

Then use this command to generate a DPI component for the Simulink subsystem `nrhdlSSBDetection/SSB Detection`.

```
rtwbuild('nrhdlSSBDetection/SSB Detection');
```

Choose Strongest PSS

This component is introduced in the `nrhdlSSBDetection` Simulink model and the top-level test bench uses this component to determine the strongest PSS from the PSSs detected by the SSB search. Use this command to generate a DPI component for the Simulink subsystem `nrhdlSSBDetection/chooseStrongestPSS`.

```
rtwbuild('nrhdlSSBDetection/chooseStrongestPSS');
```

SSB Decoding

This component is introduced in the `nrhdlSSBDecoding` Simulink model and is the DUT in this example. Use these commands to run an SSB decoding simulation in MATLAB.

```
clear all;
runSSBDecodingModel;
```

Then, use this command to generate HDL code from this component.
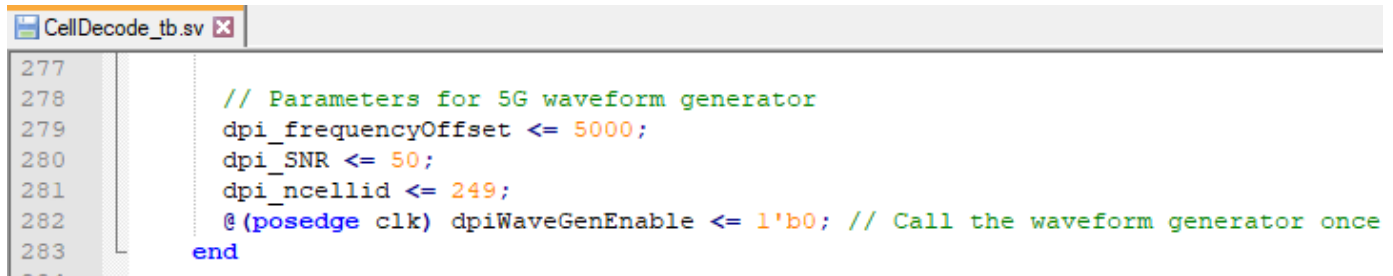
```
makehdl('nrhdlSSBDecoding/SSB Decoding','TargetLanguage','Verilog');
```

Parse PBCH Payload

This component is introduced in the `nrhdlSSBDecoding` Simulink model and the top-level test bench uses this component to parse the PBCH payload to obtain the MIB information. Use this command to generate the DPI component for the Simulink subsystem `nrhdlSSBDecoding/parsePBCHPayload`.

```
rtwbuild('nrhdlSSBDecoding/parsePBCHPayload');
```

The 5GNRCellDecodeDPITB folder contains a top-level test bench, CellDecode_tb.sv, to simulate the entire process described in the MIB Recovery Process section. In this example, the ncellid parameter is set to 249, the SNR parameter is set to 50, and the frequencyOffset parameter is set to 5000. You can modify the values of these parameters to test the design in different scenarios.

```
CellDecode_tb.sv ⊠
277
278         // Parameters for 5G waveform generator
279         dpi_frequencyOffset <= 5000;
280         dpi_SNR <= 50;
281         dpi_ncellid <= 249;
282         @(posedge clk) dpiWaveGenEnable <= 1'b0; // Call the waveform generator once
283     end
```

Run Test Bench

Add the QuestaSim simulator to the MATLAB system path, and then navigate to the 5GNRCellDecodeDPITB folder. To compile and simulate the DUT in QuestaSim, enter these commands at the MATLAB prompt.

```
!vsim < compile_dut.do
!vsim < sim_5G_waveform.do
```

Observe the following simulation results:

- In this figure of a waveform, signal names that start with "mib" carry MIB information from the waveform generator, and signal names that start with "decode" carry MIB information from the decode process. The waveform shows that the decoded MIB information matches the MIB information from the waveform generator.
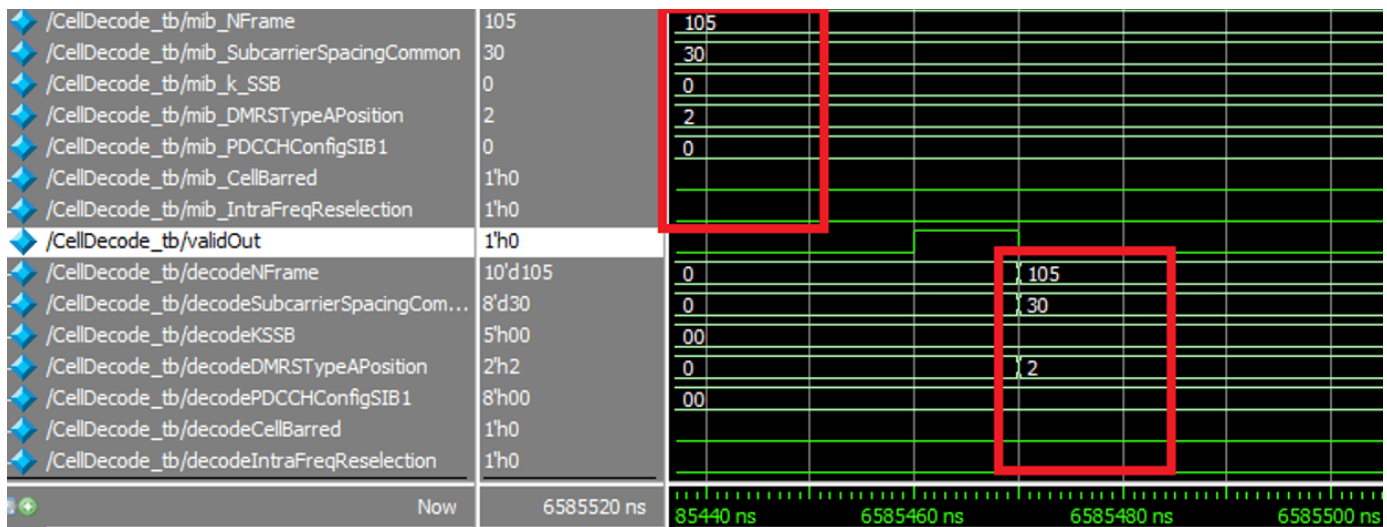
Using this approach, you can dynamically react to the status of each 5G functional component to save simulation time compared to a vector-based test bench.

- In this figure of a waveform, the value of the detection status changes from 6 to 8, indicating that the demodulation operation is complete. The SSS is found, and a demodulated resource grid is returned. In this case, you can start the SSB decoding process instead of waiting for SSB demodulation to finish processing the input vectors if you are using a vector-based HDL test bench.

/CellDecode_tb/status	4'h8	6	8
/CellDecode_tb/decodeStart	1'h1		
/CellDecode_tb/detectionReportValid	1'h0		
/CellDecode_tb/searchFrequencyOffset	32'h00000000	00000000	
/CellDecode_tb/searchTimingOffset	21'h000000	000000	
/CellDecode_tb/searchNCellID2	2'h0	0	
Now	6585520 ns	6520000 ns	6525000 ns
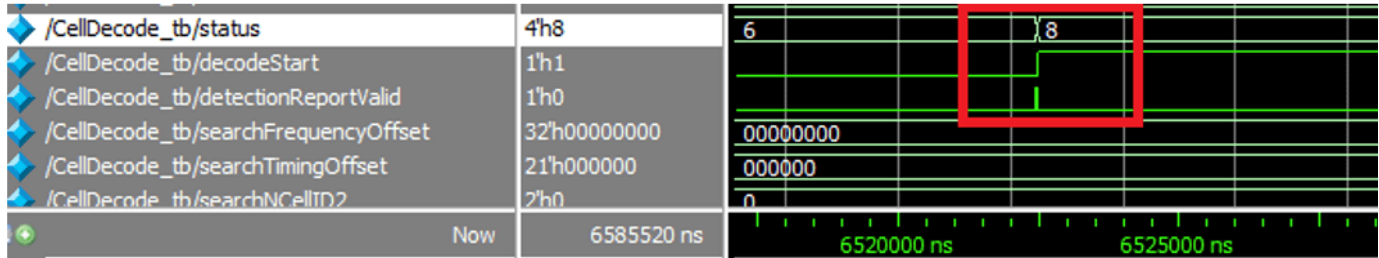
- In this figure of a waveform, the value of the decoding status changes from 2 to 4, indicating that the MIB is detected. In this case, you can stop the simulation rather than finish processing the grid resource data. In contrast, the vector-based test bench approach requires simulation for a fixed amount of time before analyzing the results.

/CellDecode_tb/bchStatus	3'h4	2			4		
/CellDecode_tb/ssbIndex3Lsb	3'h3	3					
/CellDecode_tb/pbchPayload	32'h0110004a	00a20...	14c00004	0d000...	40800...	04040...
/CellDecode_tb/validOut	1'h0					
/CellDecode_tb/nextSSB	1'h1					
/CellDecode_tb/decodeNFrame	10'h069	000			069	
/CellDecode_tb/decodeSubcarrierSpacingCom...	8'h1e	00			1e	
/CellDecode_tb/decodeKSSB	5'h00	00				
Now	6585520 ns	85440 ns	6585460 ns	6585480 ns

Reuse DPI Component

You can reuse the generated 5G function DPI components in a customized test bench.

- Use these components to test a subset of the MIB recover process. For example, reuse the 5G waveform generator to validate the behavior of an SSB search module.
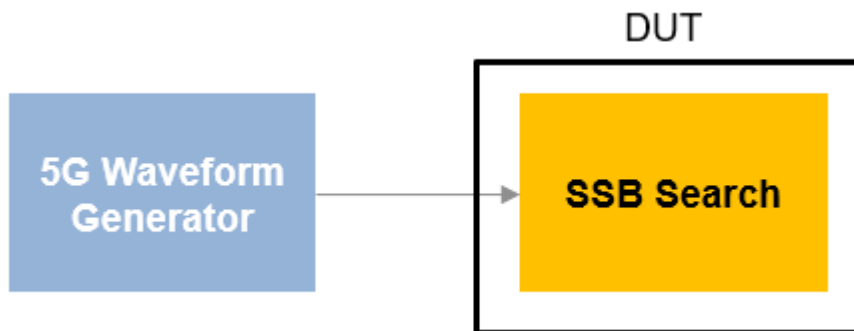
- Use these components to test different 5G function components in the MIB recovery process. For example, when you are validating the behavior of SSB demodulation in the MIB recovery process, you can reuse the 5G waveform generator, SSB search and parse PBCH payload DPI components,

and generate a DPI component from SSB decoding. Instantiate these components in your top-level test bench to validate the behavior of the SSB demodulate module.
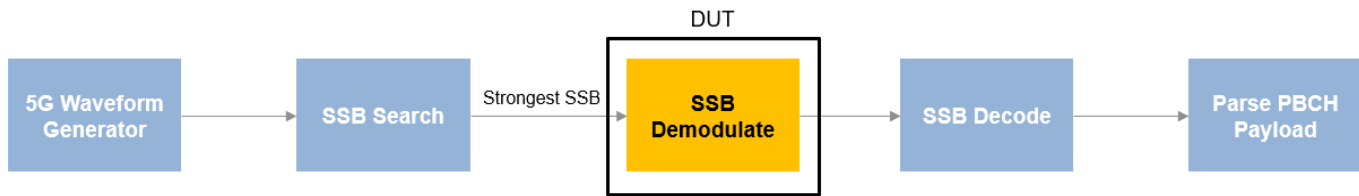
Conclusion

This example shows how to use a standalone test bench with DPI components to validate the SSB decoding module of an MIB recovery process. The HDL Verifier™ generated DPI components support tunable parameters, which enable customization of the 5G test waveform from the top-level test bench. You can reuse each generated DPI component in other custom HDL test benches. You can use this workflow for verifying the HDL IP in your wireless application.

Related Topics

- "NR HDL Cell Search" on page 5-88
- "NR HDL MIB Recovery" on page 5-56

Prototype Wireless Communications Algorithms on Hardware

Hardware support packages are Add Ons that connect MATLAB and Simulink simulations to hardware. Add Ons such as Communications Toolbox™ Support Package for Xilinx Zynq-Based Radio and SoC Blockset™ Support Package for Xilinx Devices enable you to design, prototype, and verify your designs on hardware. With these support packages, you can:

- Use the Xilinx Zynq-based radio as an I/O peripheral to transmit and receive real-time arbitrary waveforms.
- Transmit and receive RF signals out of the box and configure RF radio settings easily, enabling quick testing of SDR designs under real-world conditions.
- Model memory interfaces, ADC and DAC interfaces, and communication between the FPGA and processor on an SoC device.
- Customize and prototype wireless communications algorithms. Target only the FPGA fabric of the device, or deploy partitioned hardware-software co-design implementations across the ARM® processor and the FPGA fabric of the device.
- Run application examples to get started.

The workflow to target the FPGA on your Zynq board uses generated HDL code from HDL Coder and the HDL Coder Support Package for Xilinx Zynq Platform or HDL Coder Support Package for Xilinx RFSoC Devices. If you also target the ARM processor for a hardware-software codesign, the workflow additionally uses Simulink Coder™, Embedded Coder®, and Embedded Coder Support Package for Xilinx Zynq Platform.

For examples of how to deploy the Wireless HDL Toolbox reference applications to hardware devices, see:

- "Deploy NR HDL Reference Applications on FPGAs and SoCs" on page 5-105
- "Deploy LTE HDL Reference Applications on FPGAs and SoCs" on page 5-168
- "Deploy Satellite Communications HDL Reference Applications on FPGAs and SoCs" on page 5-313
- "Deploy WLAN HDL Reference Applications on FPGAs and SoCs" on page 5-241
- "Deploy Custom Communication Systems on FPGAs and SoCs" on page 5-216

How to Install Support Packages

A support package is an add-on that enables you to use a MathWorks product with specific third-party hardware and software. Support packages use the license of the base product. For instance, Communications Toolbox Support Package for Xilinx Zynq-Based Radio requires a license for Communications Toolbox.

Install support packages using the MATLAB **Add-Ons** menu. You can also use the **Add-Ons** menu to update installed support package software or update the firmware on third-party hardware.

To install support packages, on the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. You can filter this list by selecting categories (such as hardware vendor or application area), or by performing a keyword search.

Search the **Add-Ons** list for Zynq, and find these support packages:

- Communications Toolbox Support Package for Xilinx Zynq-Based Radio
- SoC Blockset Support Package for Xilinx Devices
- HDL Coder Support Package for Xilinx RFSoC Devices
- HDL Coder Support Package for Xilinx Zynq Platform
- Embedded Coder Support Package for Xilinx Zynq Platform (needed only for hardware-software co-design)

When the support package installation is complete, you must set up the host computer and radio hardware. For Windows® systems, the installer provides guided setup steps. For Linux® systems, the installer links to manual setup instructions.

Design Requirements for Using Communications Toolbox Support Package for Xilinx Zynq-Based Radio

The Communications Toolbox Support Package for Xilinx Zynq-Based Radio provides a reference design that you can use to create an IP core that integrates into the radio hardware. Use the HDL Workflow Advisor to guide you through generating a shareable and reusable IP core module using the reference design.

To work with the reference design, your FPGA targeted design must use a streaming data interface with a control signal that indicates the validity of each sample. Wireless HDL Toolbox blocks provide this interface. Use the Sample Control Bus Selector block to separate the valid control signal from the bus.

To deploy a design using the support package, your design must meet these preconditions.

- Each data input or output must be 16 bits. The HDL subsystem that fits into the reference design does not support complex signals at the ports. To handle complex inputs and outputs, model separate I and Q ports at the subsystem boundaries.
- Model all the ports for a given reference design, even when the ports are not used.
- In Simulink, the input and output data and valid signals must be driven at the same sample rate. Therefore, the input and output clock rates of the subsystem must be equal.
- Clock the data and valid signals at the fastest rate of the HDL subsystem.
- For the FPGA-only targeting workflow:

  - Duplex operation is not supported. Use either the transmit or the receive operation, but not both.
- For the hardware-software co-design workflow:

  - Duplex operation is supported. You can use both the Transmitter and Receiver blocks in the same design.
  - AXI4-Lite register ports can be clocked at arbitrary rates.
  - In single-channel mode, you can transmit or receive data frames containing an even number of samples only. If you use an odd number of samples, the software inserts a zero sample at the end of each frame.

The real-time design encounters a larger volume of data and a larger set of state progressions than you can simulate in Simulink. Make sure to model and generate control logic to handle the restart between subframes. Consider adding extra subsystem ports for debug visibility of these extended states once the design is deployed to the board.

Design for Debugging

Once the design is deployed to the board, you have much less visibility of the internal signals in your design. To improve visibility, you can add temporary output ports to your subsystem before you generate your IP core. Signals that can help with debugging are design state, mux select signals or other control parameters, and data values at intermediate stages of the data path. You can also add input ports and muxes to give the option for external control of parameters such as mux select signals and gain values.

When you simulate the design on the board in External mode, you can drive and view these ports from Simulink. The generated software model provides a Simulink interface to the input and output ports of your design while it is running on the board.

Once you are confident that your design is behaving as intended, you can remove these ports and regenerate the IP core.

Another debugging strategy is to include a known input signal stored in memory on the FPGA. This memory can be part of the generated HDL code from your Simulink model. The "LTE MIB Recovery and Cell Scanner Using Analog Devices AD9361/AD9364" (Communications Toolbox Support Package for Xilinx Zynq-Based Radio) example shows an input port `externalDataSel` that provides a switch between a stored data set and the live data from the radio.

See Also

More About

- "Communications Toolbox Support Package for Xilinx Zynq-Based Radio"
- "FPGA Targeting Workflow" (Communications Toolbox Support Package for Xilinx Zynq-Based Radio)
- "Hardware-Software Co-Design Workflow" (Communications Toolbox Support Package for Xilinx Zynq-Based Radio)
- "SoC Blockset Support Package for Xilinx Devices"
- "Generate Design Using SoC Builder" (SoC Blockset Support Package for Xilinx Devices)

Reference Page Examples

Modulate and Demodulate OFDM Streaming Samples

Append CRC Checksum to Streaming Data

This example shows how to use the LTE CRC Encoder block to encode data, and how to compare the hardware-friendly design with the results from LTE Toolbox™. The workflow follows these steps:

1 Generate frames of random input samples in MATLAB.

2 Generate and append a CRC checksum using the LTE Toolbox function `lteCRCEncode`.

3 Convert framed input data to a stream of samples and import the stream into Simulink®.

4 To encode the samples using a hardware-friendly architecture, run the Simulink model, which contains the Wireless HDL Toolbox™ block LTE CRC Encoder.

5 Export the stream of bits, which now has an appended CRC checksum, to the MATLAB® workspace.

6 Convert the sample stream back to framed data, and compare the frames with the reference frames and checksum.

Generate input data frames. Generate reference output data using `lteCRCEncode`.

```
frameLength = 256;
numframes    = 2;
rng(0);

txframes     = cell(1,numframes);
txcodeword   = cell(1,numframes);
rxSoftframes = cell(1,numframes);

for ii = 1:numframes

    txframes{ii}  = randi([0 1],frameLength,1)>0.5;

    CRCType = '24B';
    CRCMask = 50;
    txcodeword{ii} = lteCRCEncode(txframes{ii},CRCType,CRCMask);

end
```

Serialize input data for the Simulink model. Leave enough time between frames for each frame to be fully encoded before the next one starts. For CRC 24 encoding, the checksum adds 24 parity bits at the end of the frame. The hardware-friendly algorithm also adds *CRCLength* + 3 cycles of latency.

```
idleCyclesBetweenSamples = 0;
idleCyclesBetweenFrames  = 24+27;
outputSize               = 1;

[sampleIn,ctrlIn] = whdlFramesToSamples(...
    txframes,idleCyclesBetweenSamples,idleCyclesBetweenFrames,outputSize);
```

Run the Simulink model.

```
sampletime = 1;
simTime = length(ctrlIn);
modelname = 'ltehdlCRCEncoderModel';
open(modelname);
sim(modelname);
```
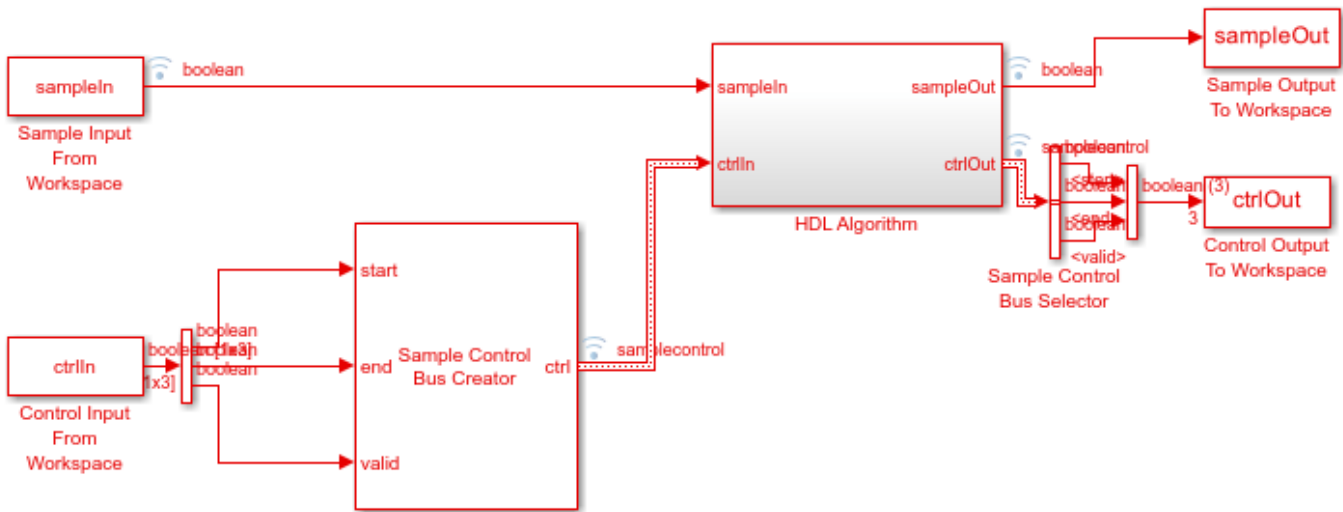
Copyright 2017 The MathWorks, Inc.

The Simulink model exports `sampleOut` and `ctrlOut` back to the MATLAB workspace. Deserialize the output samples, and compare the framed data to the reference data.

```
txhdlframes = whdlSamplesToFrames(sampleOut,ctrlOut);

fprintf('\nLTE CRC Encoder\n');
for ii = 1:numframes
    numBitsDiff = sum(double(txcodeword{ii})-double(txhdlframes{ii}));
    fprintf(['  Frame %d: Behavioral and ' ...
        'HDL simulation differ by %d bits\n'], ii, numBitsDiff);
end
```

```
Maximum frame size computed to be 280 samples.

LTE CRC Encoder
  Frame 1: Behavioral and HDL simulation differ by 0 bits
  Frame 2: Behavioral and HDL simulation differ by 0 bits
```

See Also

Blocks
LTE CRC Encoder

Functions
lteCRCEncode

More About

•   "Check for CRC Errors in Streaming Samples" on page 3-4

Check for CRC Errors in Streaming Samples

This example shows how to use the LTE CRC Decoder block to check encoded data, and how to compare the hardware-friendly design with the results from LTE Toolbox™. The workflow follows these steps:

1 Generate frames of random input samples in MATLAB.
2 Generate and append the CRC checksum using the LTE Toolbox function `lteCRCEncode`.
3 Convert framed input data and checksum to a stream of samples and import it to Simulink®.
4 To check the samples against the checksum using a hardware-friendly architecture, run the Simulink model. The model contains the Wireless HDL Toolbox™ block LTE CRC Decoder.
5 Export the stream of samples back to the MATLAB® workspace.
6 Convert the sample stream back to framed data, and compare the frames with the reference data.

Generate input data frames, then generate the CRC checksum using `lteCRCEncode`.

```
frameLength = 256;
numframes   = 2;
rng(0);

txframes     = cell(1,numframes);
txcodeword   = cell(1,numframes);
rxSoftframes = cell(1,numframes);

for ii = 1:numframes

    txframes{ii}  = randi([0 1],frameLength,1)>0.5;

    CRCType = '24B';
    CRCMask = 50;
    txcodeword{ii} = boolean(lteCRCEncode(txframes{ii},CRCType,CRCMask));

end
```

Serialize input data for the Simulink model. The LTE CRC Decoder block does not require any space between frames, but the hardware-friendly algorithm adds latency of (3 * *CRCLength* / *SampleSize*) + 5 cycles. This example uses scalar input samples, so the latency is (3 * *CRCLength*) + 5.

```
idleCyclesBetweenSamples = 0;
idleCyclesBetweenFrames  = 77;
samplesizeIn             = 1;

[sampleIn,ctrlIn] = whdlFramesToSamples(...
    txcodeword,idleCyclesBetweenSamples,idleCyclesBetweenFrames,samplesizeIn);
```

Run the Simulink model.

```
sampletime = 1;
simTime = length(ctrlIn);
modelName = 'ltehdlCRCDecoderModel';
open_system(modelName);
sim(modelName);
```
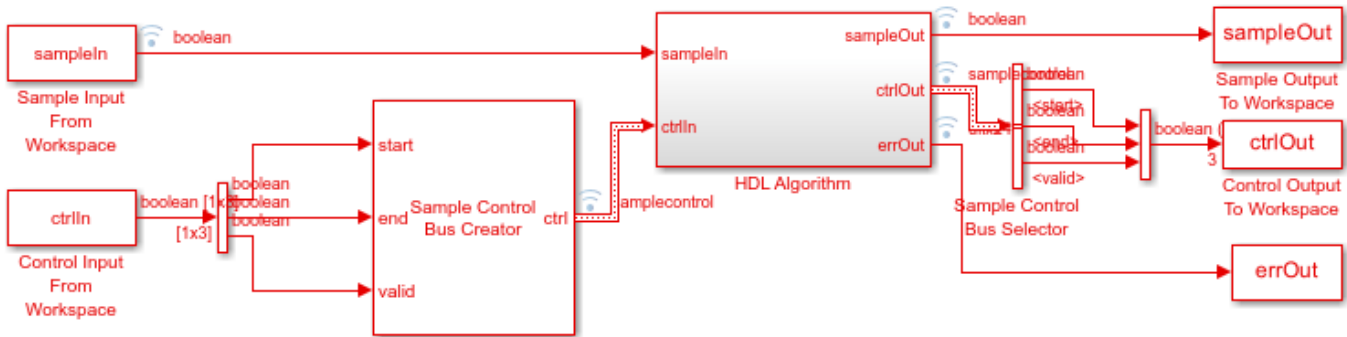
Copyright 2017 The MathWorks, Inc.

The Simulink model exports `sampleOut` and `ctrlOut` back to the MATLAB workspace. Deserialize the output samples, and compare the framed data to the input frames.

```
txhdlframes = whdlSamplesToFrames(sampleOut,ctrlOut);

fprintf('\nLTE CRC Decoder\n');
for ii = 1:numframes
    numBitsDiff = sum(double(txframes{ii})-double(txhdlframes{ii}));
    fprintf(['  Frame %d: Behavioral and ' ...
        'HDL simulation differ by %d bits\n'], ii, numBitsDiff);
end
```

```
Maximum frame size computed to be 256 samples.

LTE CRC Decoder
  Frame 1: Behavioral and HDL simulation differ by 0 bits
  Frame 2: Behavioral and HDL simulation differ by 0 bits
```

See Also

Blocks
LTE CRC Decoder

Functions
lteCRCDecode

More About

- "Append CRC Checksum to Streaming Data" on page 3-2

Turbo Encode Streaming Samples

This example shows how to use the LTE Turbo Encoder block to encode data, and how to compare the hardware-friendly design with the results from LTE Toolbox™. The workflow follows these steps:

1. Generate frames of random input samples in MATLAB®.
2. Encode the data using the LTE Toolbox function `lteTurboEncode`.
3. Convert framed input data to a stream of samples and import the stream into Simulink®.
4. To encode the samples using a hardware-friendly architecture, run the Simulink model, which contains the Wireless HDL Toolbox™ block LTE Turbo Encoder.
5. Export the stream of encoded samples to the MATLAB workspace.
6. Convert the sample stream back to framed data, and compare the frames with the reference data.

Generate input data frames. Generate reference encoded data using `lteTurboEncode`.

```
rng(0);
turboframesize = 40;
numframes = 2;

txBits    = cell(1,numframes);
codedData = cell(1,numframes);

for ii = 1:numframes
    txBits{ii} = logical(randi([0 1],turboframesize,1));
    codedData{ii} = lteTurboEncode(txBits{ii});
end
```

Serialize input data for the Simulink model. Leave enough time between frames for each frame to be fully encoded before the next one starts. The LTE Turbo Encoder block takes `inframesize` + 16 cycles to complete encoding of a frame.

```
inframes = txBits;

inframesize = size(inframes{1},1);

idlecyclesbetweensamples = 0;
idlecyclesbetweenframes = inframesize+16;

[sampleIn,ctrlIn] = ...
    whdlFramesToSamples(inframes, ...
                        idlecyclesbetweensamples, ...
                        idlecyclesbetweenframes);
```

Run the Simulink model. The simulation time equals the number of input samples. Because of the added idle cycles between frames, the streaming input data includes enough cycles for the model to complete encoding of both frames.

```
sampletime = 1;
samplesizeIn = 1;
simTime = size(ctrlIn,1);
modelname = 'ltehdlTurboEncoderModel';
open_system(modelname);
sim(modelname);
```
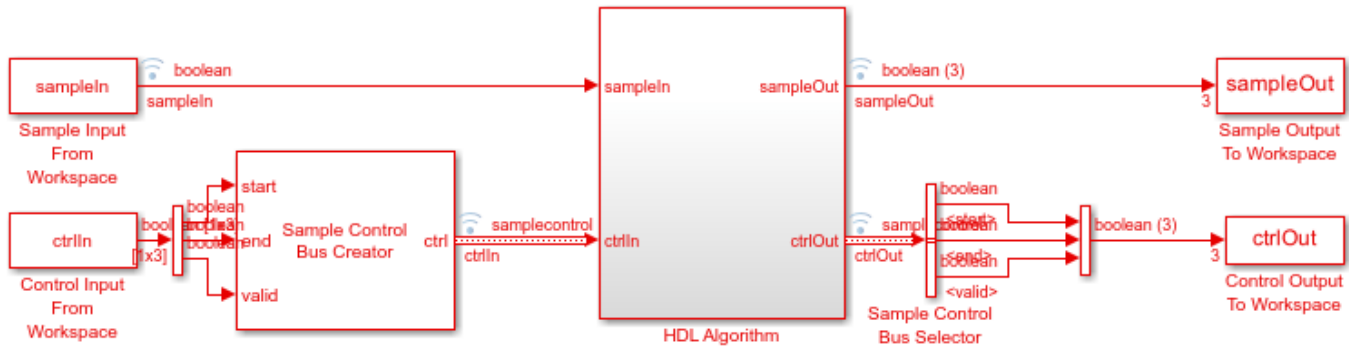
Copyright 2017-2021 The MathWorks, Inc.

The Simulink model exports `sampleOut_ts` and `ctrlOut_ts` back to the MATLAB workspace. Deserialize the output samples, and compare the framed data to the reference encoded frames.

The output samples of the LTE Turbo Encoder block are interleaved with the parity bits.

Hardware-friendly output: `S_1 P1_1 P2_1 S2 P1_2 P2_2 ... Sn P1_n P2_n`

LTE Toolbox output: `S_1 S_2 ... S_n P1_1 P1_2 ... P1_n P2_1 P2_2 ... P2_n`

Reorder the samples using the interleave option of the `whdlSamplesToFrames` function. Compare the reordered output frames with the reference encoded frames.

```
sampleOut = sampleOut';
interleaveSamples = true;
outframes = whdlSamplesToFrames(sampleOut(:),ctrlOut,[],interleaveSamples);

fprintf('\nLTE Turbo Encoder\n');
for ii = 1:numframes
    numBitsDiff = sum(outframes{ii} ~= codedData{ii});
    fprintf(['  Frame %d: Behavioral and ' ...
        'HDL simulation differ by %d bits\n'],ii,numBitsDiff);
end
```

```
Maximum frame size computed to be 132 samples.

LTE Turbo Encoder
  Frame 1: Behavioral and HDL simulation differ by 0 bits
  Frame 2: Behavioral and HDL simulation differ by 0 bits
```

See Also

Blocks
LTE Turbo Encoder

Functions
lteTurboEncode

More About

- "Turbo Decode Streaming Samples" on page 3-9

Turbo Decode Streaming Samples

This example shows how to use the **LTE Turbo Decoder** block to decode data, and how to compare the hardware-friendly design with the results from LTE Toolbox™.

1   Generate frames of random input samples in MATLAB®. Encode the samples and add noise to the data.

2   Decode the data using the LTE Toolbox function, `lteTurboDecode`.

3   Convert framed input data to a stream of samples and import the stream into Simulink®.

4   To decode the samples using a hardware-friendly architecture, execute the Simulink model, which contains the **LTE Turbo Decoder** block.

5   Export the stream of decoded bits to the MATLAB workspace.

6   Convert the sample stream back to framed data, and compare the frames with the decoded frames from Step 2.

Generate input data frames. Turbo encode the data, modulate the message, and add noise to the resulting constellation. Demodulate the noisy constellation and generate soft bit values. Generate reference decoded data using `lteTurboDecode`. For the hardware-friendly model, convert the soft bits into a fixed-point data type.

```
rng(0);
numframes = 2;

txBits   = cell(1,numframes);
softBits = cell(1,numframes);
rxBits   = cell(1,numframes);
inframes = cell(1,numframes);

for ii = 1:numframes
    txBits{ii} = randi([0 1],6144,1);
    codedData = lteTurboEncode(txBits{ii});
    txSymbols = lteSymbolModulate(codedData,'QPSK');
    noise = 0.5*complex(randn(size(txSymbols)),randn(size(txSymbols)));
    rxSymbols = txSymbols + noise;
    softBits{ii} = lteSymbolDemodulate(rxSymbols,'QPSK','Soft');
    rxBits{ii} = lteTurboDecode(softBits{ii});
    inframes{ii} = fi(softBits{ii},1,5,2);
end
```

Serialize input data for the Simulink model. Leave enough time between frames for each frame to be fully decoded before the next one starts. The **LTE Turbo Decoder** block takes 2 * numTurboIterations * *HalfIterationLatency* + (`inframesize` / `samplesizeIn`) cycles to complete decoding of a frame. For details of the *HalfIterationLatency* calculation see the Turbo Decoder block reference page.

The **LTE Turbo Decoder** block expects input samples are interleaved with the parity bits.

Hardware-friendly input: `S_1 P1_1 P2_1 S2 P1_2 P2_2 ... Sn P1_n P2_n`

LTE Toolbox input: `S_1 S_2 ... S_n P1_1 P1_2 ... P1_n P2_1 P2_2 ... P2_n`

Reorder the samples using the interleave option of the `whdlFramesToSamples` function.

```
inframesize = size(inframes{1},1); %includes 4 tail bit samples
encoderrate = 3; % rate 1/3 Turbo code
samplesizeIn = encoderrate; % 3 samples in at a time

idlecyclesbetweensamples = 0;
outframesize = size(txBits{1},1);
numTurboIterations = 6;
halfIterationLatency = (ceil(outframesize/32)+3)*32; % window size=32
algframedelay = 2*numTurboIterations*halfIterationLatency+(inframesize/samplesizeIn);
idlecyclesbetweenframes = algframedelay;

interleaveSamples = true;
[sampleIn,ctrlIn] = ...
    whdlFramesToSamples(inframes, ...
                        idlecyclesbetweensamples, ...
                        idlecyclesbetweenframes, ...
                        samplesizeIn, ...
                        interleaveSamples);
```

Run the Simulink model. The simulation time equals the number of input samples. Because of the added idle cycles between frames, the streaming input data includes enough cycles for the model to complete decoding of both frames.

```
sampletime = 1;
simTime = size(ctrlIn, 1);
modelname = 'ltehdlTurboDecoderModel';
open_system(modelname);
sim(modelname);
```
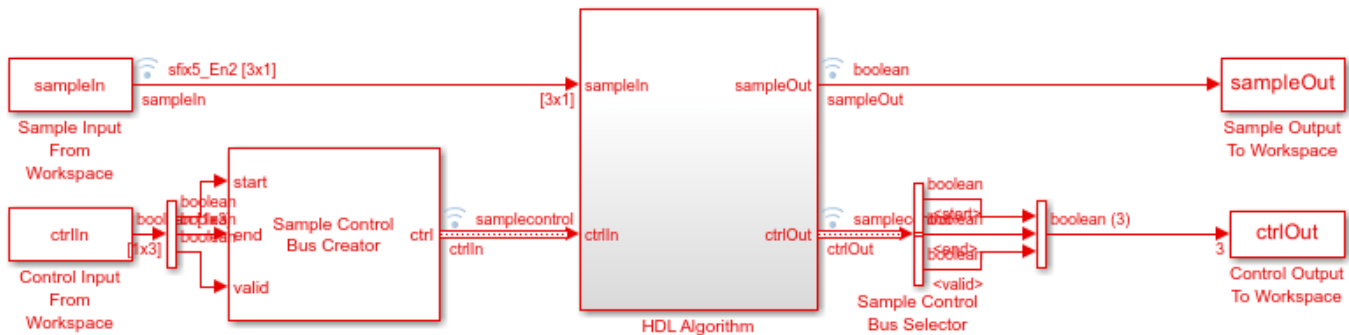

Copyright 2017-2021 The MathWorks, Inc.

The Simulink model exports `sampleOut` and `ctrlOut` back to the MATLAB workspace. De-serialize the output samples, and compare to the decoded frame.

```
outframes = whdlSamplesToFrames(sampleOut,ctrlOut);

fprintf('\nLTE Turbo Decoder\n');
for ii = 1:numframes
    numBitsDiff = sum(outframes{ii} ~= rxBits{ii});
    fprintf(['  Frame %d: Behavioral and ' ...
        'HDL simulation differ by %d bits\n'],ii,numBitsDiff);
end
```

```
Maximum frame size computed to be 6144 samples.
```

```
LTE Turbo Decoder
  Frame 1: Behavioral and HDL simulation differ by 0 bits
  Frame 2: Behavioral and HDL simulation differ by 0 bits
```

See Also

Blocks
LTE Turbo Decoder

Functions
`lteTurboDecode`

More About

- "Turbo Encode Streaming Samples" on page 3-6

Convolutional Encode of Streaming Samples

This example shows how to use the LTE Convolutional Encoder block to encode data, and how to compare the hardware-friendly design with the results from LTE Toolbox™. The workflow follows these steps:

1 Generate frames of random input samples in MATLAB®.

2 Encode the data using the LTE Toolbox function `lteConvolutionalEncode`.

3 Convert framed input data to a stream of samples and import the stream into Simulink®.

4 To encode the samples using a hardware-friendly architecture, run the Simulink model, which contains the Wireless HDL Toolbox™ block LTE Convolutional Encoder.

5 Export the stream of encoded bits to the MATLAB workspace.

6 Convert the sample stream back to framed data, and compare the frames with the reference data.

Generate input data frames. Generate reference encoded data using `lteConvolutionalEncode`.

```
rng(0);
frameLength = 256;
numframes   = 2;

txframes     = cell(1,numframes);
txcodeword   = cell(1,numframes);
rxSoftframes = cell(1,numframes);

for k = 1:numframes
    txframes{k}   = randi([0 1],frameLength,1)>0.5;
    txcodeword{k} = lteConvolutionalEncode(txframes{k});
end
```

Serialize input data for the Simulink model. Leave enough time between frames so that each frame is fully encoded before the next one starts. The block takes `frameLength` + 5 cycles to encode the frame.

```
idleCyclesBetweenSamples = 0;
idleCyclesBetweenFrames  = frameLength+5;

[sampleIn,ctrlIn] = whdlFramesToSamples(...
    txframes,idleCyclesBetweenSamples,idleCyclesBetweenFrames);
```

Run the Simulink model. Because of the added idle cycles between frames, the streaming input data includes enough cycles for the model to complete encoding of both frames.

```
sampletime = 1;
samplesizeIn = 1;
simTime = size(ctrlIn,1);
modelname = 'ltehdlConvolutionalEncoderModel';
open_system(modelname);
sim(modelname);
```
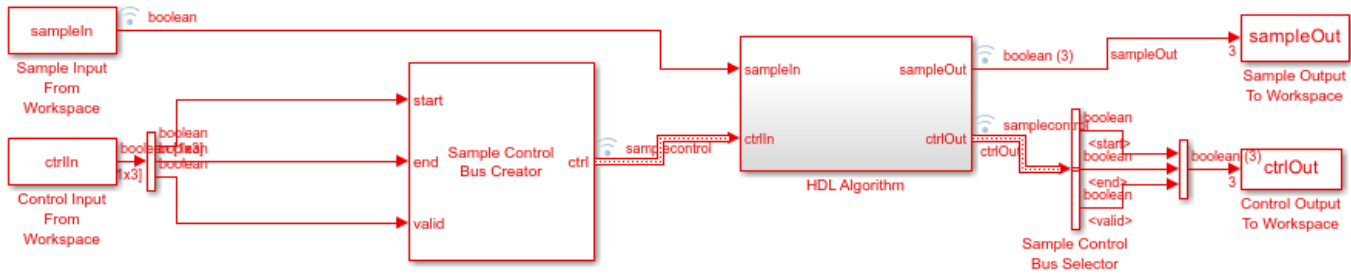
Copyright 2017 The MathWorks, Inc.

The Simulink model exports `sampleOut` and `ctrlOut` back to the MATLAB workspace. Deserialize the output samples, and compare them to the encoded frame.

The output samples of the LTE Convolutional Encoder block are the interleaved results of the three polynomials.

- Hardware-friendly output: `G0_1 G1_1 G2_1 G0_2 G1_2 G2_2 ... Gn G1_n G2_n`

- LTE Toolbox output: `G0_1 G0_2 ... G0_n G1_1 G1_2 ... G1_n G2_1 G2_2 ... G2_n`

The `whdlSamplesToFrames` function provides an option to reorder the samples. Compare the reordered output frames with the reference encoded frames.

```
interleaveSamples = true;
sampleOut = sampleOut';
txhdlframes = whdlSamplesToFrames(sampleOut(:),ctrlOut,[],interleaveSamples);

fprintf('\nLTE Convolutional Encoder\n');
for k = 1:numframes
    numBitsDiff = sum(double(txcodeword{k})-double(txhdlframes{k}));
    fprintf(['  Frame %d: Behavioral and ' ...
        'HDL simulation differ by %d bits\n'],k,numBitsDiff);
end
```

```
Maximum frame size computed to be 768 samples.

LTE Convolutional Encoder
  Frame 1: Behavioral and HDL simulation differ by 0 bits
  Frame 2: Behavioral and HDL simulation differ by 0 bits
```

See Also

Blocks
LTE Convolutional Encoder

Functions
lteConvolutionalEncode

More About

- "Convolutional Decode of Streaming Samples" on page 3-14

Convolutional Decode of Streaming Samples

This example shows how to use the LTE Convolutional Decoder block to decode data, and how to compare the hardware-friendly design with the results from LTE Toolbox™. The workflow follows these steps:

1 Generate LTE convolutionally encoded messages in MATLAB®, using LTE Toolbox.
2 Call Communications Toolbox™ functions to perform BPSK modulation, transmission through an AWGN channel, and BPSK demodulation. The result is soft-bit values that represent log-likelihood ratios (LLRs).
3 Quantize the soft bits according to the signal-to-noise ration (SNR).
4 Convert framed input data to a stream of samples and import the stream into Simulink®.
5 To decode the samples using a hardware-friendly architecture, execute the Simulink model, which contains the LTE Convolutional Decoder block.
6 Export the stream of decoded bits to the MATLAB workspace.
7 Convert the sample stream back to framed data, and compare the frames with the original input frames.

Calculate the channel SNR and create the modulator, channel, and demodulator System objects. `EbNo` is the ratio of energy per uncoded bit to noise spectral density, in dB. `EcNo` is the ratio of energy per channel bit to noise spectral density, in dB. The code rate of the convolutional encoder is 1/3. Therefore each transmitted bit contains 1/3 of a bit of information.

```
EbNo = 10;
EcNo = EbNo - 10*log10(3);

modulator = comm.BPSKModulator;
channel = comm.AWGNChannel('EbNo',EcNo);
demodulator = comm.BPSKDemodulator('DecisionMethod','Log-likelihood ratio');
```

Generate input data frames. Encode the data, modulate the message, and add channel effects to the resulting constellation. Demodulate the transmitted constellation and generate soft-bit values. For the hardware-friendly model, convert the soft bits into a fixed-point data type. The optimal soft-bit quantization step size is a function of the noise spectral density, `No`.

```
rng(0);
messageLength = 100;
numframes = 2;
numSoftBits = 5;

txMessages     = cell(1,numframes);
rxSoftMessages = cell(1,numframes);

No            = 10^((-EcNo)/10);
quantStepSize = sqrt(No/2^numSoftBits);

for k = 1:numframes

    txMessages{k}  = randi([0 1],messageLength,1,'int8');
    txCodeword = lteConvolutionalEncode(txMessages{k});

    modOut     = modulator.step(txCodeword);
```

```
    chanOut   = channel.step(modOut);
    demodOut  = -demodulator.step(chanOut)/4;

    rxSoftMessagesDouble = demodOut./quantStepSize;
    rxSoftMessages{k} = fi(rxSoftMessagesDouble,1,numSoftBits,0);
```

end

Serialize input data for the Simulink model. Leave enough time between frames so that each frame is fully decoded before the next one starts. The LTE Convolutional Decoder block takes (2 * messageLength) + 140 cycles to complete decoding of a frame.

The LTE Convolutional Decoder block expects the input data to contain the three encoded bits interleaved.

- Hardware-friendly input: G0_1 G1_1 G2_1 G0_2 G1_2 G2_2 ... G0_n G1_n G2_n

- LTE Toolbox input: G0_1 G0_2 ... G0_n G1_1 G1_2 ... G1_n G2_1 G2_2 ... G2_n

```
idleCyclesBetweenSamples = 0;
idleCyclesBetweenFrames  = 2 * messageLength + 140;
samplesizeIn             = 3;
interleaveSamples        = true;

[sampleIn,ctrlIn] = whdlFramesToSamples(rxSoftMessages,...
                    idleCyclesBetweenSamples,...
                    idleCyclesBetweenFrames,...
                    samplesizeIn,...
                    interleaveSamples);
```

Run the Simulink model. Because of the added idle cycles between frames, the streaming input variables include enough cycles for the model to complete decoding of both frames.

```
sampletime= 1;
simTime = size(ctrlIn,1);
modelname = 'ltehdlConvolutionalDecoderModel';
open(modelname);
sim(modelname);
```
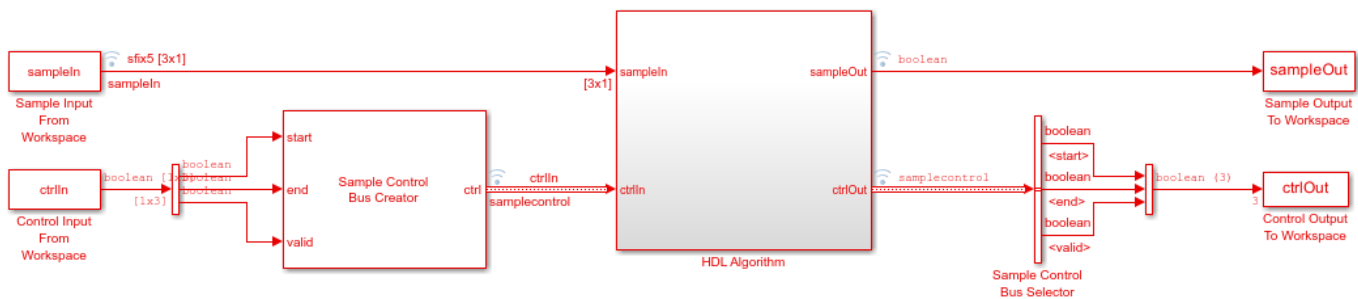

Copyright 2017 The MathWorks, Inc.

The Simulink model exports sampleOut and ctrlOut back to the MATLAB workspace. Deserialize the output samples, and compare to the decoded frame.

```
rxMessages = whdlSamplesToFrames(sampleOut,ctrlOut);
```

```matlab
fprintf('\nLTE Convolutional Decoder\n');
for k = 1:numframes
    numBitsDiff = sum(double(txMessages{k})-double(rxMessages{k}));
    fprintf(['  Frame %d: Behavioral and ' ...
        'HDL simulation differ by %d bits\n'], k, numBitsDiff);
end
```

```
Maximum frame size computed to be 100 samples.

LTE Convolutional Decoder
  Frame 1: Behavioral and HDL simulation differ by 0 bits
  Frame 2: Behavioral and HDL simulation differ by 0 bits
```

See Also

Blocks
LTE Convolutional Decoder

Functions
lteConvolutionalDecode

More About

- "Convolutional Encode of Streaming Samples" on page 3-12

Descrambling with Gold Sequence Generator

This example shows how to use the LTE Gold Sequence Generator block to implement an LTE descrambler.

The example model generates random I-Q pairs, multiplies the I and Q components with a generated Gold sequence, and interleaves the I and Q into a single data stream.

You can generate HDL from the HDL Descrambler subsystem.
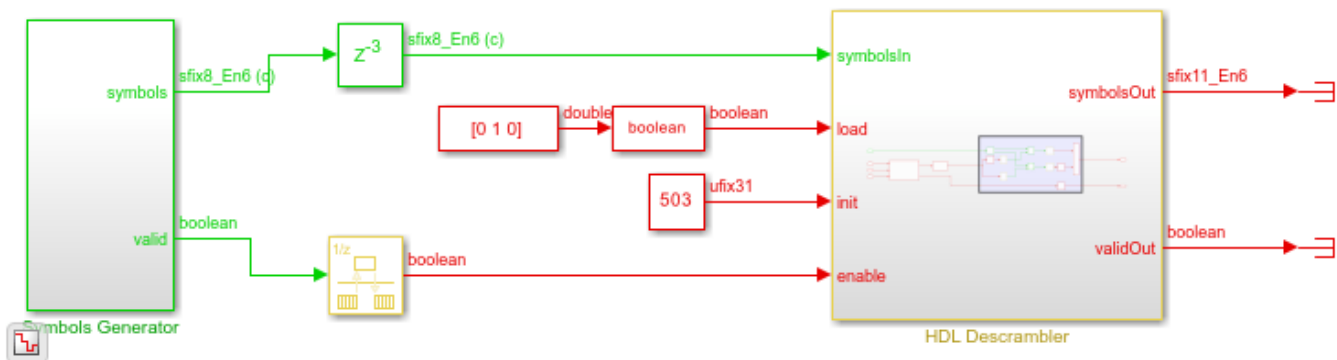
The LTE Gold Sequence Generator block has no block parameters. It is configured to match the polynomial and shift length required by LTE standard TS 36.212. You must initialize the sequence with a 31-bit value on the **init** port, and load the value into the block by setting the **load** signal to 1 for one cycle. The **enable** signal generates the Gold sequence values. The output **valid** signal indicates when the output is available.
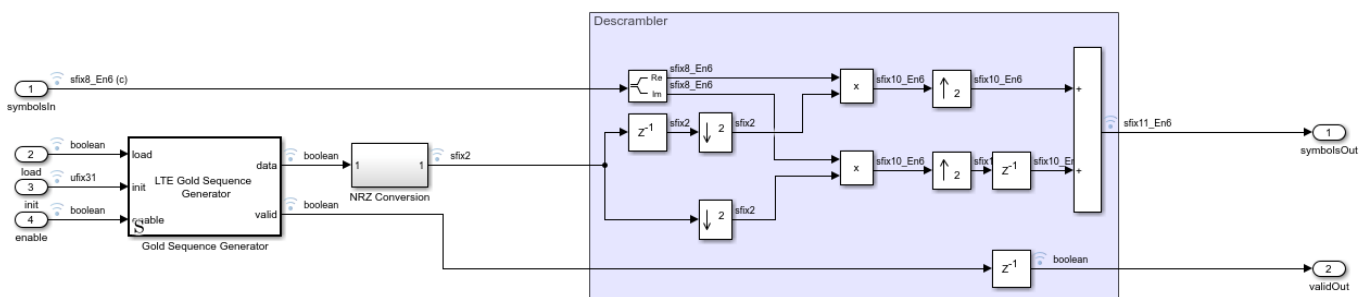
You can add data logging on the signals and use the Logic Analyzer to view the waveforms.
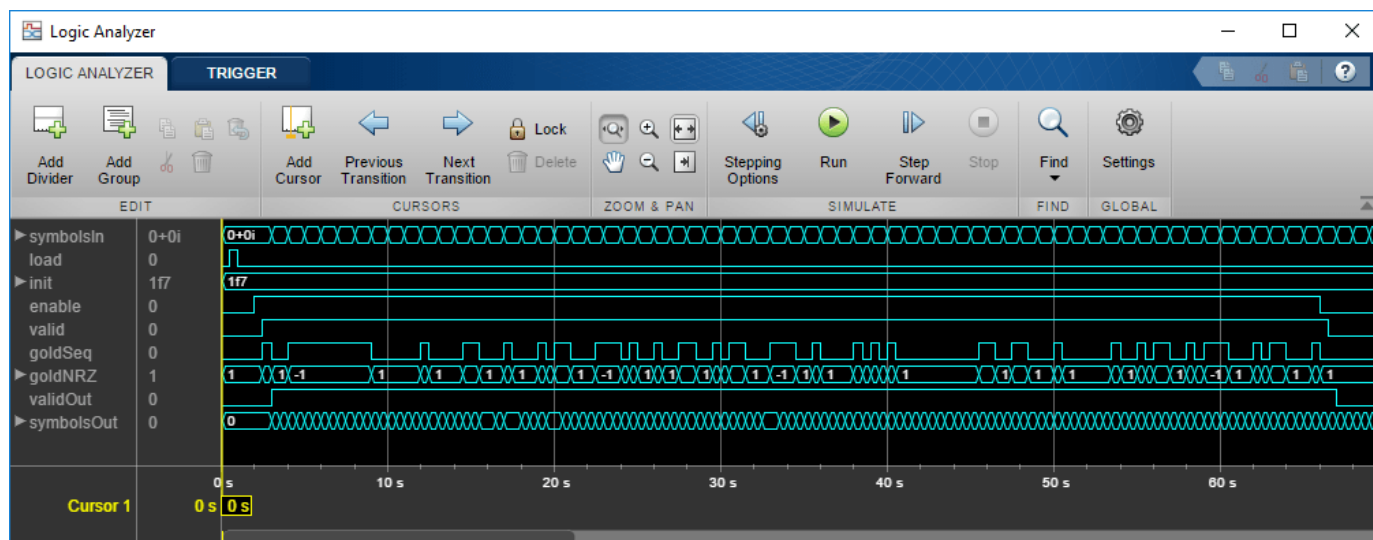
To generate and check the HDL code referenced in this example, you must have an HDL Coder™ license.

To generate the HDL code, use the following command:

```
makehdl('ltehdlGoldDescramblerModel/HDL Descrambler')
```

To generate a test bench, use the following command:

```
makehdltb('ltehdlGoldDescramblerModel/HDL Descrambler')
```

See Also

Blocks
LTE Gold Sequence Generator

Parallel Gold Sequence Generation

This example shows how to use the LTE Gold Sequence Generator block to generate multiple sequences in parallel for use in channel estimation.

The example model initializes the LTE Gold Sequence Generator block with a vector that represents the **init** values for each of four channels. The block returns four independent Gold sequences.

You can generate HDL from the HDL Gold Sequence Generator subsystem.
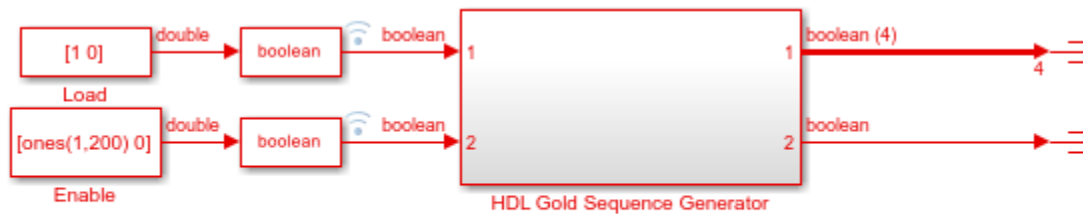
Copyright 2017 The MathWorks, Inc.

The LTE Gold Sequence Generator block has no block parameters. It is configured to match the polynomial and shift length required by LTE standard TS 36.212. You must initialize the sequence with a 31-bit value on the **init** port, and load the value into the block by setting the **load** signal to 1 for one cycle. This model has four **init** values, representing four channels.

The **enable** signal generates the Gold sequence values. The output is a vector of four values. The output **valid** signal indicates when the output data is available.
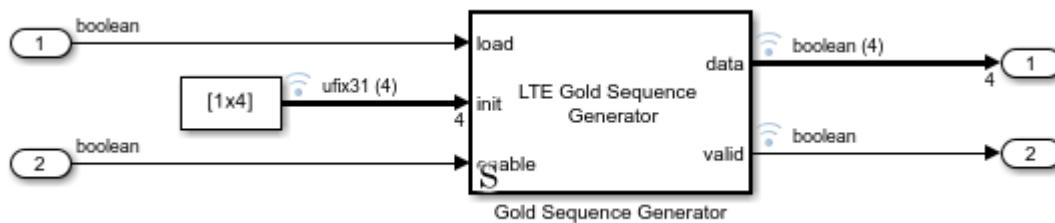
You can add data logging on the signals and use the Logic Analyzer to view the waveforms.
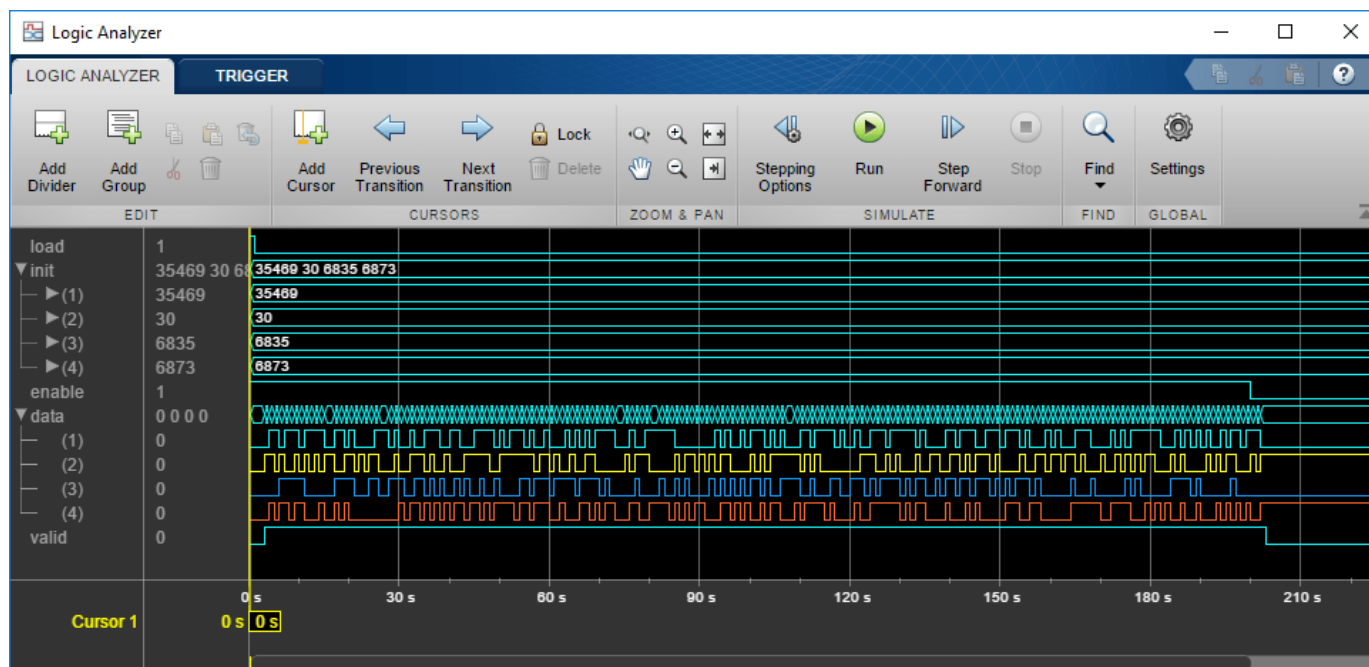
To generate and check the HDL code referenced in this example, you must have an HDL Coder™ license.

To generate the HDL code, use the following command:

makehdl('ltehdlGoldVectorModel/HDL Gold Sequence Generator')

To generate a test bench, use the following command:

makehdltb('ltehdlGoldVectorModel/HDL Gold Sequence Generator')

See Also

Blocks
LTE Gold Sequence Generator

LTE OFDM Demodulation of Streaming Samples

This example shows how to use the LTE OFDM Demodulator block to return the LTE resource grid from streaming samples. You can generate HDL code from this block.

Generate input LTE OFDM symbols using LTE Toolbox™. Select a reference channel based on NDLRB, and specify the type of cyclic prefix.

```
enb = lteRMCDL('R.5');
enb.TotSubframes = 1;
enb.CyclicPrefix = 'Normal';  % or 'Extended'
% ----------------------------------------------------------
%     NDLRB  |   Reference Channel
% ----------------------------------------------------------
%   6        |   R.4
%   15       |   R.5
%   25       |   R.6
%   50       |   R.7
%   75       |   R.8
%   100      |   R.9
% ----------------------------------------------------------

[waveform,LTEGrid,info] = lteRMCDLTool(enb,[1;0;0;1]);
%%In this example, the Input data sample rate parameter is set to |Use
% maximum input data sample rate|. Hence, the LTE OFDM Demodulator block
% expects input samples at 30.72 MHz sample rate to correspond to the
% size of the FFT. The sample rate of |waveform| depends on NDLRB,
% so the generated waveform might be at a lower rate. To generate
% a test waveform, upsample the signal to 30.72 MHz, normalize the power,
% and add noise. Scale the signal magnitude to be in the range -1 to 1 for
% easy conversion to fixed-point types.

FsRx = 30.72e6;
FsTx = info.SamplingRate;
% ------------------------------------------------------------------
%     NDLRB             |   Sampling Rate (MHz)
%   --------------------------------------------------------------
%   1) 6               |   1.92
%   2) 15              |   3.84
%   3) 25              |   7.68
%   4) 50              |   15.36
%   5) 75              |   30.72
%   6) 100             |   30.72
%   --------------------------------------------------------------

tx = resample(waveform,FsRx,FsTx);
avgTxPower =  (tx' * tx) / length(tx);
tx = tx / sqrt(avgTxPower);
n = 0.1 * complex(randn(length(tx),1),randn(length(tx),1));
rx = tx + n;
rx = 0.99 * rx / max(abs(rx));
```

Use an LTE Toolbox function as a behavioral reference for the OFDM demodulation. Downsample the test waveform to the actual sample rate for the selected NDLRB. Then, compensate for the scale factor that results from the difference in FFT sizes.

3-21

```
refInput = resample(rx,FsTx,FsRx);
refGrid = lteOFDMDemodulate(info,refInput);
refGrid = refGrid * FsRx/FsTx;
```

Set up the Simulink™ model input data. Convert the test waveform to a fixed-point data type to model the result from a 12-bit ADC. The Simulink sample time is 30.72 MHz.

The Simulink model imports the sample stream `dataIn` and `validIn`, the input parameters `NDLRB` and `cyclicPrefixType`, and the variable `stopTime`.

```
NDLRB = info.NDLRB;
if strcmp(info.CyclicPrefix,'Normal')
    cyclicPrefixType = false;
else
    cyclicPrefixType = true;
end

sampling_time = 1/FsRx;
dataIn = fi(rx,1,12,11);
validIn = true(length(dataIn),1);
```

Calculate the Simulink simulation time, accounting for the latency of the LTE OFDM Demodulator block. The latency of the FFT is fixed because the block uses a 2048-point FFT. Assume the maximum possible latency of the cyclic prefix removal and subcarrier selection operations. The simulation must run long enough to apply the input data, plus the latency of the final input symbol.

```
FFTlatency = 4137;
CPRemove_max = 512; % extended CP
carrierSelect_max = 424; % NDRLB 100
stopTime = sampling_time*(length(dataIn)+CPRemove_max+FFTlatency+carrierSelect_max);
```

Run the Simulink model. The model imports the `dataIn` and `validIn` structures and returns `dataOut` and `validOut`.

```
modelname = 'LTEOFDMDemodulatorExample';
open(modelname)
set_param(modelname,'SampleTimeColors','on');
set_param(modelname,'SimulationCommand','Update');
sim(modelname)
```
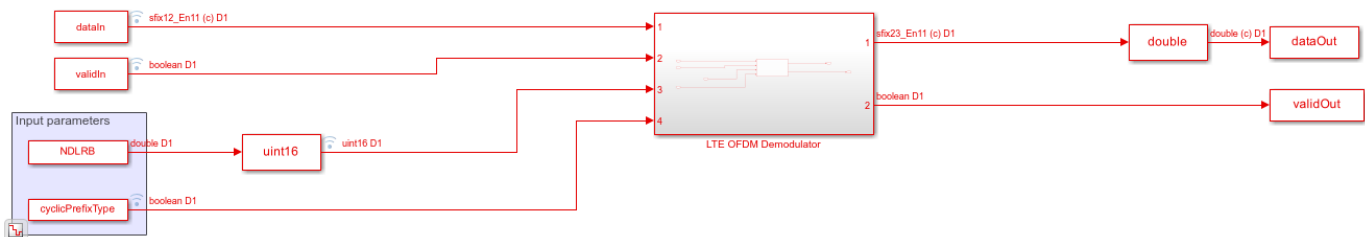

Compare the output of the Simulink model against the behavioral results, and calculate the SQNR of the HDL-optimized LTE OFDM Demodulator block.

```
rxgridSimulink = dataOut(validOut);

figure('units','normalized','outerposition',[0 0 1 1])
subplot(2,1,1)
plot(real(refGrid(:)))
```

```matlab
hold on
plot(squeeze(real(rxgridSimulink)))
legend('Real part of behavioral waveform','Real part of HDL-optimized waveform')
title('Comparison of LTE Time-Domain Downlink Waveform')
xlabel('OFDM Subcarriers')
ylabel('Real Part of Time-Domain Waveform')

subplot(2,1,2)
plot(imag(refGrid(:)))
hold on
plot(squeeze(imag(rxgridSimulink)))
legend('Imag part of behavioral waveform','Imag part of HDL-optimized waveform')
title('Comparison of LTE Time-Domain Downlink Waveform')
xlabel('OFDM Subcarriers')
ylabel('Imag Part of Time-Domain Waveform')

sqnrRealdB = 10*log10(var(real(rxgridSimulink))/abs(var(real(rxgridSimulink))-var(real(refGrid(:
sqnrImagdB = 10*log10(var(imag(rxgridSimulink))/abs(var(imag(rxgridSimulink))-var(imag(refGrid(:

fprintf('\n LTE OFDM Demodulator: \n SQNR of real part is %.2f dB',sqnrRealdB)
fprintf('\n SQNR of imaginary part is %.2f dB\n',sqnrImagdB)
```

```
 LTE OFDM Demodulator:
 SQNR of real part is 25.98 dB
 SQNR of imaginary part is 23.23 dB
```

Comparison of LTE Time-Domain Downlink Waveform

Comparison of LTE Time-Domain Downlink Waveform
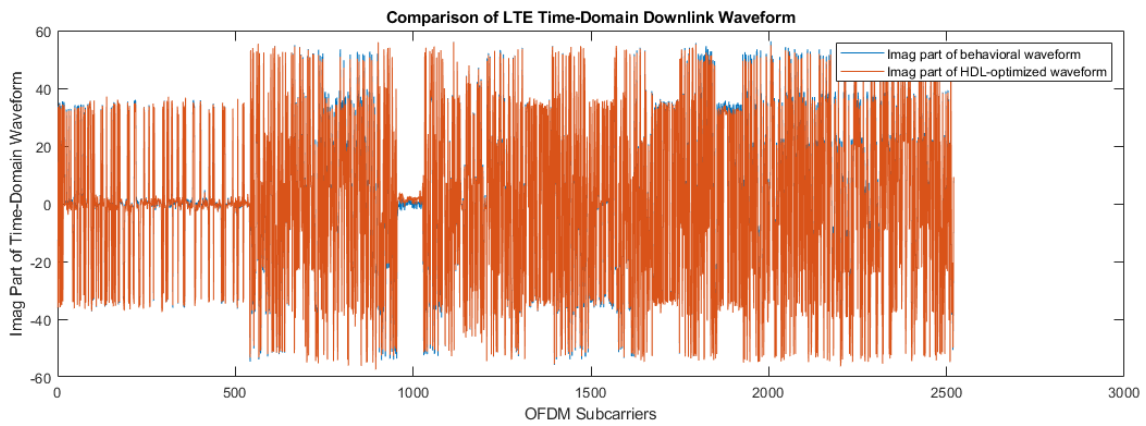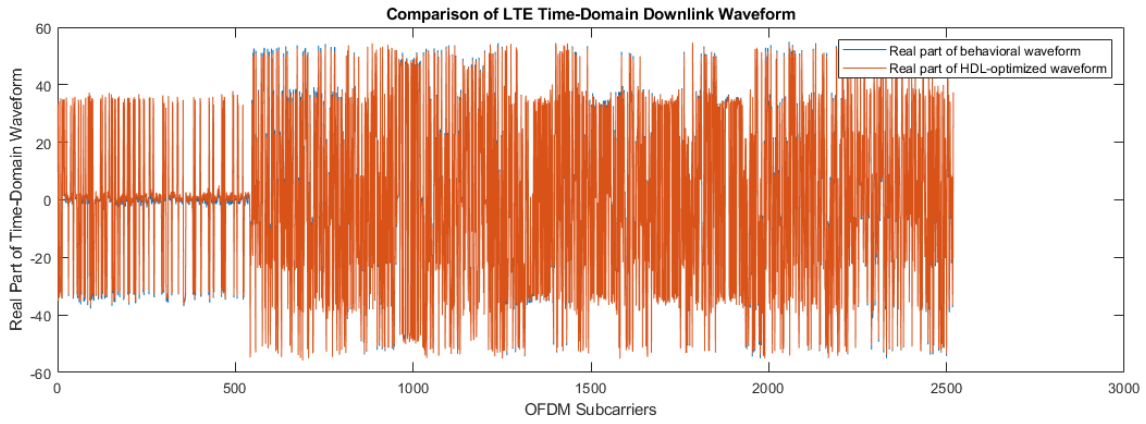
See Also

Blocks
LTE OFDM Demodulator

Reset and Restart LTE OFDM Demodulation

This example shows how to recover the LTE OFDM Demodulator block from an unfinished LTE cell. The input data is truncated to simulate the loss of a signal or a reset from the upstream parts of the receiver. The example model uses the reset signal to clear the internal state counters of the LTE OFDM Demodulator block and then restart calculations on the next cell. In this example, the Input data sample rate parameter of LTE OFDM Demodulator is set to Use maximum input data sample rate. So, the base sampling rate of the block is 30.72 MHz.

Generate two input LTE OFDM cells that use different NDLRBs or different types of cyclic prefix. Upsample both waveforms to the base sampling rate of 30.72 MHz.

```
% -------------------------------------------------------------
%    NDLRB   |   Reference Channel
% -------------------------------------------------------------
%   6        |   R.4
%   15       |   R.5
%   25       |   R.6
%   50       |   R.7
%   75       |   R.8
%   100      |   R.9
% -------------------------------------------------------------

enb1 = lteRMCDL('R.9');
enb1.TotSubframes = 1;
enb1.CyclicPrefix = 'Normal';  % or 'Extended'
[waveform1,grid1,info1] = lteRMCDLTool(enb1,[1;0;0;1]);

enb2 = lteRMCDL('R.6');
enb2.TotSubframes = 1;
enb2.CyclicPrefix = 'Normal';  % or 'Extended'
[waveform2,grid2,info2] = lteRMCDLTool(enb2,[1;0;0;1]);

FsRx = 30.72e6;
tx1 = resample(waveform1,FsRx,info1.SamplingRate);
tx2 = resample(waveform2,FsRx,info2.SamplingRate);
```

Truncate the first waveform two-thirds through the cell. Concatenate the shortened cell with the second generated cell, leaving some invalid samples in between. Add noise, and scale the signal magnitude to be in the range [-1, 1] for easy conversion to fixed point.

```
tx1 = tx1(1:2*length(tx1)/3);

Lgap1 = 3000;
Lgap2 = 10000;
rx = [zeros(Lgap1,1); tx1; zeros(Lgap2,1); tx2];

L = length(rx);
rx = rx + 2e-4*complex(randn(L,1),randn(L,1));

dataIn_fp = 0.99*rx/max(abs(rx));
```

The LTE OFDM Demodulator block maintains internal counters of subframes within each cell. The block requires a reset after an incomplete cell to clear the counters before it can correctly demodulate subsequent cells. Create a reset pulse signal at the end of the first waveform.

```
resetIndex = Lgap1 + length(tx1);
resetIn = false(length(rx),1);
resetIn(resetIndex) = true;
```

Set up the Simulink™ model input data. Convert the test waveform to a fixed-point data type to model the result from a 12-bit ADC. The Simulink sample time is 30.72 MHz.

The Simulink model imports the sample stream `dataIn` and `validIn`, the input parameters `NDLRB` and `cyclicPrefixType`, the reset signal `resetIn`, and the simulation length `stopTime`.

```
dataIn = fi(dataIn_fp,1,12,11);

validIn = [false(Lgap1,1); true(length(tx1),1); false(Lgap2,1); true(length(tx2),1)];
validIn(resetIndex+1:Lgap1+length(tx1)) = false;

NDLRB = uint16([info1.NDLRB*ones(Lgap1 + length(tx1),1); info2.NDLRB*ones(Lgap2 + length(tx2),1)]

cpType1 = strcmp(info1.CyclicPrefix,'Extended');
cpType2 = strcmp(info2.CyclicPrefix,'Extended');
cyclicPrefixType = [repmat(cpType1,Lgap1 + length(tx1),1); repmat(cpType2,Lgap2 + length(tx2),1)]
```

Calculate the Simulink simulation time, accounting for the latency of the LTE OFDM Demodulator block. The latency of the FFT is fixed because the block uses a 2048-point FFT. Assume the maximum possible latency of the cyclic prefix removal and the subcarrier selection operations.

```
FFTlatency = 4137;
CPRemove_max = 512; % extended CP
carrierSelect_max = 424; % NDRLB 100

sampling_time = 1/FsRx;
stopTime = sampling_time*(length(dataIn) + CPRemove_max + FFTlatency + carrierSelect_max);
```

Run the Simulink model. The model imports the `dataIn` and `validIn` structures and returns `dataOut` and `validOut`.

```
modelname = 'LTEOFDMDemodResetExample';
open(modelname)
set_param(modelname,'SampleTimeColors','on');
set_param(modelname,'SimulationCommand','Update');
sim(modelname)
```
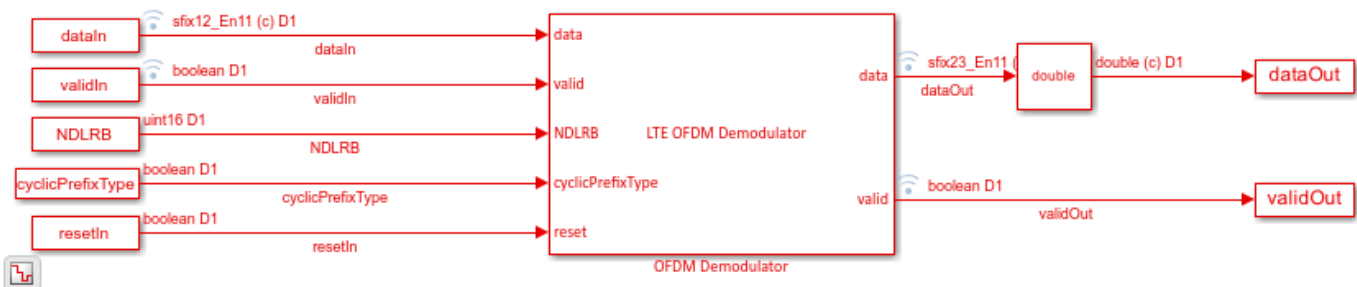

Split `dataOut` and `validOut` into two parts as divided by the reset pulse. The block applies the reset to the output data one cycle after the reset is applied on the input. Use the `validOut` signal to collect the valid output samples.

```
dataOut1 = dataOut(1:resetIndex);
dataOut2 = dataOut(resetIndex+1:end);
```

```matlab
validOut1 = validOut(1:resetIndex);
validOut2 = validOut(resetIndex+1:end);

demodData1 = dataOut1(validOut1);
demodData2 = dataOut2(validOut2);
```

Generate reference data by flattening and normalizing the unmodulated resource grid data. Truncate the first cell in the same way as the modulated input data. Apply complex scaling to each demodulated sequence so that it can be compared to its corresponding reference data.

```matlab
refData1 = grid1(:);
refData1 = refData1(1:length(demodData1));
refData2 = grid2(:);

refData1 = refData1/norm(refData1);
refData2 = refData2/norm(refData2);

demodData1 = demodData1/(refData1'*demodData1);
demodData2 = demodData2/(refData2'*demodData2);
```

Compare the output of the Simulink model against the truncated input grid, and display the results.

```matlab
figure('units','normalized','outerposition',[0 0 1 1])
subplot(2,2,1)
plot(real(refData1(:)))
hold on
plot(squeeze(real(demodData1)))
legend('Input grid','Demodulated output')
title(sprintf('Cell 1 (NDLRB %d) - Real part', info1.NDLRB))
xlabel('OFDM Subcarriers')

subplot(2,2,2)
plot(imag(refData1(:)))
hold on
plot(squeeze(imag(demodData1)))
legend('Input grid','Demodulated output')
title(sprintf('Cell 1 (NDLRB %d) - Imaginary part', info1.NDLRB))
xlabel('OFDM Subcarriers')

subplot(2,2,3)
plot(real(refData2(:)))
hold on
plot(squeeze(real(demodData2)))
legend('Input grid','Demodulated output')
title(sprintf('Cell 2 (NDLRB %d) - Real part', info2.NDLRB))
xlabel('OFDM Subcarriers')

subplot(2,2,4)
plot(imag(refData2(:)))
hold on
plot(squeeze(imag(demodData2)))
legend('Input grid','Demodulated output')
title(sprintf('Cell 2 (NDLRB %d) - Imaginary part', info2.NDLRB))
xlabel('OFDM Subcarriers')

sqnrRealdB1 = 10*log10(var(real(demodData1))/abs(var(real(demodData1)) - var(real(refData1(:)))))
sqnrImagdB1 = 10*log10(var(imag(demodData1))/abs(var(imag(demodData1)) - var(imag(refData1(:)))))
```

```
fprintf('\n Cell 1: SQNR of real part is %.2f dB',sqnrRealdB1)
fprintf('\n Cell 1: SQNR of imaginary part is %.2f dB\n',sqnrImagdB1)

sqnrRealdB2 = 10*log10(var(real(demodData2))/abs(var(real(demodData2)) - var(real(refData2(:)))))
sqnrImagdB2 = 10*log10(var(imag(demodData2))/abs(var(imag(demodData2)) - var(imag(refData2(:)))))

fprintf('\n Cell 2: SQNR of real part is %.2f dB',sqnrRealdB2)
fprintf('\n Cell 2: SQNR of imaginary part is %.2f dB\n',sqnrImagdB2)
```

```
 Cell 1: SQNR of real part is 33.71 dB
 Cell 1: SQNR of imaginary part is 52.26 dB

 Cell 2: SQNR of real part is 32.41 dB
 Cell 2: SQNR of imaginary part is 36.72 dB
```
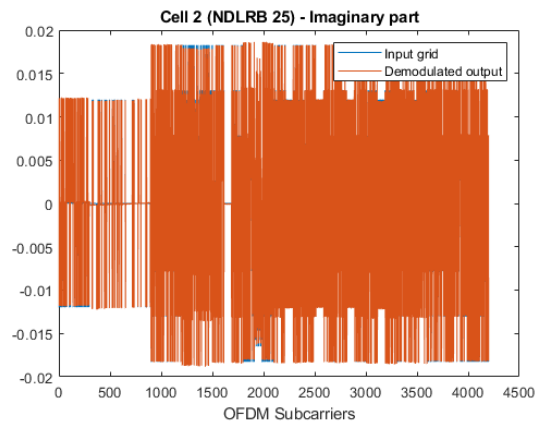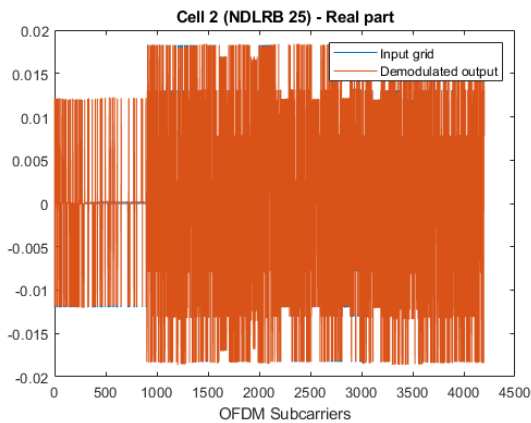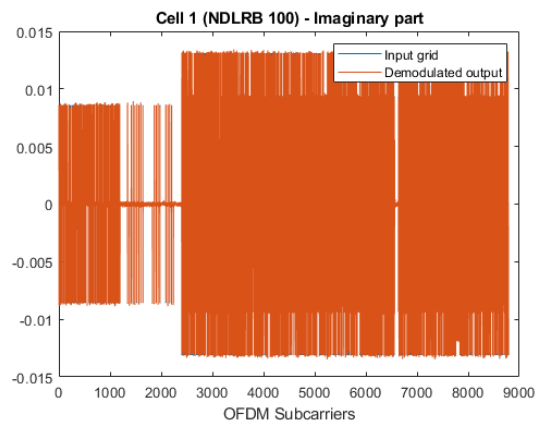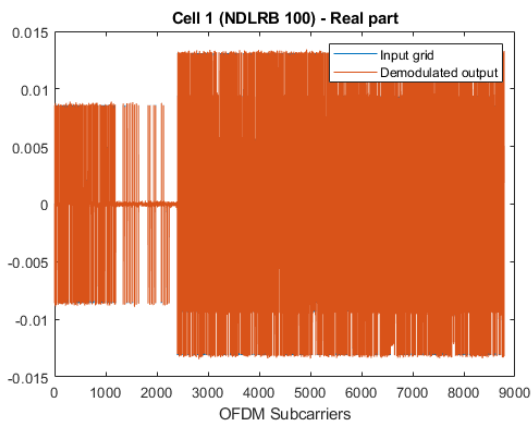

See Also

Blocks
LTE OFDM Demodulator

Modulate and Demodulate LTE Resource Grid

This example shows how to modulate and demodulate LTE resource grid samples. The model connects the LTE OFDM Modulator block to the LTE OFDM Demodulator block. To verify the algorithms of both the blocks, compare the output of the demodulator with the input of the modulator. You can generate HDL code from either block.

Generate the input resource grid using LTE Toolbox™.

```
enb = lteRMCDL('R.6');
enb.CyclicPrefix='Normal';
enb.TotSubframes = 1;

% ----------------------------------------------------------------
%      NDLRB                |    Sampling Rate (MHz)
% ----------------------------------------------------------------
%       6                   |    R.4
%       15                  |    R.5
%       25                  |    R.6
%       50                  |    R.7
%       75                  |    R.8
%       100                 |    R.9
% ----------------------------------------------------------------

[~,LTEGrid,info] = lteRMCDLTool(enb,[1;0;0;1]);

NDLRB=info.NDLRB;
if strcmp(enb.CyclicPrefix,'Normal')
    CPType=false;
else
    CPType=true;
end

sampling_time=1/30.72e6;
modulatorLatency=4137+2048*2;
demodulatorLatency=4137+2048*2;
stoptime=enb.TotSubframes*(30720+modulatorLatency+demodulatorLatency)*sampling_time;
```

Convert the LTEGrid sample frames to a stream of samples with control signals for input to the Simulink® model.

```
idlecyclesbetweensamples = 0;
idlecyclesbetweenframes  = 0;

[dataIn,ctrl] = whdlFramesToSamples(mat2cell(LTEGrid(:),numel(LTEGrid),1),...
                idlecyclesbetweensamples,idlecyclesbetweenframes);
validIn = logical(ctrl(:,3));
```

Run the Simulink model to modulate and demodulate the samples, and save the output samples to a workspace variable.

```
open_system('LTEHDLOFDMModDemodExample')
sim('LTEHDLOFDMModDemodExample');

rxgridSimulink = dataOut(validOut);
```
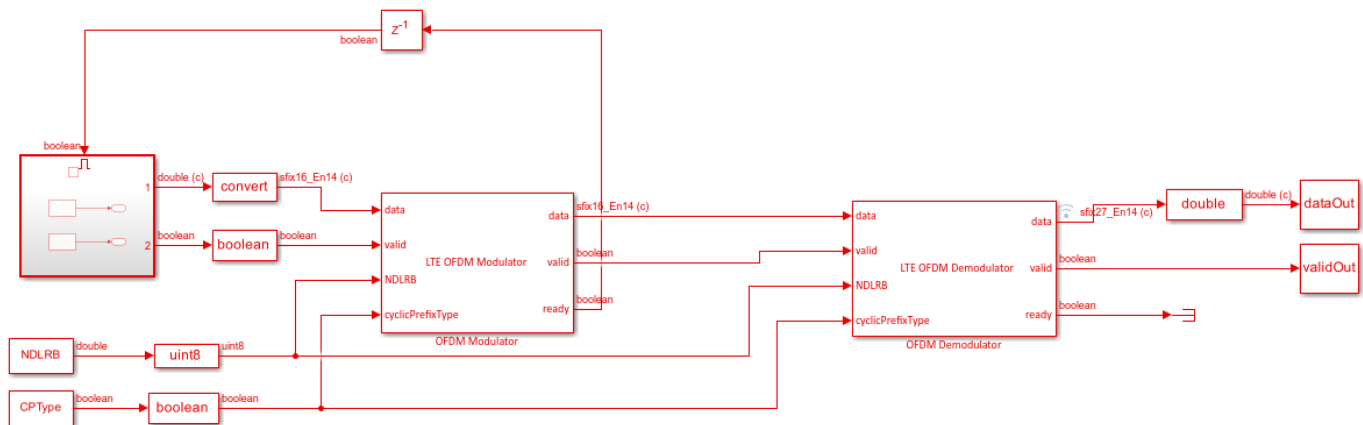
Copyright 2018 The MathWorks, Inc.

Compare the input of the modulator, generated from the `lteRMCDLTool` function, and the output of the demodulator from the model.

```
figure('units','normalized','outerposition',[0 0 1 1])
subplot(2,1,1);
plot(real(LTEGrid(:)));
hold on
plot(squeeze(real(rxgridSimulink)));
legend('Real part of LTE grid','Real part of demodulated waveform');
title('Comparision of Input to OFDM Modulator with Output from OFDM Demodulator');
xlabel('OFDM Subcarriers');
ylabel('Real part of the time-domain waveform');

subplot(2,1,2)
plot(imag(LTEGrid(:)))
hold on
plot(squeeze(imag(rxgridSimulink)))
legend('Imag part of LTE grid','Imag part of demodulated waveform');
title('Comparision of Input to OFDM Modulator with Output from OFDM Demodulator');
xlabel('OFDM Subcarriers');
ylabel('Imag part of the time-domain waveform');
```
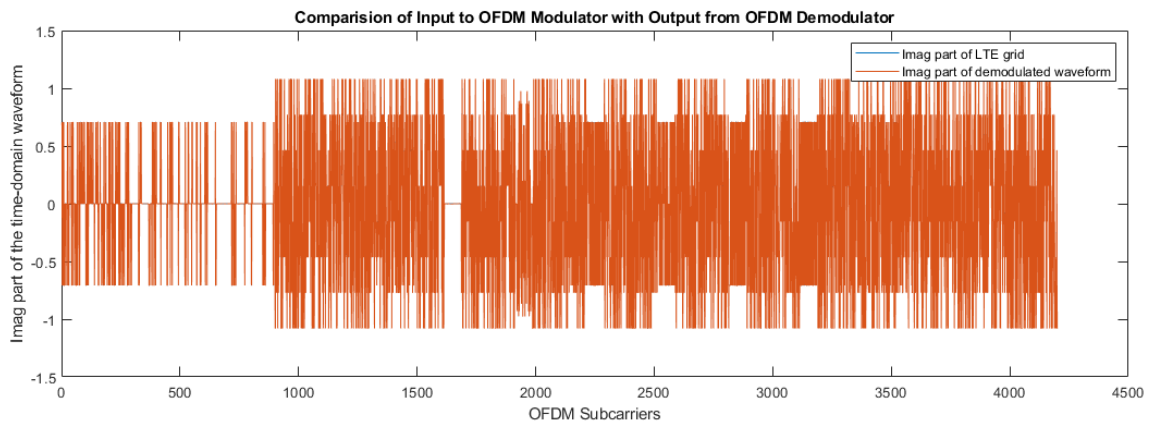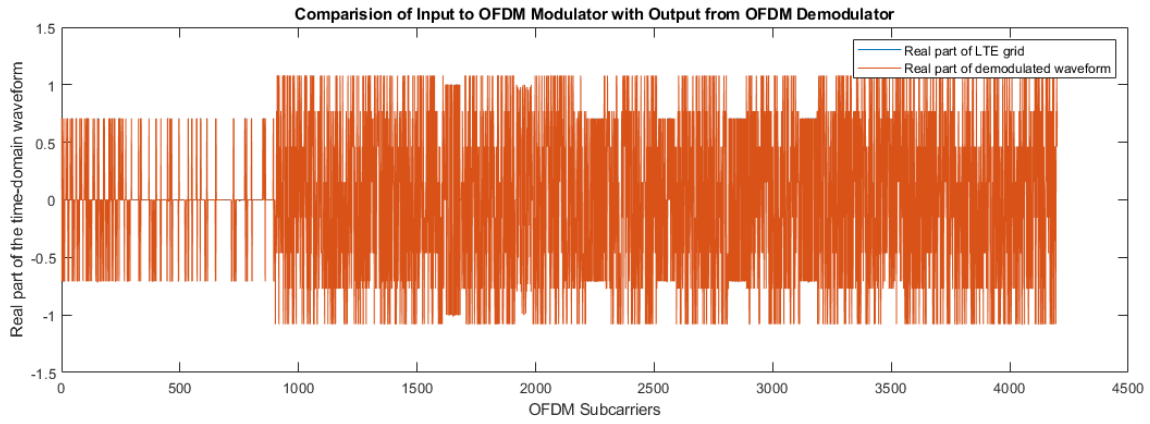
See Also

Blocks
LTE OFDM Demodulator | LTE OFDM Modulator

OFDM Modulation of LTE Resource Grid Samples

This example shows how to use the LTE OFDM Modulator block to modulate LTE resource grid samples to an equivalent time-domain signal output. You can generate HDL code from this block.

Generate the input resource grid using LTE Toolbox™ function.

```matlab
enb = lteRMCDL('R.6');
enb.CyclicPrefix='Normal';
enb.TotSubframes = 1;
% --------------------------------------------------------------
%      NDLRB              |    Sampling Rate (MHz)
% --------------------------------------------------------------
%      6                  |    R.4
%      15                 |    R.5
%      25                 |    R.6
%      50                 |    R.7
%      75                 |    R.8
%      100                |    R.9
% --------------------------------------------------------------

[~,LTEGrid,info] = lteRMCDLTool(enb,[1;0;0;1]);
[eNodeBOutput,~] = lteOFDMModulate(enb,LTEGrid);
```

Convert the `LTEGrid` sample frames to a stream of samples with control signals for input to the Simulink® model.

```matlab
NDLRB=info.NDLRB;
if strcmp(enb.CyclicPrefix,'Normal')
    CPType=false;
else
    CPType=true;
end

sampling_time=1/30.72e6;
stoptime=enb.TotSubframes*(30720+4137+2048*2)*sampling_time;

idlecyclesbetweensamples = 0;
idlecyclesbetweenframes  = 0;

[dataIn,ctrl] = whdlFramesToSamples(mat2cell(LTEGrid(:),numel(LTEGrid),1),...
    idlecyclesbetweensamples,idlecyclesbetweenframes);
validIn = logical(ctrl(:,3));
```

Run the Simulink model.

```matlab
modelname = 'OFDMModulatorModelExample';
open_system(modelname);
sim(modelname);
```
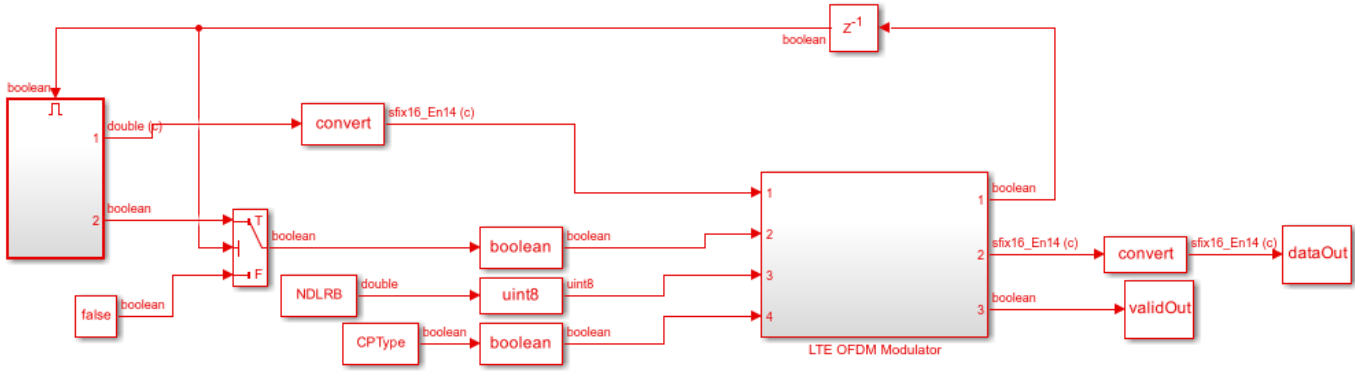
Copyright 2018 The MathWorks, Inc.

Save the output of the Simulink model and then compare the output of the model against the output of the `lteOFDMModulate` function.

```
rxgridSimulink=dataOut(validOut);

figure('units','normalized','outerposition',[0 0 1 1])
subplot(2,1,1);
plot(real(eNodeBOutput));
hold on
plot(squeeze(real(rxgridSimulink)));
legend('Real part of behavioral waveform','Real part of HDL-optimized waveform');
title('Comparison of LTE Time-Domain Downlink Waveforms from Behavioral and HDL-Optimized Algori
xlabel('OFDM subcarriers');
ylabel('Real part of the time-domain waveform');

subplot(2,1,2)
plot(imag(eNodeBOutput))
hold on
plot(squeeze(imag(rxgridSimulink)))
legend('Imag part of behavioral waveform','Imag part of HDL-optimized waveform');
title('Comparison of LTE Time-Domain Downlink Waveforms from Behavioral and HDL-Optimized Algori
xlabel('OFDM subcarriers');
ylabel('Imag part of the time-domain waveform');
```

Comparison of LTE Time-Domain Downlink Waveforms from Behavioral and HDL-Optimized Algorithms

Comparison of LTE Time-Domain Downlink Waveforms from Behavioral and HDL-Optimized Algorithms
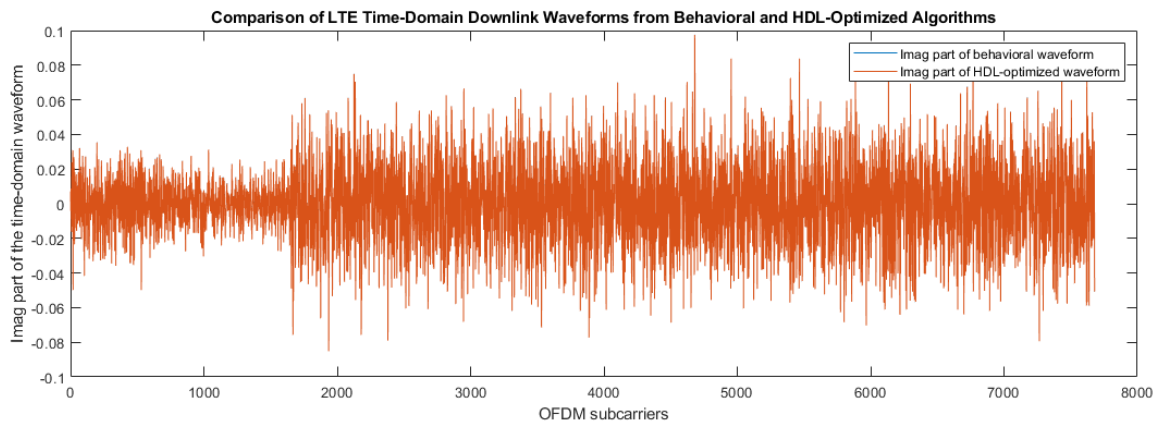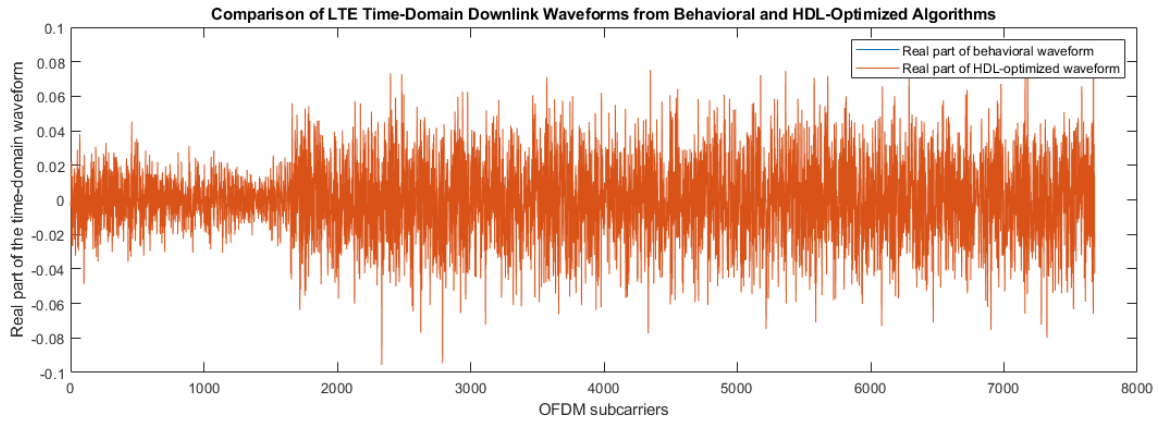
See Also

Blocks
LTE OFDM Modulator

Depuncture and Decode Streaming Samples

This example shows how to use the hardware-friendly Depuncturer block and Viterbi Decoder block to decode samples encoded at WLAN code rates.

Generate input samples in MATLAB® by encoding random data, BPSK-modulating the samples, applying a channel model, demodulating the samples, and creating received soft-decision bits. Then, import the soft-decision bits into a Simulink® model to depuncture and decode the samples. Export the result of the Simulink simulation back to MATLAB and compare it against the original input samples.

The example model supports HDL code generation for the HDL Depuncture and Decode subsystem.

```
modelname  = 'ltehdlViterbiDecoderModel';
open_system(modelname);
```
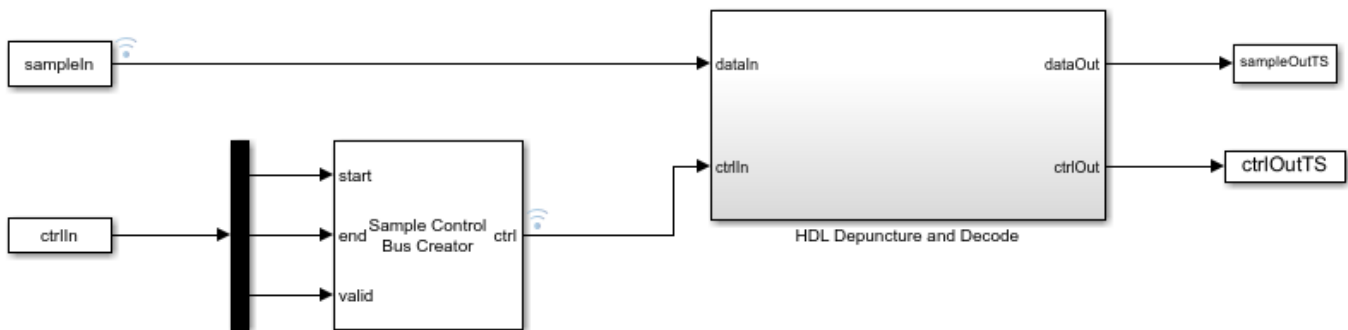

Copyright 2018 The MathWorks, Inc.

Set Up Code Rate Parameters

Set up workspace variables that describe the code rate. The Viterbi Decoder block supports constraint lengths in the range [3,9] and polynomial lengths in the range [2,7].

Choose a traceback depth in the range [3,128]. For non-punctured samples, the recommended depth is 5 times the *constraintLength*. For punctured samples, the recommended depth is 10 times the *constraintLength*.

Starting from a code rate of 1/2, IEEE 802.11 WLAN specifies three puncturing patterns to generate three additional code rates. Choose one of these code rates, and then set the frame size and puncturing pattern based on that rate. You can also choose the unpunctured code rate of 1/2.

IEEE 802.11 WLAN specifies different modulation types for different code rates and uses `'Terminated'` mode. This example uses BPSK modulation for all rates and can run with `'Terminated'` or `'Truncated'` operation mode. The blocks also support `'Continuous'` mode, but it is not included in this example.

```
constraintLength = 7;
codeGenerator = [133 171];
opMode = 'Terminated';
tracebackDepth = 10*constraintLength;
```

```matlab
trellis = poly2trellis(constraintLength,...
    codeGenerator);

% IEEE 802.11n-2009 WLAN 1/2 (7, [133 171])
% Rate    Puncture Pattern      Maximum Frame Size
% 1/2      [1;1;1;1]                    2592
% 2/3      [1;1;1;0]                    1728
% 3/4      [1;1;1;0;0;1]                1944
% 5/6      [1;1;1;0;0;1;1;0;0;1]    2160
codeRate = 3/4;

if (codeRate == 2/3)
    puncVector = logical([1;1;1;0]);
    frameSize = 1728;
elseif (codeRate == 3/4)
    puncVector = logical([1;1;1;0;0;1]);
    frameSize = 1944;
elseif (codeRate == 5/6)
    puncVector = logical([1;1;1;0;0;1;1;0;0;1]);
    frameSize = 2160;
else % codeRate == 1/2
    puncVector = logical([1;1;1;1]);
    frameSize = 2592;
end

if strcmpi(opMode,'Terminated')
    % Terminate the state at the end of the frame
    tailLen = constraintLength-1;
else
    % Truncated mode
    tailLen = 0;
end
```

Generate Samples for Decoding

Use Communications Toolbox™ functions and System objects to generate encoded samples and apply channel noise. Demodulate the received samples, and create soft-decision values for each sample.

```matlab
EbNo = 10;
EcNo = EbNo - 10*log10(numel(codeGenerator));

numFrames = 5;
numSoftBits = 4;

txMessages = cell(1,numFrames);
rxSoftMessages = cell(1,numFrames);

No = 10^((-EcNo)/10);
quantStepSize = sqrt(No/2^numSoftBits);

modulator = comm.BPSKModulator;
channel = comm.AWGNChannel('EbNo',EcNo);
demodulator = comm.BPSKDemodulator('DecisionMethod','Log-likelihood ratio');

for ii = 1:numFrames
    txMessages{ii} = [randn(frameSize - tailLen,1)
        zeros(tailLen,1)]>0;
    % Convolutional encoding and puncturing
```

```
    txCodeword = convenc(txMessages{ii},trellis,puncVector);
    % Modulation
    modOut = modulator.step(txCodeword);
    % Channel
    chanOut = channel.step(modOut);
    % Demodulation
    demodOut = -demodulator.step(chanOut)/4;
    % Convert to soft-decision values
    rxSoftMessagesDouble = demodOut./quantStepSize;
    rxSoftMessages{ii} = fi(rxSoftMessagesDouble,1,numSoftBits,0);
end
```

Set Up Variables for Simulink Simulation

The Simulink model requires streaming samples with accompanying control signals. Use the `whdlFramesToSamples` function to convert the framed `rxSoftMessages` to streaming samples and generate the matching control signals.

Calculate the required simulation time from the latency of the depuncture and decoder blocks.

```
samplesizeIn = 1;
idlecyclesbetweensamples = 0;
idlecyclesbetweenframes = 0;
if strcmpi(opMode,'Truncated')
    % Truncated mode requires a gap between frames of at least constraintLength-1
    idlecyclesbetweenframes = constraintLength - 1;
end

[sampleIn,ctrlIn] = whdlFramesToSamples(rxSoftMessages, ...
    idlecyclesbetweensamples,idlecyclesbetweenframes,samplesizeIn);

depunLatency = 6;
vitLatency = 4*tracebackDepth + constraintLength + 13;
latency = vitLatency + depunLatency;

simTime = size(ctrlIn,1) + latency;
sampletime = 1;
```

Run the Simulink Model

Call the Simulink model to depuncture and decode the samples. The model exports the decoded samples to the MATLAB workspace. The Depuncture and Viterbi Decoder block parameters are configured using workspace variables. Because **Operation mode** is a list parameter, use `set_param` to assign the workspace value.

Convert the streaming samples back to framed data for comparison.

```
set_param([modelname '/HDL Depuncture and Decode'],'Open','on');
set_param([modelname '/HDL Depuncture and Decode/Viterbi Decoder'],...
        'TerminationMethod',opMode);
sim(modelname);

sampleOut = squeeze(sampleOutTS.Data);
ctrlOut = [squeeze(ctrlOutTS.start.Data) ...
    squeeze(ctrlOutTS.end.Data) ...
    squeeze(ctrlOutTS.valid.Data)];
rxMessages = whdlSamplesToFrames(sampleOut,ctrlOut);
```

```
Maximum frame size computed to be 1944 samples.
```
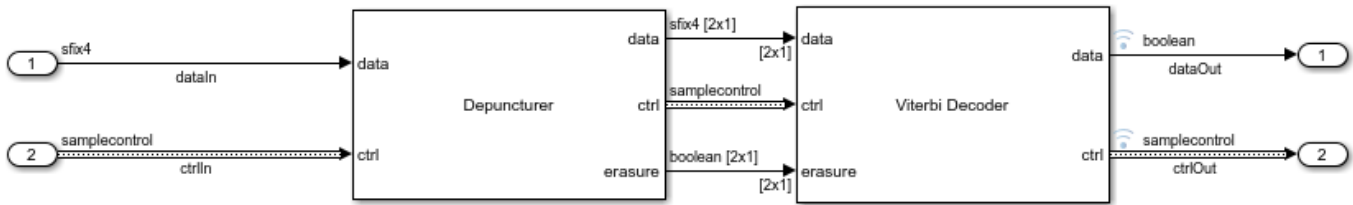

Verify Results

Compare the output samples against the generated input samples.

```
fprintf('\nDecoded Samples\n');
for ii = 1:numFrames
    numBitsErr = sum(xor(txMessages{ii},rxMessages{ii}));
    fprintf('Frame #%d: %d bits mismatch \n',ii,numBitsErr);
end
```

```
Decoded Samples
Frame #1: 0 bits mismatch
Frame #2: 0 bits mismatch
Frame #3: 0 bits mismatch
Frame #4: 0 bits mismatch
Frame #5: 0 bits mismatch
```

See Also

Blocks
Depuncturer | Viterbi Decoder

LTE Symbol Modulation of Data Bits

This example shows how to use the LTE Symbol Modulator block to modulate data bits to complex data symbols. You can generate HDL code from this block.

Set up input data parameters. Choose a data length for each modulation type. The data length must be an integer multiple of number of bits per symbol.

```
rng(0);
framesize = 240;

% Map modulation names to values
% 0 - BPSK
% 1 - QPSK
% 2 - 16-QAM
% 3 - 64-QAM
% 4 - 256-QAM
% others - QPSK

% For LTE Symbol Modulator Simulink block
modSelVal = [0;1;2;3;4];

% For |lteSymbolModulate| function
modSelStr = {'BPSK','QPSK','16QAM','64QAM','256QAM'};

outWordLength = 16;
numframes = length(modSelVal);
dataBits  = cell(1,numframes);
modSelTmp = cell(1,numframes);
lteFcnOutput = cell(1,numframes);
```

Generate frames of random input samples.

```
for ii = 1:numframes
    dataBits{ii} = logical(randi([0 1],framesize,1));
    modSelTmp{ii} = fi(modSelVal(ii)*ones(framesize,1),0,3,0);

end
```

Convert the framed input data to a stream of samples and input the stream to the LTE Symbol Modulator Simulink block.

```
idlecyclesbetweensamples = 0;
idlecyclesbetweenframes  = 0;
[sampleIn, ctrl] = whdlFramesToSamples(dataBits,idlecyclesbetweensamples,...
    idlecyclesbetweenframes);
[modSel, ~] = whdlFramesToSamples(modSelTmp,idlecyclesbetweensamples,...
    idlecyclesbetweenframes);
load = logical(ctrl(:,1)');
validIn = logical(ctrl(:,3)');

sampletime = 1;
samplesizeIn = 1;
simTime = size(ctrl,1);
```

Run the Simulink model.

```
modelname = 'ltehdlSymbolModulatorModel';
open_system(modelname);
sim(modelname);
```
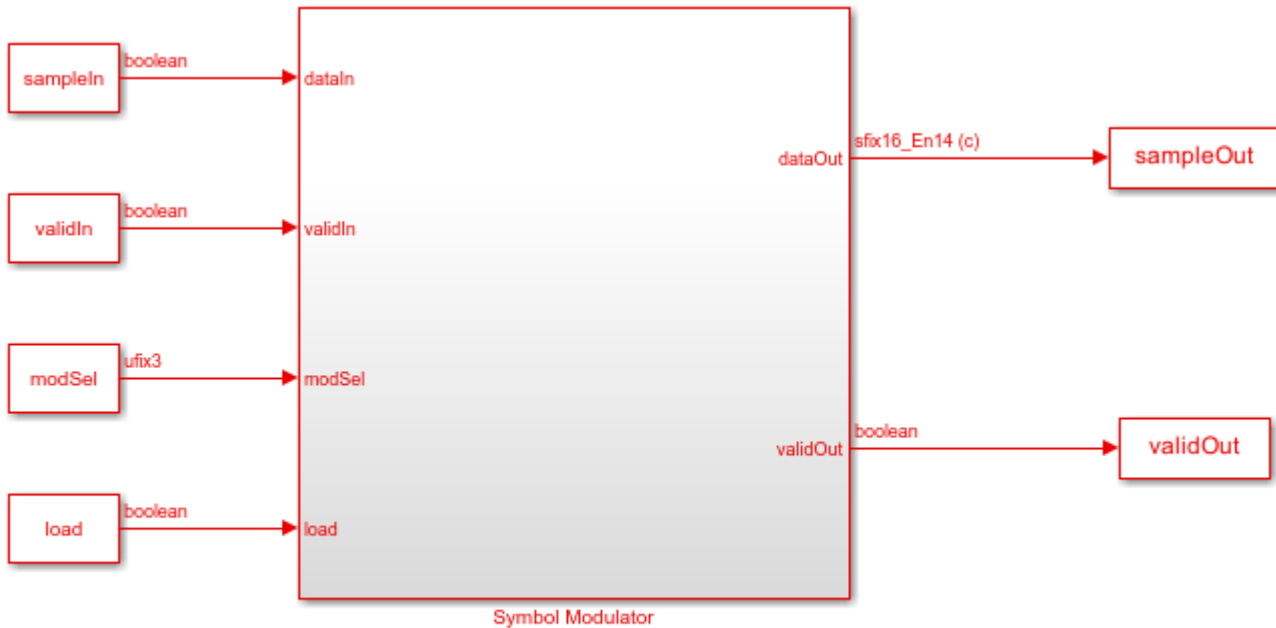

Copyright 2018 The MathWorks, Inc.

Export the stream of modulated samples from Simulink to the MATLAB workspace.

```
sampleOut = squeeze(sampleOut).';
lteHDLOutput = sampleOut(squeeze(validOut));
```

Modulate data bits with `lteSymbolModulate` function and use its output as a reference data.

```
for ii = 1:numframes
    lteFcnOutput{ii} = lteSymbolModulate(dataBits{ii},modSelStr{ii}).';
end
```

Compare the output of the Simulink model against the output of `lteSymbolModulate` function.

```
fprintf('\nLTE Symbol Modulator\n');
lteFcnOutput = fi(cell2mat(lteFcnOutput),1,outWordLength,outWordLength-2);
difference = sum(abs(lteHDLOutput-lteFcnOutput(1:length(lteHDLOutput))));
fprintf('\nTotal number of samples differed between Simulink block output and Reference data outp
```

```
LTE Symbol Modulator
```

Total number of samples differed between Simulink block output and Reference data output: 0

See Also

Blocks
LTE Symbol Modulator

NR Symbol Modulation of Data Bits

This example shows how to use the NR Symbol Modulator block to modulate data bits to complex data symbols. You can generate HDL code from this block.

Set up input data parameters. Choose a data length for each modulation type. The data length must be an integer multiple of number of bits per symbol.

```
rng(0);
framesize = 240;

% Map modulation names to values
% 0 - BPSK
% 1 - QPSK
% 2 - 16-QAM
% 3 - 64-QAM
% 4 - 256-QAM
% 5 - pi/2-BPSK
% others - QPSK

% for NR Symbol Modulator Simulink block
modSelVal = [0;1;2;3;4;5];

% for nrSymbolModulate function
modSelStr = {'BPSK','QPSK','16QAM','64QAM','256QAM','pi/2-BPSk'};

outWordLength = 16;
numframes = length(modSelVal);
dataBits  = cell(1,numframes);
modSelTmp = cell(1,numframes);
nrFcnOutput = cell(1,numframes);
```

Generate frames of random input samples.

```
for ii = 1:numframes
    dataBits{ii} = logical(randi([0 1],framesize,1));
    modSelTmp{ii} = fi(modSelVal(ii)*ones(framesize,1),0,3,0);

end
```

Convert the framed input data to a stream of samples and input the stream to the Simulink block.

```
idlecyclesbetweensamples = 0;
idlecyclesbetweenframes  = 0;
[sampleIn, ctrl] = whdlFramesToSamples(dataBits,idlecyclesbetweensamples,...
    idlecyclesbetweenframes);
[modSel, ~] = whdlFramesToSamples(modSelTmp,idlecyclesbetweensamples,...
    idlecyclesbetweenframes);
load = logical(ctrl(:,1)');
validIn = logical(ctrl(:,3)');

sampletime = 1;
samplesizeIn = 1;
simTime = size(ctrl,1);
```

Run the Simulink model.

```
modelname = 'nrhdlSymbolModulatorModel';
open_system(modelname);
sim(modelname);
```
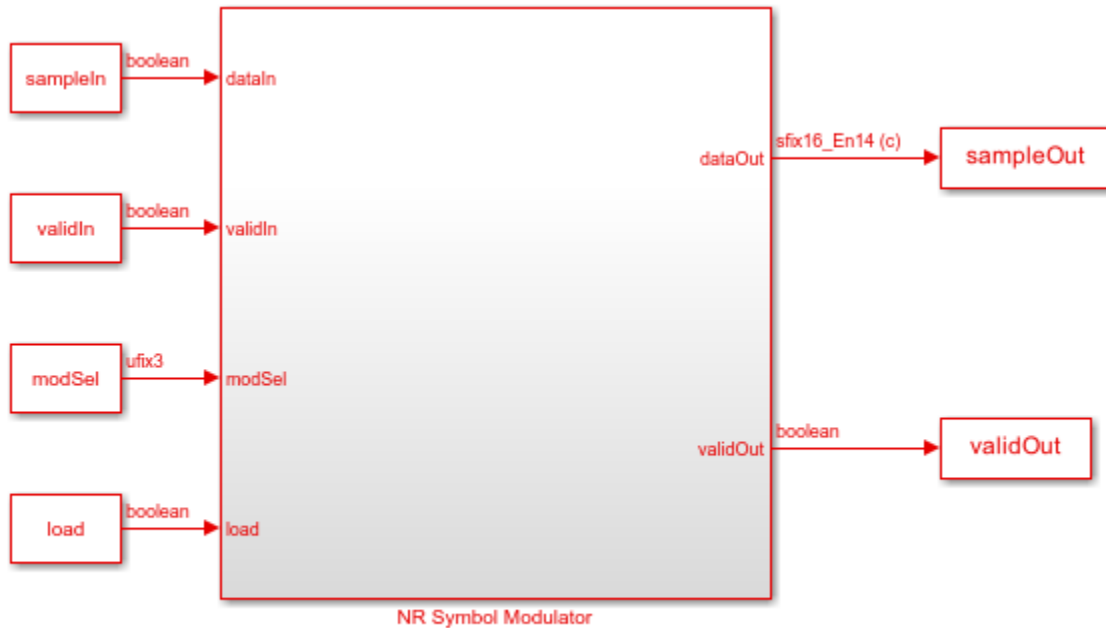

NR Symbol Modulator

Copyright 2018 The MathWorks, Inc.

Export the stream of modulated samples from Simulink to the MATLAB workspace.

```
sampleOut = squeeze(sampleOut).';
nrHDLOutput = sampleOut(squeeze(validOut));
```

Modulate frame data bits with nrSymbolModulate function and use the output of this function as a reference data.

```
for ii = 1:numframes
    nrFcnOutput{ii} = nrSymbolModulate(dataBits{ii},modSelStr{ii}).';
end
```

Compare the output of the Simulink model against the output of nrSymbolModulate function.

```
fprintf('\nNR Symbol Modulator\n');
nrFcnOutput = fi(cell2mat(nrFcnOutput),1,outWordLength,outWordLength-2);
error = sum(abs(nrHDLOutput-nrFcnOutput(1:length(nrHDLOutput))));
fprintf('\nTotal number of samples differed between Behavioral and HDL simulation: %d \n',error)
```

NR Symbol Modulator

```
Total number of samples differed between Behavioral and HDL simulation: 0
```

See Also

Blocks
NR Symbol Modulator

LTE Symbol Demodulation of Complex Data Symbols

This example shows how to use the LTE Symbol Demodulator block to demodulate complex LTE data symbols to data bits or LLR values. The workflow follows these steps:

1   Set up input data parameters.

2   Generate frames of random input samples.

3   Convert framed input data to a stream of samples and import the stream into Simulink®.

4   Run the Simulink® model, which contains the LTE Symbol Demodulator block.

5   Export the stream of demodulated samples from Simulink to the MATLAB® workspace.

6   Demodulate data symbols with `lteSymbolDemodulate` function to use its output as a reference data.

7   Compare Simulink block output data with the reference MATLAB function output.

Set up input data parameters.

Map modulation names to values. The numerical values are used to set up the LTE Symbol Demodulator block. The strings are used to configure the `lteSymbolDemodulator` function.

```
rng(0);
framesize = 10;

% 0 - BPSK
% 1 - QPSK
% 2 - 16-QAM
% 3 - 64-QAM
% 4 - 256-QAM
% others - QPSK
modSelVal = [0;1;2;3;4];
modSelStr = {'BPSK','QPSK','16QAM','64QAM','256QAM'};

decType = 'Soft';

numframes = length(modSelVal);
dataSymbols  = cell(1,numframes);
modSelTmp = cell(1,numframes);
lteFcnOutput = cell(1,numframes);
```

Generate frames of random input samples.

```
for ii = 1:numframes
    dataSymbols{ii} = complex(randn(framesize,1),randn(framesize,1));
    modSelTmp{ii} = fi(modSelVal(ii)*ones(framesize,1),0,3,0);
end
```

Convert the framed input data to a stream of samples and input the stream to the LTE Symbol Demodulator Simulink block.

```
idlecyclesbetweensamples = 0;
idlecyclesbetweenframes  = 0;
[sampleIn, ctrl] = whdlFramesToSamples(dataSymbols,idlecyclesbetweensamples,...
    idlecyclesbetweenframes);
[modSel, ~] = whdlFramesToSamples(modSelTmp,idlecyclesbetweensamples,...
```

```
        idlecyclesbetweenframes);
validIn = logical(ctrl(:,3)');

sampletime = 1;
samplesizeIn = 1;
simTime = size(ctrl,1)*8;
```

Run the Simulink model.

```
modelname = 'ltehdlSymbolDemodulatorModel';
open_system(modelname);
set_param([modelname '/Demod/LTE Symbol Demodulator'],'DecisionType',decType)
sim(modelname);
```
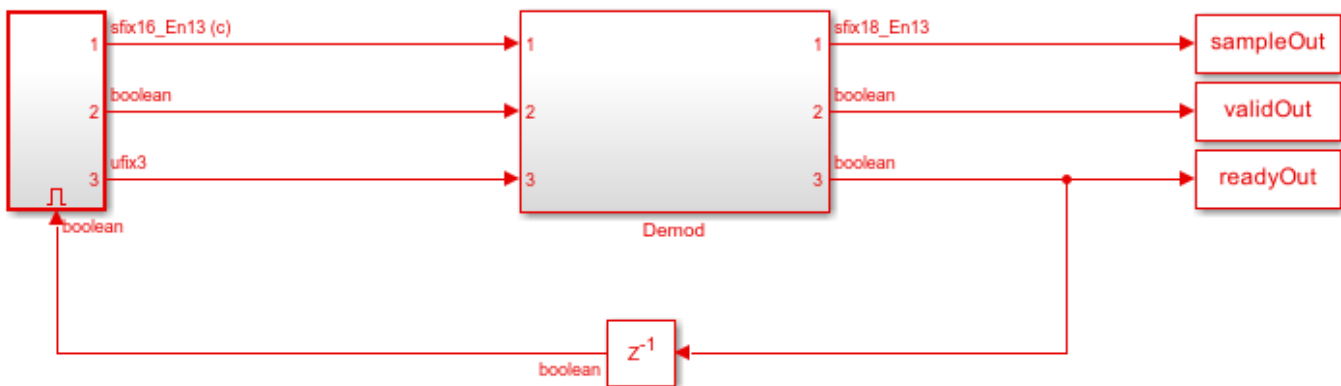

Copyright 2019 The MathWorks, Inc.

Export the stream of demodulated samples from Simulink to the MATLAB workspace.

```
lteHDLOutput = sampleOut(validOut).';
```

Demodulate data symbols with `lteSymbolDemodulate` function and use its output as a reference data.

```
for ii = 1:numframes
 lteFcnOutput{ii} = lteSymbolDemodulate(dataSymbols{ii},modSelStr{ii},decType).';
end
```

Compare the output of the Simulink model against the output of `lteSymbolDemodulate` function.

```
lteFcnOutput = double(cell2mat(lteFcnOutput));

figure(1)
stem(lteHDLOutput,'b')
hold on
stem(lteFcnOutput,'--r')
grid on
legend('Reference','Simulink')
xlabel('Sample Index')
ylabel('Magnitude')
title('Comparison of Simulink Block and MATLAB Function')
```
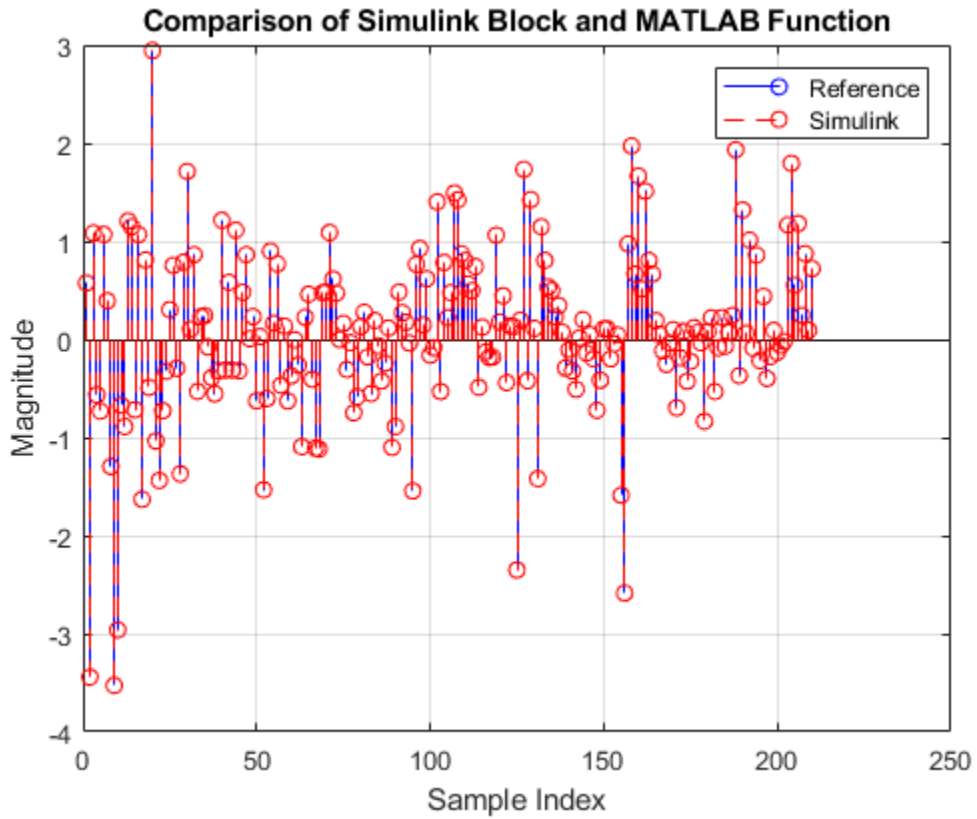
See Also

Blocks
LTE Symbol Demodulator

NR Symbol Demodulation of Complex Data Symbols

This example shows how to use the NR Symbol Demodulator block to demodulate complex NR data symbols to data bits or LLR values. The workflow follows these steps:

1   Set up input data parameters.
2   Generate frames of random input samples.
3   Convert framed input data to a stream of samples and import the stream into Simulink.
4   Run the Simulink® model, which contains the NR Symbol Demodulator block.
5   Export the stream of demodulated samples from Simulink to the MATLAB® workspace.
6   Demodulate data symbols with `nrSymbolDemodulate` function to use its output as a reference data.
7   Compare Simulink block output data with the reference MATLAB function output.

Set up input data parameters.

Map modulation names to values. The numerical values are used to set up the NR Symbol Demodulator block. The strings are used to configure the `nrSymbolDemodulator` function.

```
rng(0);
framesize = 10;

% 0 - BPSK
% 1 - QPSK
% 2 - 16-QAM
% 3 - 64-QAM
% 4 - 256-QAM
% 5 - pi/2-BPSK
% others - QPSK
modSelVal = [0;1;2;3;4;5];
modSelStr = {'BPSK','QPSK','16QAM','64QAM','256QAM','pi/2-BPSK'};

decType = 'Soft';

numframes = length(modSelVal);
dataSymbols  = cell(1,numframes);
modSelTmp = cell(1,numframes);
nrFcnOutput = cell(1,numframes);
```

Generate frames of random input samples.

```
for ii = 1:numframes
    dataSymbols{ii} = complex(randn(framesize,1),randn(framesize,1));
    modSelTmp{ii} = fi(modSelVal(ii)*ones(framesize,1),0,3,0);
end
```

Convert the framed input data to a stream of samples and input the stream to the NR Symbol Demodulator Simulink block.

```
idlecyclesbetweensamples = 0;
idlecyclesbetweenframes  = 0;
[sampleIn, ctrl] = whdlFramesToSamples(dataSymbols,idlecyclesbetweensamples,...
    idlecyclesbetweenframes);
```

```
[modSel, ~] = whdlFramesToSamples(modSelTmp,idlecyclesbetweensamples,...
    idlecyclesbetweenframes);
validIn = logical(ctrl(:,3)');

sampletime = 1;
samplesizeIn = 1;
simTime = size(ctrl,1)*8;
```

Run the Simulink model.

```
modelname = 'nrhdlSymbolDemodulatorModel';
open_system(modelname);
set_param([modelname '/NRDemod/NR Symbol Demodulator'],'DecisionType',decType)
sim(modelname);
```
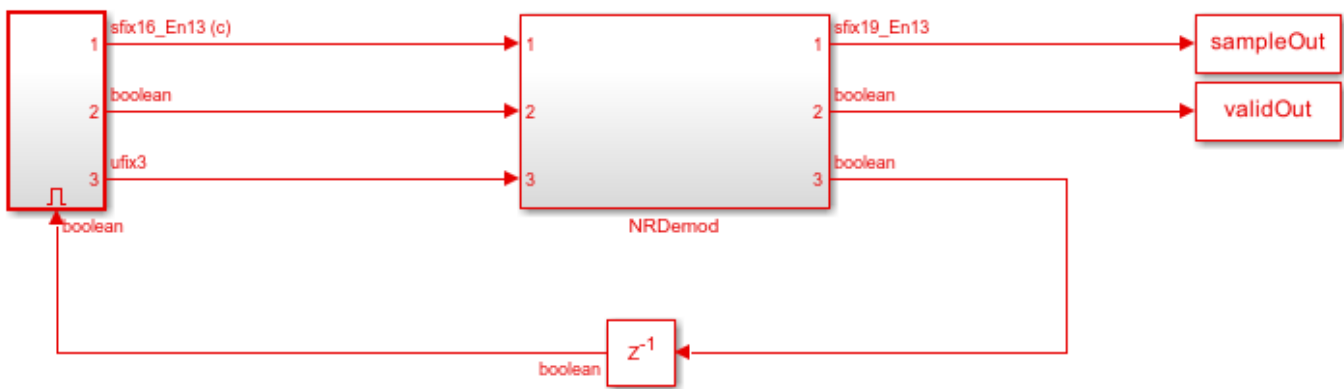

Copyright 2019 The MathWorks, Inc.

Export the stream of demodulated samples from Simulink to the MATLAB workspace.

```
nrHDLOutput = sampleOut(validOut).';
```

Demodulate data symbols with `nrSymbolDemodulate` function and use its output as a reference data.

```
for ii = 1:numframes
 nrFcnOutput{ii} = nrSymbolDemodulate(dataSymbols{ii},modSelStr{ii},'DecisionType',decType,1).';
end
```

Compare the output of the Simulink model against the output of `nrSymbolDemodulate` function.

```
nrFcnOutput = double(cell2mat(nrFcnOutput));

figure(1)
stem(nrHDLOutput,'b')
hold on
stem(nrFcnOutput,'--r')
grid on
legend('Reference','Simulink')
xlabel('Sample Index')
ylabel('Magnitude')
title('Comparison of Simulink block and MATLAB function')
```
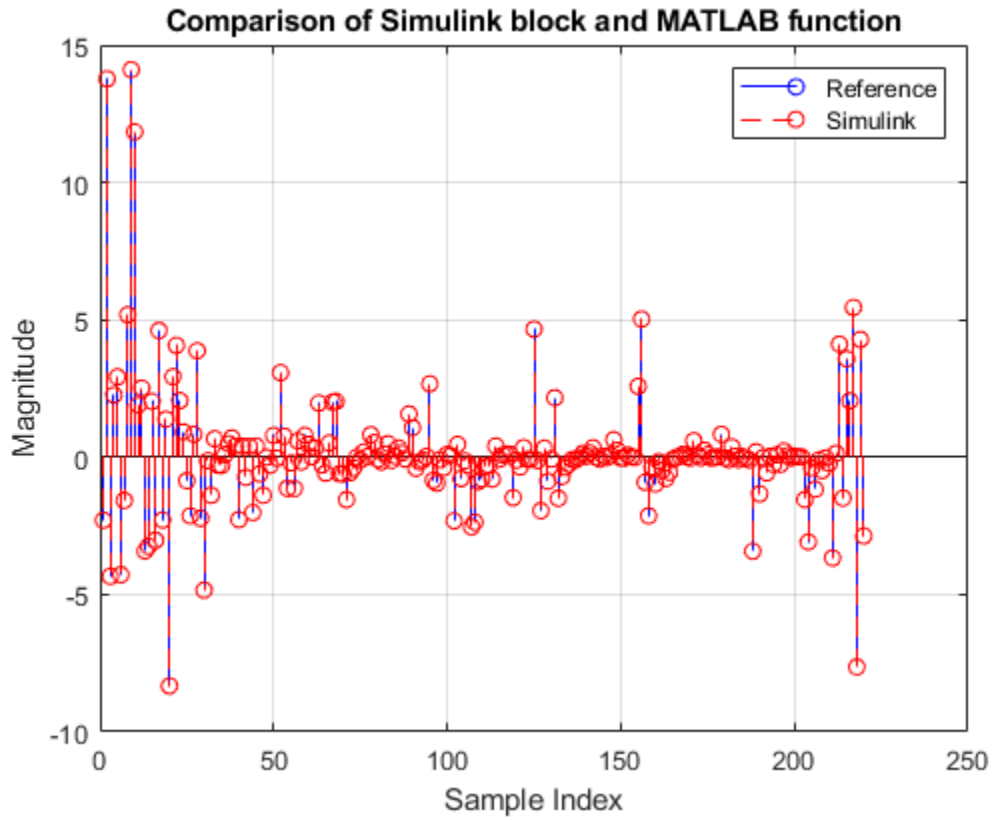
See Also

Blocks
NR Symbol Demodulator

Application of FFT 1536 block in LTE OFDM Demodulation

This example shows how to use the FFT 1536 block in LTE OFDM demodulation.

1   Generate transmitter waveform.

2   Remove cyclic prefix.

3   Prepare inputs for FFT 1536 simulation.

4   Form resource grid.

5   Compare the CellRS symbols from the grid with that of `lteCellRS` function.

6   Generate HDL code.

Generate transmitter waveform.

```
cfg = lteTestModel('1.1','15MHz');
cfg.TotSubframes = 1;
tx = lteTestModelTool(cfg);
```

The above transmitter waveform generation uses a 2048-point FFT, which results in a scaling factor of $\frac{1}{2048}$ in OFDM modulation. If a 1536-point FFT were used, the waveform would have a scaling factor of $\frac{1}{1536}$. This example multiplies the waveform by a factor of $\frac{2048}{1536}$ to achieve the correct scaling.

```
tx = tx*(2048/1536);
```

To achieve a 23.04 Msps sampling rate, resample the `tx` samples by $\frac{3}{4} = \frac{23.04e6}{30.72e6}$

```
rx = resample(tx,3,4); % rate conversion from 30.72Msps to 23.04Msps
```

Remove cyclic prefix. The first symbol of each slot has 12 additional CP samples.

```
rx(11520+1:11520+12) = []; % discard 12 CP samples in slot 2
rx(1:12) = []; % discard 12 CP samples in slot 1
rx = reshape(rx,108+1536,14); % reshape to form 14 OFDM symbols
rx(1:108,:) = []; % discard remaining 108 CP samples from all symbols
```

Prepare inputs for FFT 1536 simulation.

```
SampleTime = 4.3e-8; % 1/23.04e6;
data = rx(:);
valid = true(1536*14,1);
data = fi(data,1,22,20);

dataIn = timeseries(data,(0:length(data)-1).'*SampleTime);
validIn = timeseries(valid,(0:length(valid)-1).'*SampleTime);

FFT1536Latency = 3180;

NofClks = FFT1536Latency+length(data); % number of simulation clock cycles
StopTime = (NofClks)*SampleTime;

open_system HDLFFT1536model;
sim HDLFFT1536model;
```
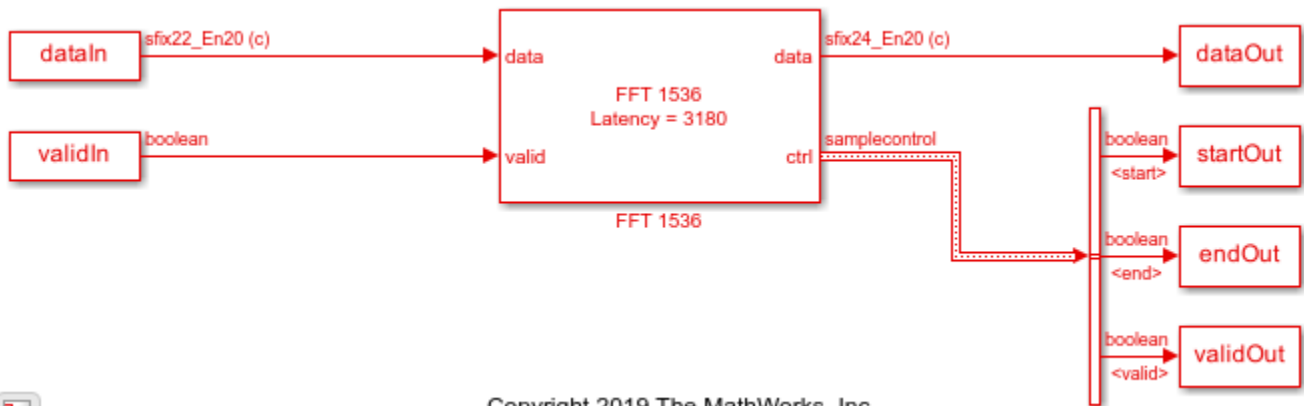
Copyright 2019 The MathWorks, Inc.

```
simOut = dataOut(validOut);
simOut = double(simOut(:)*1536);
```

Form the resource grid and remove the DC subcarrier.

```
fftOut = fftshift(reshape(simOut,1536,14));
resourceGrid = fftOut(318+1:318+1+900,:);
resourceGrid(900/2+1,:) = [];
```

Compare the CellRS symbols from the grid with the symbols returned from the `lteCellRS` function.

```
cellRS = lteCellRS(cfg);
cellRSIndices = lteCellRSIndices(cfg);
simCellRS = resourceGrid(cellRSIndices);
figure;
plot(real(simCellRS),imag(simCellRS),'o','MarkerSize',15);
hold on;
plot(real(cellRS),imag(cellRS),'*','MarkerSize',10)
legend('CellRS symbols from the FFT 1536 simulation grid'...
    ,'CellRS symbols from lteCellRS function','Location','southoutside')
axis([-1 1 -1 1]);
```
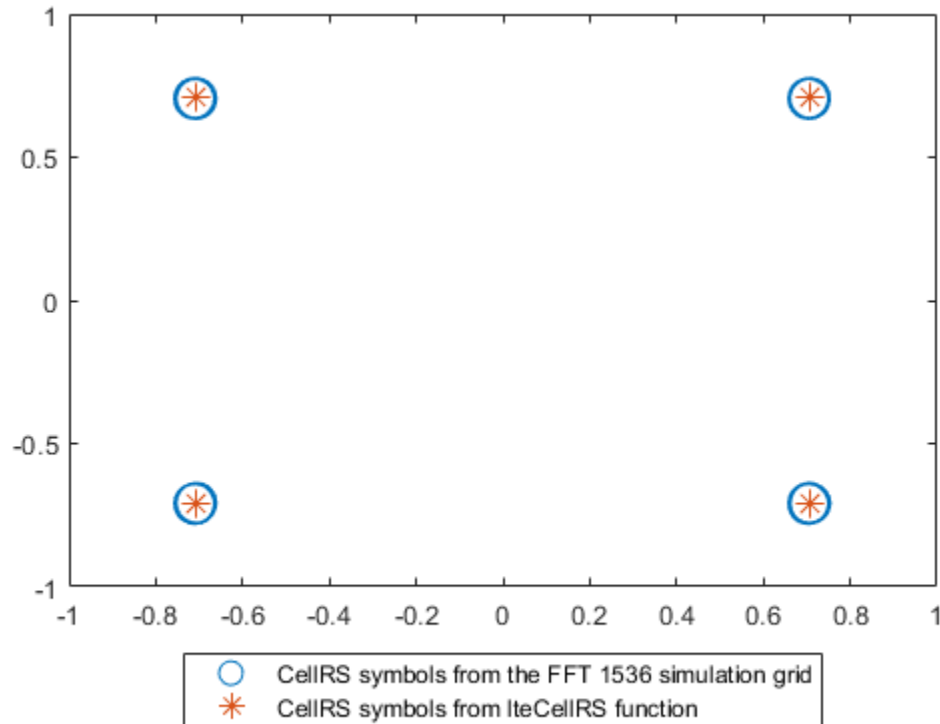
To generate HDL code for the FFT 1536 block, you must have an HDL Coder™ license. To generate HDL code from the FFT 1536 block in this model, right-click the block and select Create Subsystem from Selection. Then right-click the subsystem and select HDL Code > Generate HDL Code for Subsystem.

See Also

Blocks
FFT 1536

Convolutional Encode and Puncture Streaming Samples

This example shows how to use the hardware-friendly Convolutional Encoder and Puncturer blocks to encode samples at WLAN code rates.

1. Generate random input frame samples with frame control signals by using the `whdlFramesToSamples` function in MATLAB®.
2. Import these samples into a Simulink® model and run the model to encode and puncture the samples.
3. Export the result of the Simulink simulation back to MATLAB.
4. Generate reference samples using the `convenc` MATLAB function with puncturing enabled.
5. Compare the Simulink results with the reference samples.

The example model supports HDL code generation for the `EncodeAndPuncture` subsystem, that contains the Convolutional Encoder and Puncturer blocks.

```
modelname  = 'GenConvEncPuncturerModel';
open_system(modelname);
```


Copyright 2019 The MathWorks, Inc.

Set up workspace variables that describe the code rate. The Convolutional Encoder block supports constraint lengths in the range [3,9] and polynomial lengths in the range [2,7].

Starting from a code rate of 1/2, IEEE 802.11 WLAN specifies three puncturing patterns to generate three additional code rates. Choose one of these code rates, and then set the frame size and puncturing pattern based on that code rate. You can also choose the unpunctured code rate of 1/2.

IEEE 802.11 WLAN specifies different code rates and uses `'Terminated'` mode. The blocks also support `'Continuous'` mode and `'Truncated'` modes, but they are not included in this example.

```
constraintLength = 7;
codeGenerator = [133 171];

trellis = poly2trellis(constraintLength,...
    codeGenerator);

% IEEE 802.11n-2009 WLAN 1/2 (7, [133 171])
```

```
% Rate    Puncture Pattern       Maximum Frame Size
% 1/2     [1;1;1;1]                      2592
% 2/3     [1;1;1;0]                      1728
% 3/4     [1;1;1;0;0;1]                  1944
% 5/6     [1;1;1;0;0;1;1;0;0;1]    2160
codeRate = 3/4;
if (codeRate == 2/3)
    puncVector = logical([1;1;1;0]);
    frameSize = 1728;
elseif (codeRate == 3/4)
    puncVector = logical([1;1;1;0;0;1]);
    frameSize = 1944;
elseif (codeRate == 5/6)
    puncVector = logical([1;1;1;0;0;1;1;0;0;1]);
    frameSize = 2160;
else % codeRate == 1/2
    puncVector = logical([1;1;1;1]);
    frameSize = 2592;
end
```

Generate input frame samples for encoding and puncturing by using Communications Toolbox™ System objects to generate encoded samples.

```
numFrames = 5;

txMessages = cell(1,numFrames);
txCodeword = cell(1,numFrames);

for ii = 1:numFrames
    txMessages{ii} = logical(randn(frameSize-constraintLength+1,1));
end
```

Set up variables for Simulink simulation. The Simulink model requires streaming samples with accompanying control signals. Calculate the required simulation time from the latency of the Convolutional Encoder and Puncturer blocks.

```
samplesizeIn = 1;
idlecyclesbetweensamples = 0;
idlecyclesbetweenframes = constraintLength-1;
[sampleIn,ctrlIn] = whdlFramesToSamples(txMessages, ...
    idlecyclesbetweensamples,idlecyclesbetweenframes,samplesizeIn);

startIn = ctrlIn(:,1);
endIn = ctrlIn(:,2);
validIn = ctrlIn(:,3);

simTime = size(ctrlIn,1)+6;
sampletime = 1;
```

Run the Simulink model.

```
set_param([modelname '/EncodeAndPuncture'],'Open','on');
sim(modelname);
```
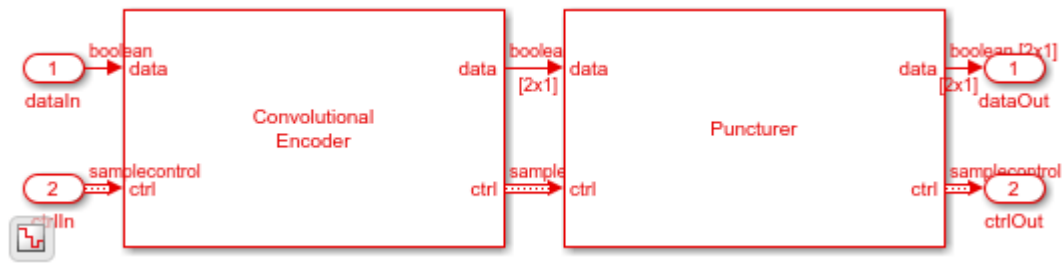
Convert the streaming samples from the Simulink block output to framed data for comparison.

```
sampleOut = squeeze(sampleOut);
startOut = ctrlOut(:,1);
endOut   = ctrlOut(:,2);
validOut = ctrlOut(:,3);

idxStart = find(startOut.*validOut);
idxEnd = find(endOut.*validOut);
```

Generate reference samples using `convenc` MATLAB function.

```
for ii = 1:numFrames
    txCodeword{ii} = convenc([txMessages{ii};false(constraintLength-1,1)],...
        trellis,puncVector);
end
```

Compare the output samples against the generated input samples.

```
fprintf('\nEncoded Samples\n');
for ii = 1:numFrames
    idx = idxStart(ii):idxEnd(ii);
    idxValid = (validOut(idx));
    dataOut = sampleOut(:,idx);
    hdlTxCoded = dataOut(:,idxValid);
    numBitsErr = sum(xor(txCodeword{ii},hdlTxCoded(:)));
    fprintf('Number of samples mismatched in the frame #%d: %d bits\n',ii,numBitsErr);
end
```

```
Encoded Samples
Number of samples mismatched in the frame #1: 0 bits
Number of samples mismatched in the frame #2: 0 bits
Number of samples mismatched in the frame #3: 0 bits
Number of samples mismatched in the frame #4: 0 bits
Number of samples mismatched in the frame #5: 0 bits
```

See Also

Blocks
Convolutional Encoder | Puncturer

OFDM Demodulation of Streaming Samples

This example shows how to use the OFDM Demodulator block to demodulate complex time-domain OFDM samples to subcarriers for a vector input. This example model supports HDL code generation for the `OFDMDemod` subsystem.

Set Up Input Data Parameters

Set up these workspace variables for the model to use. You can modify these values according to your requirement.

```
rng('default');
numOFDMSym = 2;
maxFFTLen = 128;
DCRem = true;
RoundingMethod = 'floor';
Normalize = false;
cpFraction = 1;
fftLen = 64;
cpLen = 16;
numLG = 6;
numRG = 5;
if DCRem
    NullInd = [1:numLG fftLen/2+1 fftLen-numRG+1:fftLen];
else
    NullInd = [1:numLG fftLen-numRG+1:fftLen]; %#ok<UNRCH>
end
symbOffset = floor(cpFraction*cpLen);
vecLen = 2;
```

Generate Frames of Random Input Samples

Generate frames of random samples using the MATLAB function randn.

```
data = randn(fftLen,numOFDMSym)+1i*randn(fftLen,numOFDMSym);
dataIn = ofdmmod(data,fftLen,cpLen);
```

Convert Frames to Stream of Random Samples

Convert frames of random samples to a stream of random samples to provide them as input to the block.

```
data = dataIn(:);
valid = true(length(dataIn)/vecLen,1);
fftSig = fftLen*ones(length(dataIn),1);
CPSig = cpLen*ones(length(dataIn),1);
LGSig = numLG*ones(length(dataIn),1);
RGSig = numRG*ones(length(dataIn),1);
resetSig = false(length(data),1);
sampleTime = 1/vecLen;
stopTime = (maxFFTLen*3*numOFDMSym)/vecLen;
```

Run Simulink model

Running the model imports the input signal variables to the block from the script and exports a stream of demodulated output samples from the block to the MATLAB workspace.

```
modelname = 'genhdlOFDMDemodulatorModel';
open_system(modelname);
out = sim(modelname);
simOut = squeeze(out.dataOut(:,1,out.validOut==1));
```
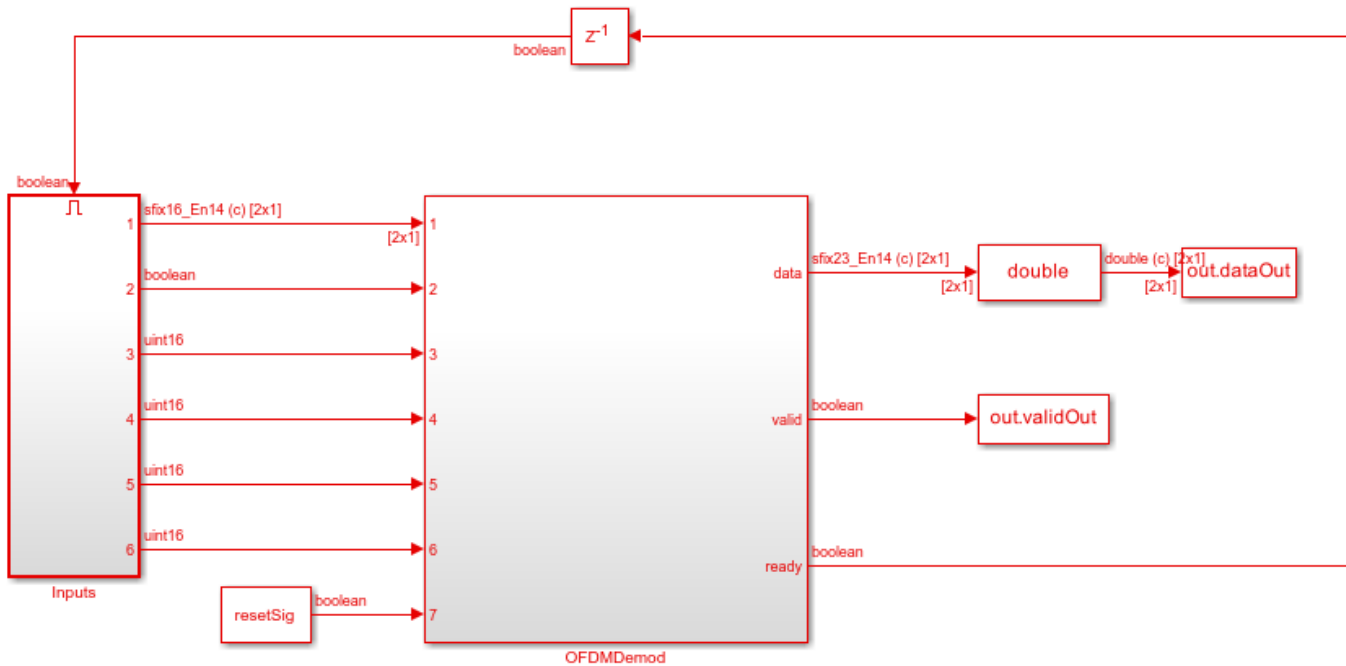

Copyright 2019 The MathWorks, Inc.

Demodulate Stream Samples Using MATLAB Function

Demodulate stream of random input samples using the `ofdmdemod_baseline` function.

```
[dataOut1] = ofdmdemod_baseline(dataIn,fftLen,cpLen,symbOffset,NullInd.',[],Normalize,RoundingMe
matOut = dataOut1(:);
```

Compare Simulink Block Output with MATLAB Function Output

Compare the output of the Simulink model against the output of `ofdmdemod_baseline` function.

```
figure('units','normalized','outerposition',[0 0 1 1])
subplot(2,1,1)
plot(real(matOut(:)));
hold on;
plot(real(simOut(:)));
grid on
legend('Reference','Simulink')
xlabel('Sample Index')
ylabel('Magnitude')
title('Comparison of Simulink block and MATLAB function - Real part')

subplot(2,1,2)
plot(imag(matOut(:)));
hold on;
plot(imag(simOut(:)));
```

```matlab
grid on
legend('Reference','Simulink')
xlabel('Sample Index')
ylabel('Magnitude')
title('Comparison of Simulink block and MATLAB function - Imaginary part')

sqnrRealdB=10*log10(var(real(simOut(:)))/abs(var(real(simOut(:)))-var(real(matOut(:)))));
sqnrImagdB=10*log10(var(imag(simOut(:)))/abs(var(imag(simOut(:)))-var(imag(matOut(:)))));

fprintf('\n OFDM Demodulator: \n SQNR of real part is %.2f dB',sqnrRealdB);
fprintf('\n SQNR of imaginary part is %.2f dB\n',sqnrImagdB);
```

```
 OFDM Demodulator:
 SQNR of real part is 47.77 dB
 SQNR of imaginary part is 42.69 dB
```

Comparison of Simulink block and MATLAB function - Real part

Comparison of Simulink block and MATLAB function - Imaginary part
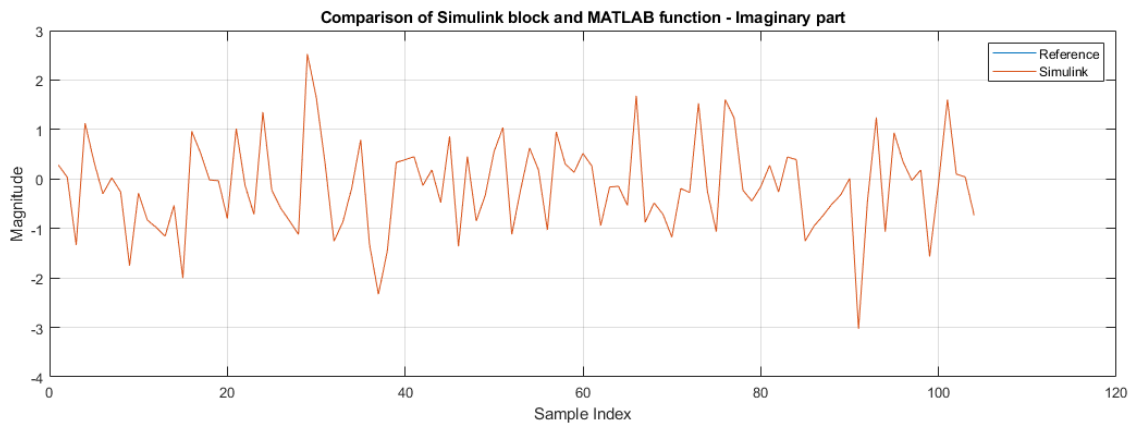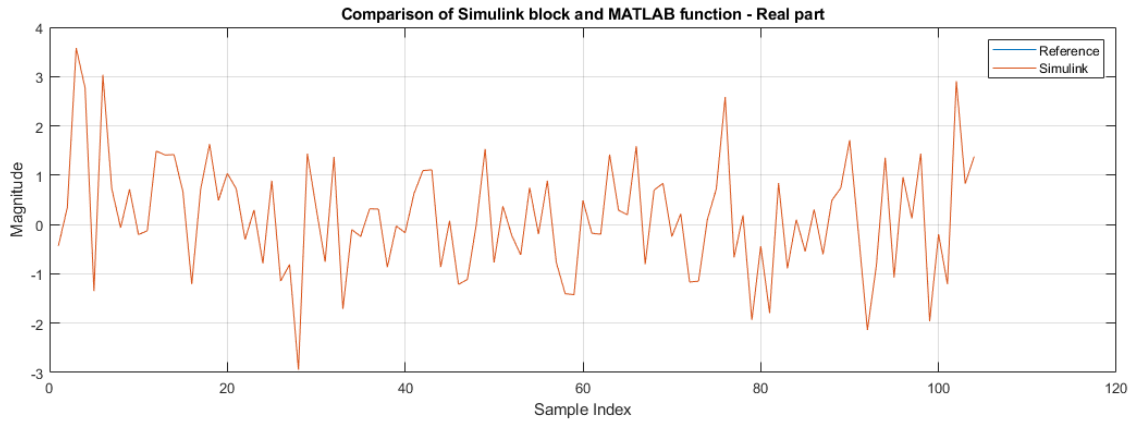
See Also

Blocks
OFDM Demodulator

Decode and recover message from RS codeword

This example shows how to use RS Decoder block to decode and recover a message from a Reed-Solomon (RS) codeword. In this example, a set of random inputs are generated and provided to the `comm.RSEncoder` function and its output is provided to the RS Decoder block. The output of the RS Decoder block is compared with the input of the `comm.RSEncoder` function to check whether any errors are encountered. The example model supports HDL code generation for the `RS Decoder` subsystem.

Set Up Input Data parameters

Specify the input variables.

```
n = 255;
k = 239;
primPoly = [1 0 0 0 1 1 1 0 1];
B = 1;
nMessages = 4;
data = zeros(k,nMessages);
inputMsg = (zeros(n,nMessages));
startSig = [];
endSig = [];
```

Generate Random Input Samples

Generate random samples based on n,k, and m values and provide them as input to the `comm.RSEncoder` function. Here, n is the codeword length, k is the message length, and m is the gap between the frames.

```
hRSEnc = comm.RSEncoder;
hRSEnc.CodewordLength = n;
hRSEnc.MessageLength = k;
m=0;

for ii = 1:nMessages
data(:,ii) = randi([0 n],k,1);
[inputMsg(1:n,ii)] = hRSEnc(data(:,ii));
inputMsg1(1:n,ii) = inputMsg(1:n,ii);
[inputMsg(n+1:n+m,ii)] = zeros(m,1);
validIn(1:n,ii) = true;
validIn(n+1:n+m) = false;

endSig = [endSig [false(n-1,1); true;false(m,1);]];
startSig = [startSig [true;false(n+m-1,1)]];

end

refOutput = data(:);
```

Import Encoded Random Input Samples to the Simulink® Model

Provide the output of the `comm.RSEncoder` function as input to the Simulink model.

```
simDataIn = inputMsg(:);
simStartIn = startSig(:);
simEndIn = endSig(:);
simValidIn = validIn(:);
```

Run the Simulink Model

Run the Simulink model to export the decoded samples of the Simulink block to the MATLAB®
workspace.

```
modelname = 'RSDecoder';
open_system(modelname);
out = sim(modelname);
simOutput = out.dataOut(out.validOut);
```
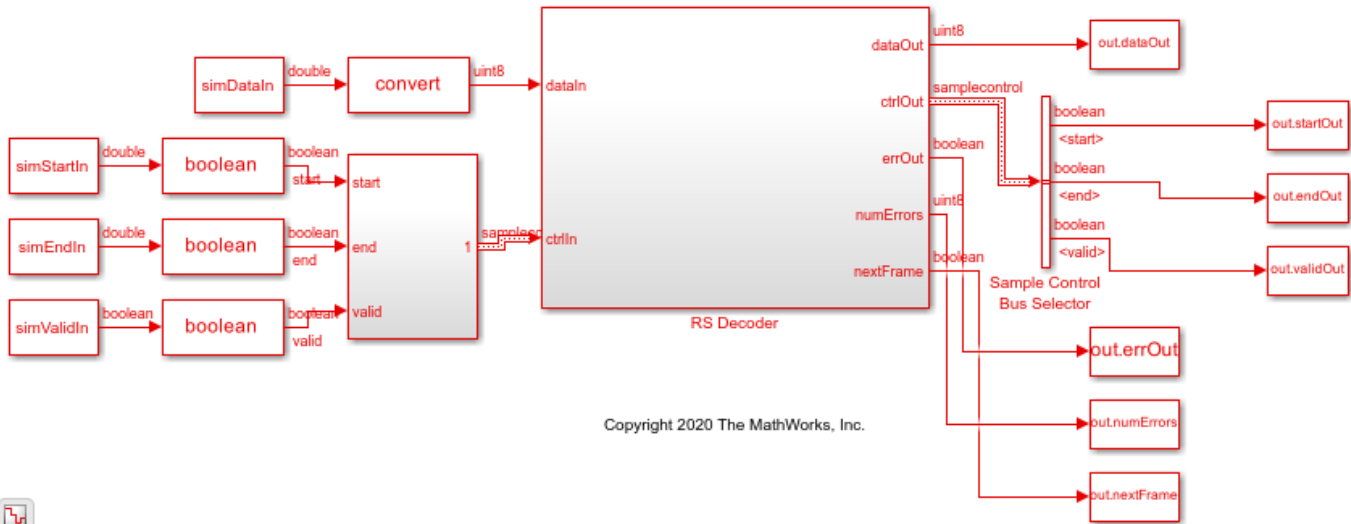

Copyright 2020 The MathWorks, Inc.

Compare Simulink Block Output with MATLAB Function Input

Compare the output of the Simulink block with the input provided to the `comm.RSEncoder` function.

```
fprintf('\nHDL RS Decoder\n');
difference = double(simOutput) - double(refOutput);
fprintf('\nTotal number of samples differed between Simulink block output and MATLAB function ou
```

HDL RS Decoder

Total number of samples differed between Simulink block output and MATLAB function output is: 0

See Also

Blocks
RS Decoder

LDPC Encode and Decode of 5G NR Streaming Data

This example shows how to simulate the NR LDPC Encoder and NR LDPC Decoder Simulink® blocks and compare the hardware-optimized results with the results from the 5G Toolbox™ functions. These blocks support scalar and vector inputs. The NR LDPC Decoder block enables you to select either Min-sum or Normalized min-sum algorithm for decoding operation.

Generate Input Data for Encoder

Choose a series of input values for bgn and liftingSize according to the 5G new radio (NR) standard. Generate the corresponding input vectors for the selected base graph number (bgn) and liftingSize values. Generate random frames of input data and convert them to Boolean data and control signal that indicates the frame boundaries. `encFrameGap` accommodates the latency of the NR LDPC Encoder block for bgn and liftingSize values. Use the **nextFrame** signal to determine when the block is ready to accept the start of the next input frame.

```
bgn         = [0; 1; 1; 0];
liftingSize = [4; 384; 144; 208];
numFrames = 4;
serial = false; % true for serial inputs and false for parallel inputs

encbgnIn = [];encliftingSizeIn = [];
msg = {numFrames};
K =[];N = [];
encSampleIn = [];encStartIn = [];encEndIn = [];encValidIn = [];
encFrameGap = 2500;
for ii = 1:numFrames
    if bgn(ii) == 0
        K(ii) = 22;
        N(ii) = 66;
    else
        K(ii) = 10;
        N(ii) = 50;
    end
    frameLen = liftingSize(ii) * K(ii);
    msg{ii} = randi([0 1],1,frameLen);
    if serial
    len = K(ii) * liftingSize(ii);
    encFrameGap = liftingSize(ii) * N(ii) + 2500;
    else
    len = K(ii) * ceil(liftingSize(ii)/64); %#ok<*UNRCH>
    encFrameGap = 2500;
    end

    encIn = ldpc_dataFormation(msg{ii},liftingSize(ii),K(ii),serial);

    encSampleIn = logical([encSampleIn encIn zeros(size(encIn,1),encFrameGap)]); %#ok<*AGROW>
    encStartIn = logical([encStartIn  1  zeros(1,len-1)  zeros(1,encFrameGap)]);
    encEndIn = logical([encEndIn    zeros(1,len-1) 1   zeros(1,encFrameGap)]);
    encValidIn = logical([encValidIn    ones(1,len)      zeros(1,encFrameGap)]);
    encbgnIn = logical([encbgnIn   repmat(bgn(ii),1,len) zeros(1,encFrameGap)]);
    encliftingSizeIn = uint16([encliftingSizeIn repmat(liftingSize(ii),1,len) zeros(1,encFrameGap
end

encSampleIn = timeseries(logical(encSampleIn'));
```

```
sampleTime = 1;
simTime = length(encValidIn);     %#ok<NASGU>
```

Run Encoder Model

The HDL Algorithm subsystem contains the NR LDPC Encoder block. Running the model imports the input signal variables encSampleIn, encStartIn, encEndIn, encValidIn, encbgnIn, encliftingSizeIn, sampleTime, and simTime and exports sampleOut and ctrlOut variables to the MATLAB® workspace.

```
open_system('NRLDPCEncoderHDL');
encOut = sim('NRLDPCEncoderHDL');
```
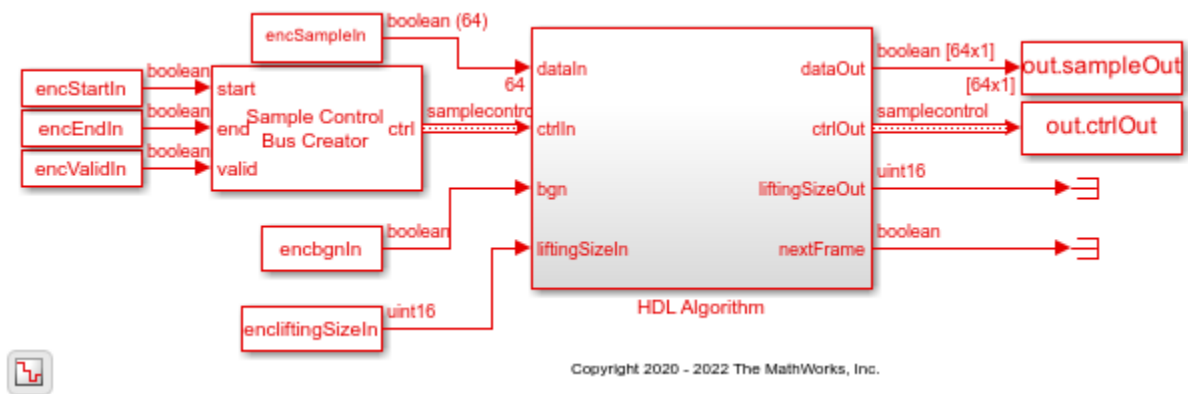

Copyright 2020 - 2022 The MathWorks, Inc.

Verify Encoder Results

Convert the streaming data output of the block to frames and then compare them with the output of the nrLDPCEncode function.

```
startIdx = find(encOut.ctrlOut.start.Data);
endIdx = find(encOut.ctrlOut.end.Data);

for ii = 1:numFrames
    encHDL{ii} = ldpc_dataExtraction(encOut.sampleOut.Data,liftingSize(ii),startIdx(ii),endIdx(ii
    encRef = nrLDPCEncode(msg{ii}',bgn(ii)+1);
    error = sum(abs(encRef - encHDL{ii}));
    fprintf(['Encoded Frame %d: Behavioral and ' ...
        'HDL simulation differ by %d bits\n'],ii,error);
end
```

```
Encoded Frame 1: Behavioral and HDL simulation differ by 0 bits
Encoded Frame 2: Behavioral and HDL simulation differ by 0 bits
Encoded Frame 3: Behavioral and HDL simulation differ by 0 bits
Encoded Frame 4: Behavioral and HDL simulation differ by 0 bits
```

Generate Input Data for Decoder

Use the encoded data from the NR LDPC Encoder block to generate input log-likelihood ratio (LLR) values for the NR LDPC Decoder block. Use channel, modulator, and demodulator system objects to add some noise to the signal. Again, create vectors of bgn and liftingSize and convert the frames of data to LLRs with a control signal that indicates the frame boundaries. decFrameGap accommodates

the latency of the NR LDPC Decoder block for bgn, liftingSize, and number of iterations. Use the **nextFrame** signal to determine when the block is ready to accept the start of the next input frame.

```matlab
nVar = 1.2;
chan = comm.AWGNChannel('NoiseMethod','Variance','Variance',nVar);
bpskMod = comm.BPSKModulator;
bpskDemod = comm.BPSKDemodulator('DecisionMethod', ...
    'Approximate log-likelihood ratio','Variance',nVar);

algo = 'Normalized min-sum'; % 'Min-sum' or 'Normalized min-sum'
if strcmpi(algo,'Min-sum')
    alpha = 1;
else
    alpha = 0.75;
end

numIter = 8;
decbgnIn = [];decliftingSizeIn = [];
rxLLR = {numFrames};
decSampleIn = [];decStartIn = [];decEndIn = [];decValidIn = [];

for ii=1:numFrames
    mod = bpskMod(double(encHDL{ii}));
    rSig = chan(mod);
    rxLLR{ii} = fi(bpskDemod(rSig),1,6,0);

    if serial
        len = N(ii)* liftingSize(ii);
        decFrameGap = numIter *7000 + liftingSize(ii) * K(ii);
     else
        len = N(ii)* ceil(liftingSize(ii)/64);
        decFrameGap = numIter *1200;
     end

    decIn = ldpc_dataFormation(rxLLR{ii}',liftingSize(ii),N(ii),serial);

    decSampleIn = [decSampleIn decIn zeros(size(decIn,1),decFrameGap)]; %#ok<*AGROW>
    decStartIn = logical([decStartIn  1  zeros(1,len-1)  zeros(1,decFrameGap)]);
    decEndIn = logical([decEndIn    zeros(1,len-1) 1   zeros(1,decFrameGap)]);
    decValidIn = logical([decValidIn    ones(1,len)       zeros(1,decFrameGap)]);
    decbgnIn = logical([decbgnIn   repmat(bgn(ii),1,len) zeros(1,decFrameGap)]);
    decliftingSizeIn = uint16([decliftingSizeIn repmat(liftingSize(ii),1,len) zeros(1,decFrameGap)
end

decSampleIn = timeseries(fi(decSampleIn',1,6,0));

simTime = length(decValidIn);
```

Run Decoder Model

The `HDL Algorithm` subsystem contains the NR LDPC Decoder block. Running the model imports the input signal variables `decSampleIn`, `decStartIn`, `decEndIn`, `decValidIn`, `decbgnIn`, `decliftingSizeIn`, `numIter`, `sampleTime`, and `simTime` and exports a stream of decoded output samples `sampleOut` along with control signal `ctrlOut` to the MATLAB workspace.

```matlab
open_system('NRLDPCDecoderHDL');
if alpha ~= 1
    set_param('NRLDPCDecoderHDL/HDL Algorithm/NR LDPC Decoder','Algorithm','Normalized min-sum')
```

```
else
    set_param('NRLDPCDecoderHDL/HDL Algorithm/NR LDPC Decoder','Algorithm','Min-sum');
end
decOut = sim('NRLDPCDecoderHDL');
```
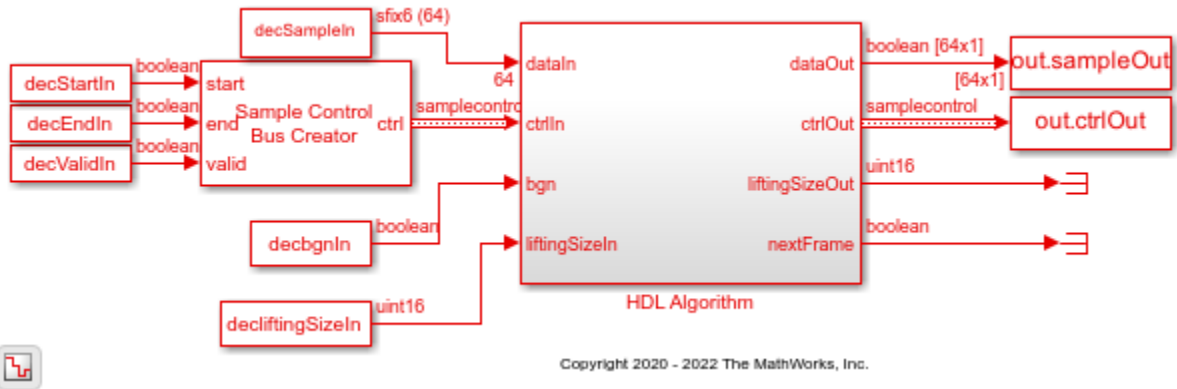

Copyright 2020 - 2022 The MathWorks, Inc.

Verify Decoder Results

Convert the streaming data output of the block to frames and then compare them with the output of the `nrLDPCDecode` function.

```
startIdx = find(decOut.ctrlOut.start.Data);
endIdx = find(decOut.ctrlOut.end.Data);

for ii = 1:numFrames
    decHDL{ii} = ldpc_dataExtraction(decOut.sampleOut.Data,liftingSize(ii),startIdx(ii),endIdx(ii));
    decRef = nrLDPCDecode(double(rxLLR{ii}),bgn(ii)+1,numIter, 'Algorithm','Normalized min-sum','S
            'Termination','max');
    error = sum(abs(double(decRef) - decHDL{ii}));
    fprintf(['Decoded Frame %d: Behavioral and ' ...
        'HDL simulation differ by %d bits\n'],ii,error);
end

Decoded Frame 1: Behavioral and HDL simulation differ by 0 bits
Decoded Frame 2: Behavioral and HDL simulation differ by 0 bits
Decoded Frame 3: Behavioral and HDL simulation differ by 0 bits
Decoded Frame 4: Behavioral and HDL simulation differ by 0 bits
```

See Also

Blocks
NR LDPC Decoder | NR LDPC Encoder

Functions
`nrLDPCDecode` | `nrLDPCEncode`

Estimate Channel Using Input Data and Reference Subcarriers

This example shows how to use the OFDM Channel Estimator block to estimate a channel using input data and reference subcarriers. In this example model, the averaging and interpolation features are enabled. The `HDL Algorithm` subsystem in this example model supports HDL code generation.

Set Input Data Parameters

Set up these workspace variables for the model to use. You can modify these values according to your requirement.

```
rng('default');
numOFDMSym = 980;
numOFDMSymToBeAvg = 14;
interpolFac = 3;
maxNumScPerSym = 72;
numOFDMSymPerFrame = 140;
numScPerSym = 72;
```

Generate Sinusoidal Input Data Subcarriers

Use the `numScPerSym` and `numOFDMSym` variables to generate complex sinusoidal input data subcarriers with their real and imaginary parts generated separately.

```
dataInGrid = zeros(numScPerSym,numOFDMSym);
for numScPerSymCount = 0:numScPerSym - 1
    for numOFDMSymCount = 0:numOFDMSym - 1
        realXgain = 1 + .2*sin(2*pi*numScPerSymCount/numScPerSym);
        realYgain = 1 + .5*sin(2*pi*numOFDMSymCount/numOFDMSymPerFrame);
        imagXgain = 1 + .3*sin(2*pi*numScPerSymCount/numScPerSym);
        imagYgain = 1 + .4*sin(2*pi*numOFDMSymCount/numOFDMSymPerFrame);
        dataInGrid(numScPerSymCount+1,numOFDMSymCount+1) = realXgain*realYgain + 1i*(imagXgain*i
    end
end
validIn = true(1,length(dataInGrid(:)));

figure(1);
surf(real(dataInGrid))
xlabel('OFDM Symbols')
ylabel('Subcarriers')
zlabel('Magnitude')
title('Input Data Grid (Real Part)')

figure(2);
surf(imag(dataInGrid))
xlabel('OFDM Symbols')
ylabel('Subcarriers')
zlabel('Magnitude')
title('Input Data Grid (Imaginary Part)')
```

Input Data Grid (Real Part)

Input Data Grid (Imaginary Part)
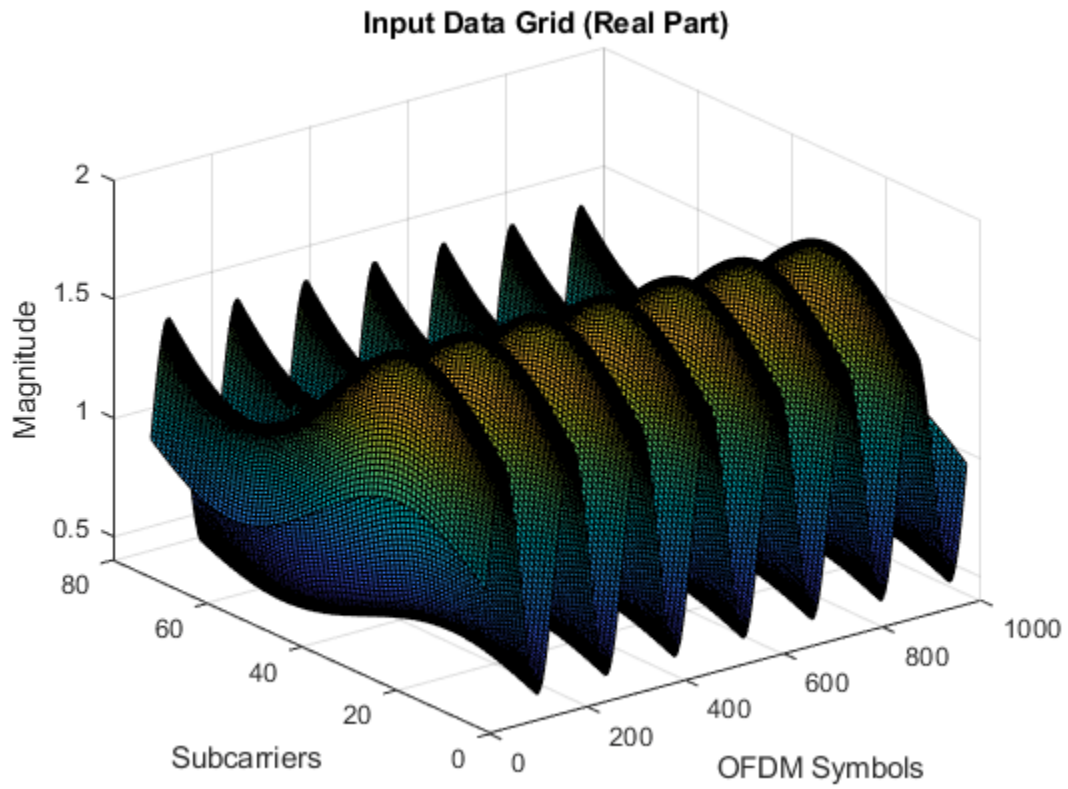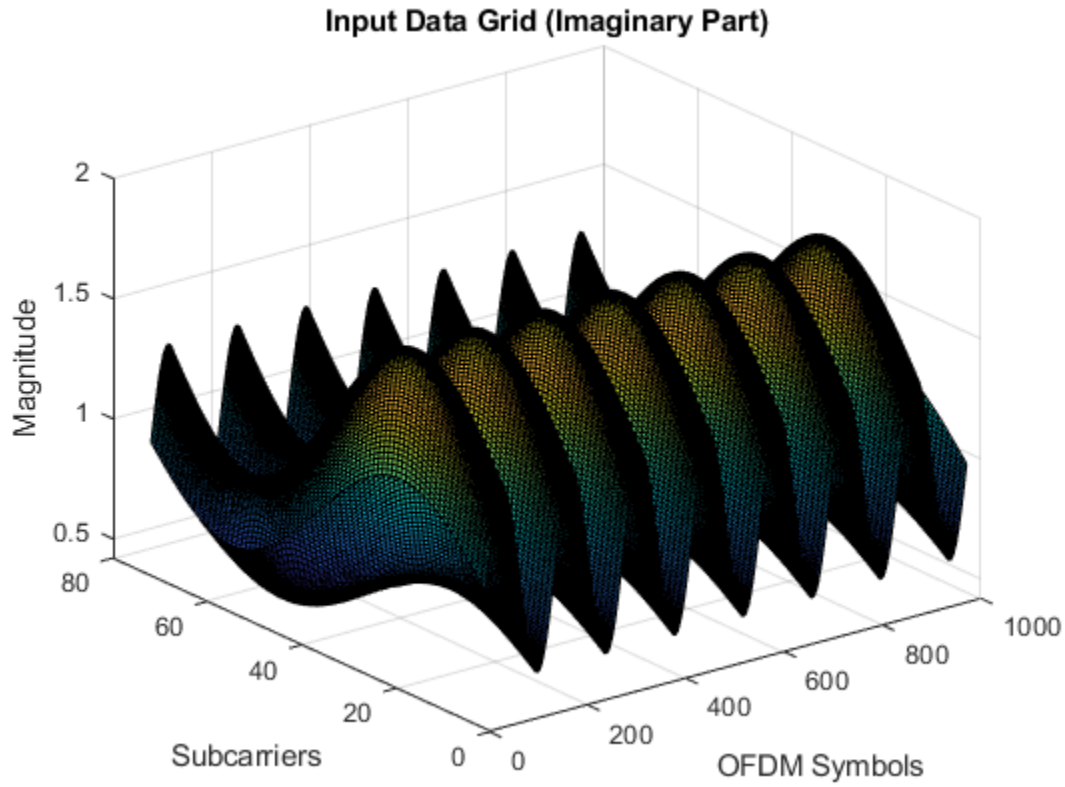
Generate Reference Data Subcarriers

Generate reference data subcarriers.

```
refDataIn = randsrc(size(dataInGrid(:),1),size(dataInGrid(:),2),[1 1]);
refValidIn = boolean(zeros(1,numOFDMSym*numScPerSym));
startRefValidIndex = randi(interpolFac,1,1);
for numOFDMSymCount = 1:numOFDMSym
    refValidIn(startRefValidIndex+(numOFDMSymCount-1)*numScPerSym:interpolFac:numScPerSym*numOFDM
end
```

Generate Signal with Number of Subcarriers per Symbol

Generate a signal with the number of subcarriers per symbol.

```
numScPerSymIn = numScPerSym*true(1,length(dataInGrid(:)));
resetSig = false(1,length(dataInGrid(:)));
```

Run Simulink® Model

Run the model. Running the model imports the input signal variables from the MATLAB workspace to the OFDM Channel Estimator block in the model.

```
modelname = 'genhdlOFDMChannelEstimatorModel';
open_system(modelname);
out = sim(modelname);
```
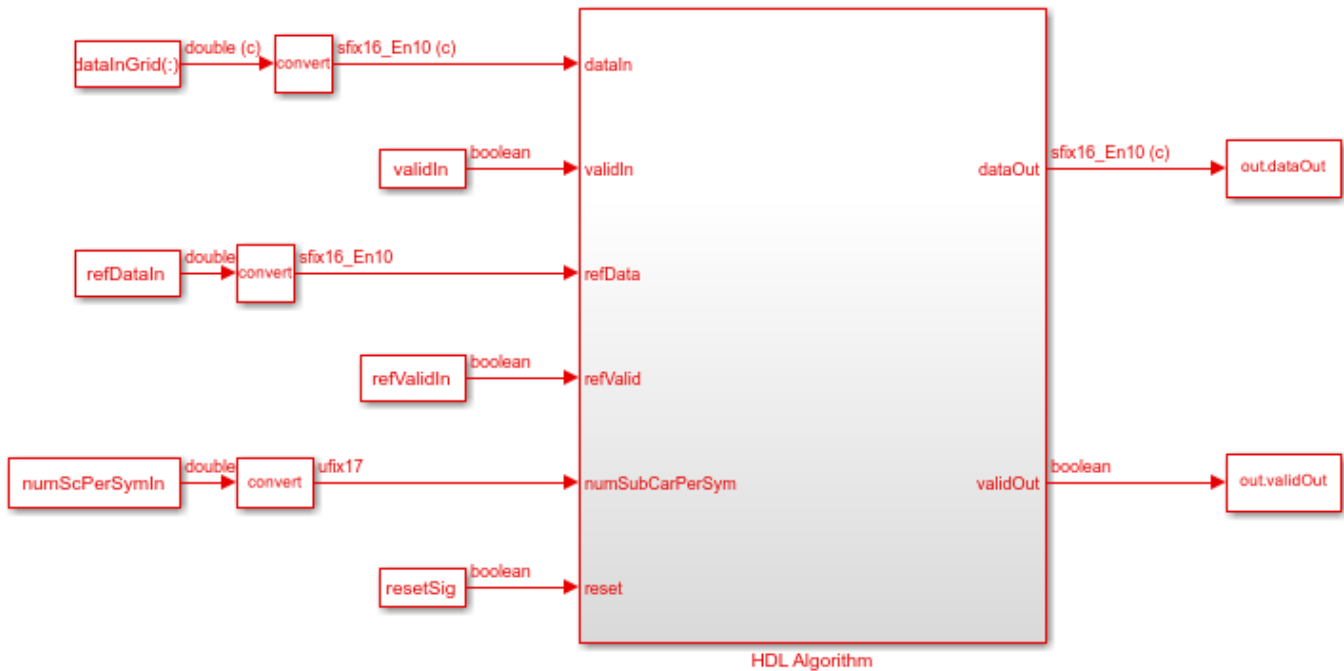
Copyright 2020 The MathWorks, Inc.

Export Stream of Channel Estimates from Simulink to MATLAB® Workspace

Export the output of the OFDM Channel Estimator block to the MATLAB workspace. Plot the real part and imaginary part of the exported block output.

```
simOut = out.dataOut.Data(out.validOut.Data);
N = length(simOut) - mod(length(simOut),numScPerSym);
temp = simOut(1:N);
channelEstimateSimOut = reshape(temp,numScPerSym,length(temp)/numScPerSym);

figure(3);
surf(real(channelEstimateSimOut))
xlabel('OFDM Symbols')
ylabel('Subcarriers')
zlabel('Magnitude')
title('Channel Estimator Output (Real Part)')

figure(4);
surf(imag(channelEstimateSimOut))
xlabel('OFDM Symbols')
ylabel('Subcarriers')
zlabel('Magnitude')
title('Channel Estimator Output (Imaginary Part)')
```

Channel Estimator Output (Real Part)

Channel Estimator Output (Imaginary Part)
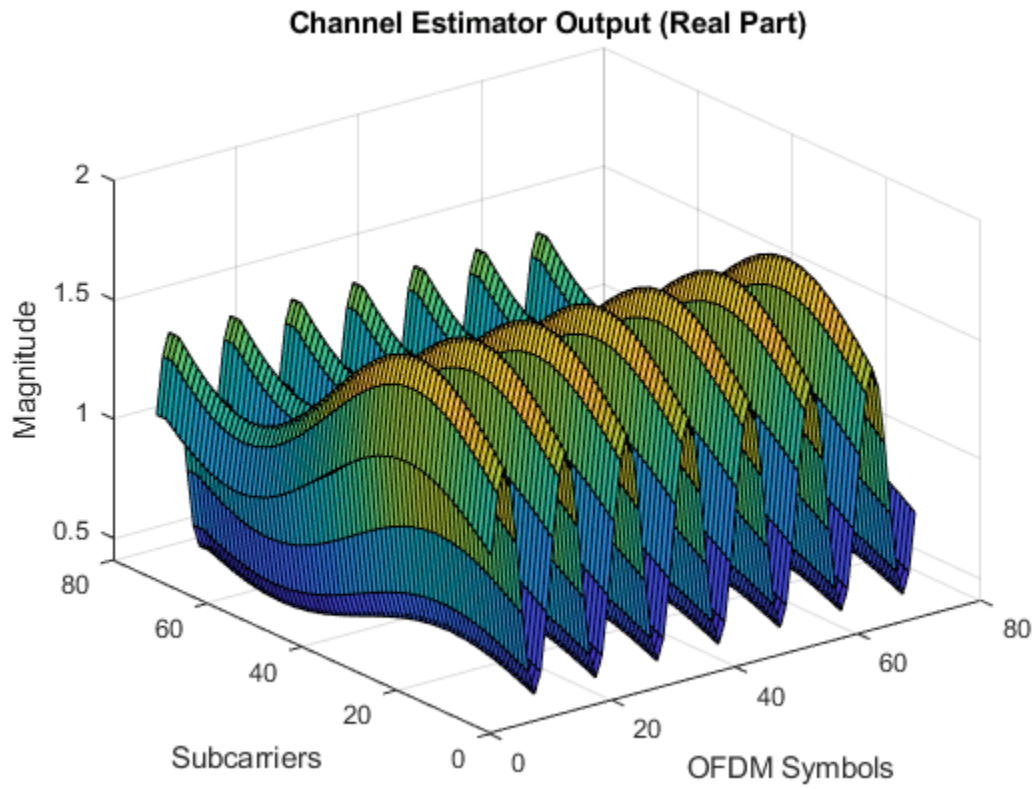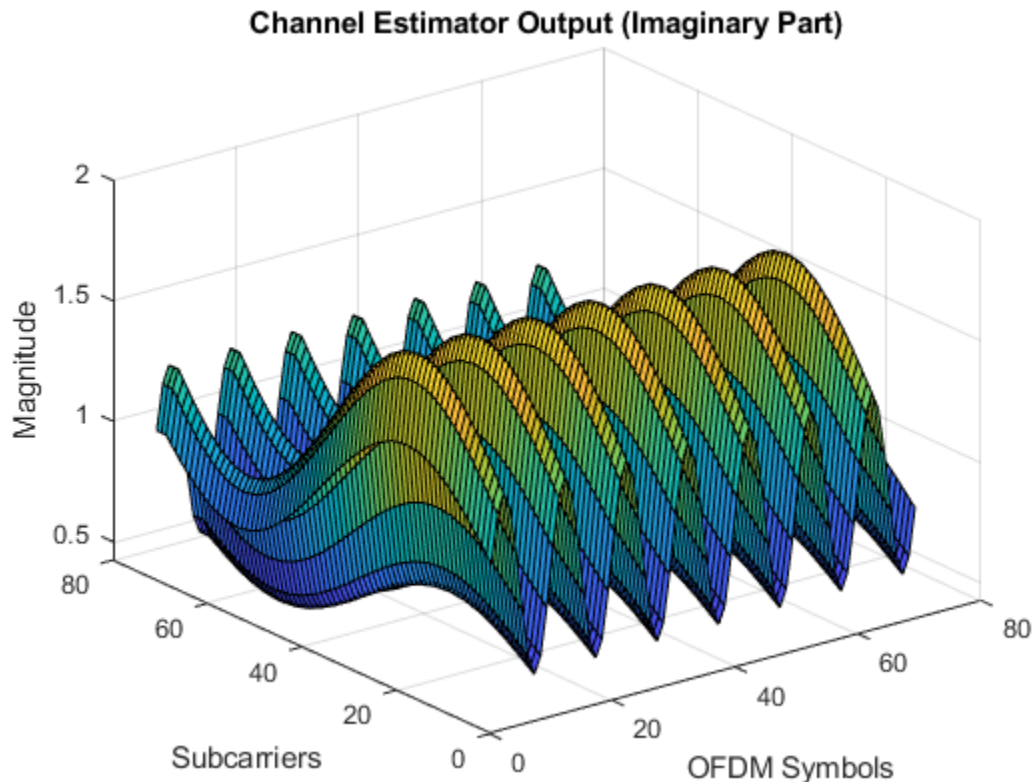
Estimate Channel Using MATLAB Function

Estimate the channel by using the `channelEstReference` function with the sinusoidal input data subcarriers.

```
dataOut1 = channelEstReference(...
    numOFDMSymToBeAvg,interpolFac,numScPerSym,numOFDMSym, ...
    dataInGrid(:),validIn,refDataIn,refValidIn,numScPerSymIn);
matlabOut = dataOut1(:);
matOut = zeros(numel(matlabOut)*numScPerSym,1);
for ii= 1:numel(matlabOut)
loadArray = [matlabOut(ii).dataOut; zeros((numel(matlabOut)-1)*numScPerSym,1)];
shiftArray = circshift(loadArray,(ii-1)*numScPerSym);
matOut =  matOut + shiftArray;
end
```

Compare Simulink Block Output with MATLAB Function Output

Compare the OFDM Channel Estimator block output with `channelEstReference` function output. Plot the output comparison as a real part and an imaginary part using separate plots.

```
figure('units','normalized','outerposition',[0 0 1 1])
subplot(2,1,1)
plot(real(matOut(:)));
hold on;
plot(real(simOut(:)));
grid on
legend('MATLAB reference output','Simulink block output')
```

```matlab
xlabel('Sample Index')
ylabel('Magnitude')
title('Comparison of Simulink Block and MATLAB Function (Real Part)')

subplot(2,1,2)
plot(imag(matOut(:)));
hold on;
plot(imag(simOut(:)));
grid on
legend('MATLAB reference output','Simulink block output')
xlabel('Sample Index')
ylabel('Magnitude')
title('Comparison of Simulink Block and MATLAB Function (Imaginary Part)')

sqnrRealdB = 10*log10(double(var(real(simOut(:)))/abs(var(real(simOut(:)))-var(real(matOut(:))))
sqnrImagdB = 10*log10(double(var(imag(simOut(:)))/abs(var(imag(simOut(:)))-var(imag(matOut(:))))

fprintf('\n OFDM Channel Estimator \n SQNR of real part: %.2f dB',sqnrRealdB);
fprintf('\n SQNR of imaginary part: %.2f dB\n',sqnrImagdB);
```

```
 OFDM Channel Estimator
 SQNR of real part: 38.54 dB
 SQNR of imaginary part: 37.77 dB
```

Comparison of Simulink Block and MATLAB Function (Real Part)

Comparison of Simulink Block and MATLAB Function (Imaginary Part)
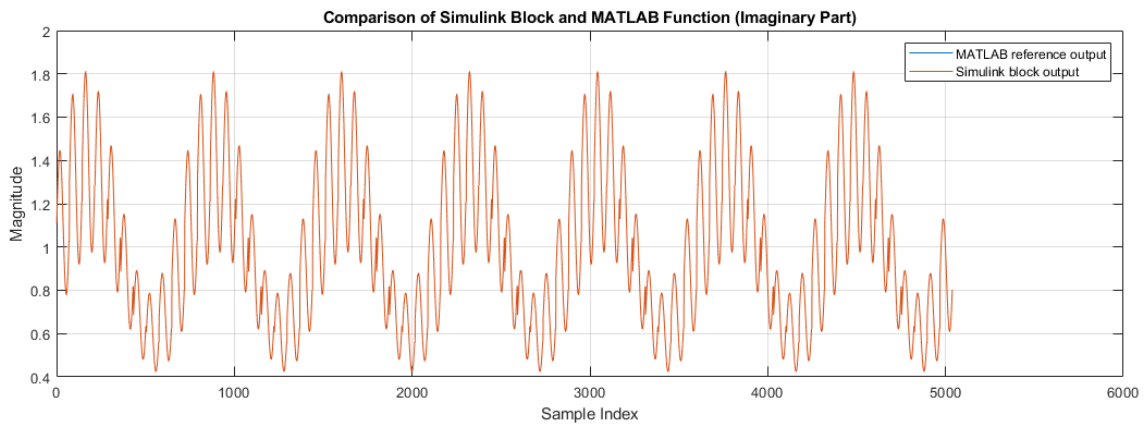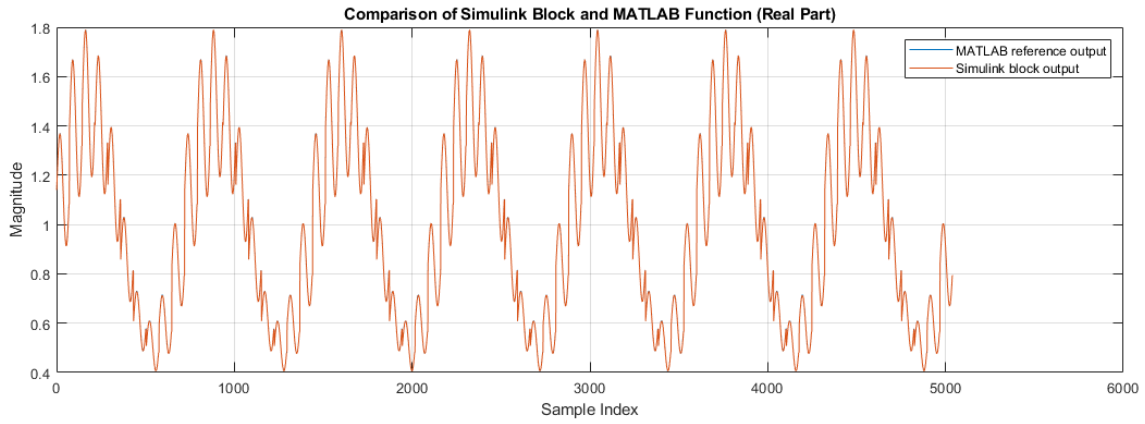
See Also

Blocks
OFDM Channel Estimator

Modulate and Demodulate OFDM Streaming Samples

This example model shows how to use OFDM Modulator and OFDM Demodulator blocks in Wireless HDL Toolbox™. In this model, an OFDM Modulator and an OFDM Demodulator block are connected back-to-back. The **OFDM parameters source** parameter in these blocks is set to `Input port`, enabling you to dynamically change the input values of these blocks. You can change these values using the script in this example. These blocks support scalar and vector inputs. To verify the functionality of these blocks, the input provided to the OFDM Modulator block is compared with the output of the OFDM Demodulator block. The `OFDMModDemod` HDL subsystem in this example supports HDL code generation.

Set Input Data Parameters

Set up these workspace variables for the Simulink® model to use. You can modify these values according to your requirement. The model in this example uses these workspace variables `fftLen`, `maxFFTLen`, `cpLen`, `numLG`, `numRG`, `numSymb`, and `DCNull` to configure the OFDM Modulator and OFDM Demodulator blocks.

```
fftLen = 64;        % FFT length
maxFFTLen = 128;    % Maximum FFT length
cpLen = 16;         % Cylic prefix length
numLG = 6;          % Number of left guard carriers
numRG = 5;          % Number of right guard carriers
numSymb = 2;        % Number of right guard carriers
DCNull = 1;         % 1 or 0
vecLen = 4;         % Vector length - 1, 2, 4, 8, 16, 32, or 64
if DCNull==1
    numActData = fftLen - (numLG+numRG+1);
else
    numActData = fftLen - (numLG+numRG);
end
```

Generate Input Data Frames

Generate random frames of complex input data and a control signal that indicates the frame boundaries.

```
rng default;
dataIn = complex(randn(numActData*numSymb,1),randn(numActData*numSymb,1));
dataVec = []; % Store data arranged in vector form
presentSymbDataStartIndex = 0;
for ii = 1:numSymb
    counter = 0;
    for jj = 1:ceil(numActData/vecLen)
        if jj == ceil(numActData/vecLen)
            numZerosTobeAppended = vecLen - (numActData-counter);
            dataVec = [dataVec [dataIn(presentSymbDataStartIndex+counter+(1:vecLen-numZerosTobeA
        else
            dataVec = [dataVec dataIn(presentSymbDataStartIndex+counter+(1:vecLen))];
        end
        counter = counter + vecLen;
    end
    presentSymbDataStartIndex = presentSymbDataStartIndex + numActData;
end
```

```
data = dataVec.';
valid = boolean(ones(size(data,1),1)); % Valid signal generation

sampling_time = 1;
stoptime = maxFFTLen*6*numSymb;
```

Run Simulink Model

Run the model to import the input signal variables `dataIn`, `validIn`, `fftLen`, `maxFFTLen`, `cpLen`, `numLG`, `numRG`, `numSymb`, and `DCNull` from the workspace to the OFDM Modulator block. The OFDM Modulator block returns OFDM-modulated output samples and a control signal. These OFDM-modulated samples are fed to the OFDM Demodulator block, which returns OFDM demodulated samples.

```
open_system('genhdlOFDMModDemodExample')
sim('genhdlOFDMModDemodExample');

% Store valid data from Simulink model
dataOut1 = dataOut.data;
simOut = dataOut1(:,:,validOut);
simOut = simOut(:);
```
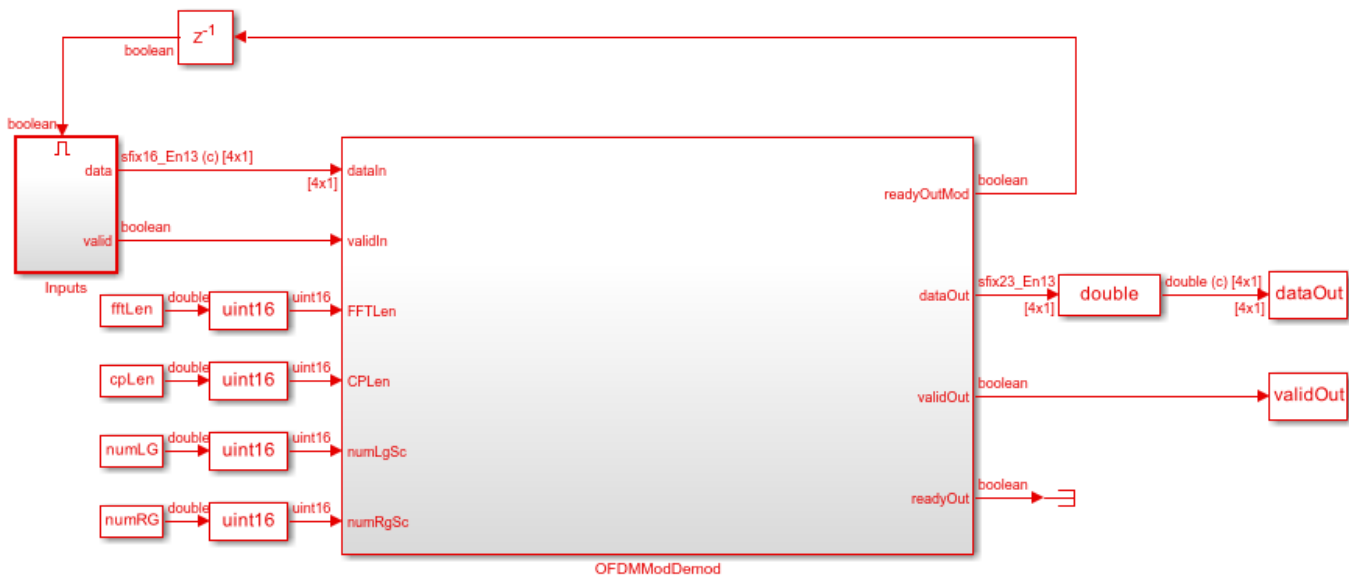

Copyright 2019 - 2021 The MathWorks, Inc.

Compare OFDM Modulator Input with OFDM Demodulator Output

Compare the input data provided to the OFDM Modulator block with the output data generated from the OFDM Demodulator block.

```
figure('units','normalized','outerposition',[0 0 1 1])
subplot(2,1,1);
plot(real(dataIn(1:size(simOut))));
hold on
plot(squeeze(real(simOut)));
legend('Real part of reference data','Real part of demodulated data');
title('Comparison of OFDM Modulator Input with OFDM Demodulator Output - Real Part');
```

```
xlabel('OFDM Subcarriers');
ylabel('Real Part');

subplot(2,1,2)
plot(imag(dataIn(1:size(simOut))));
hold on
plot(squeeze(imag(simOut)))
legend('Imaginary part of reference data','Imaginary part of demodulated data');
title('Comparison of OFDM Modulator Input with OFDM Demodulator Output - Imaginary Part');
xlabel('OFDM Subcarriers');
ylabel('Imaginary Part');
```
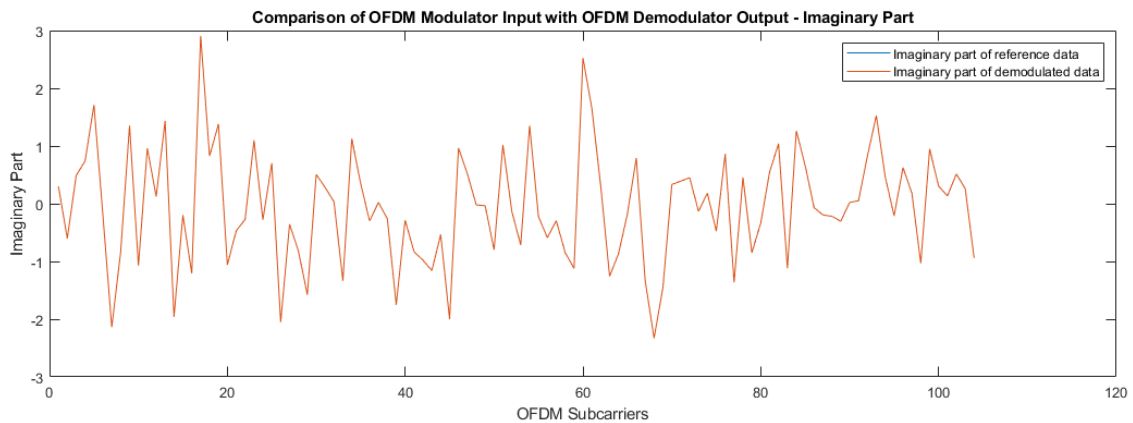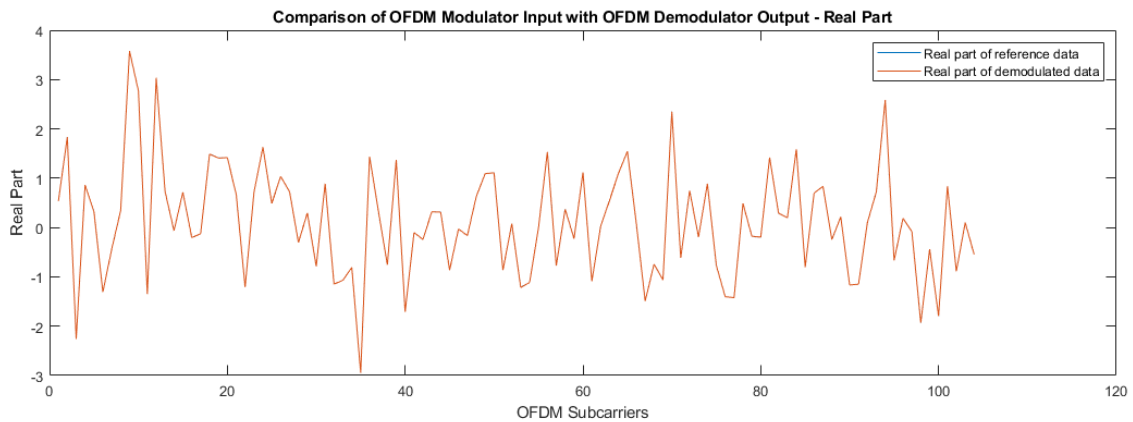

See Also

Blocks
OFDM Demodulator | OFDM Modulator

Polar Encode and Decode of Streaming Samples

This example shows how to simulate the NR Polar Encode and Decode blocks and compare the hardware-optimized results with the results from 5G Toolbox™ functions.

Generate Input Data for Encoder

You must specify the link direction because the coding scheme is different for downlink and uplink messages. Downlink messages are encoded with interleaving and use a CRC length of 24 bits. Uplink messages do not use interleaving, and use a CRC length of 6 ($18 < K < 25$) or 11 bits ($31 < K < 1023$).

This example uses uplink mode with K values greater than 31, so each message must have 11 CRC bits.

Choose a series of input values for **K** and **E**. These values must be valid pairs supported by the 5G NR standard. Generate random frames of input data and add a CRC codeword.

Convert the message frames to streams of Boolean samples and control signals that indicate the frame boundaries. Generate input vectors of **K** and **E** values over time. The example model imports the workspace variables `encSampleIn`, `encCtrlIn`, `encKfi`, `encEfi`, `sampleTime`, and `simTime`.

For this example, the number of invalid cycles between frames is empirically chosen to accommodate the latency of the NR Polar Encoder block for the specified **K** and **E** values. When the values of **K** and **E** are larger than in this example, the number of invalid cycles between frames must be longer. Use the **nextFrame** output signal of the block to determine when the block is ready to accept the start of the next input frame.

```
K = [132; 132; 132; 54];
E = [256; 256; 256; 124];
numFrames = 4;
numCRCBits = 11;
idleCyclesBetweenSamples = 0;
idleCyclesBetweenFrames = 500;
samplesPerCycle = 1;
btwSamples = false(idleCyclesBetweenSamples,1);
btwFrames = false(idleCyclesBetweenFrames,1);

encKfi = [];
encEfi = [];
dataIn = {numFrames};
for ii = 1:numFrames
    msg = randi([0 1],K(ii)-numCRCBits,1);
    msg = nrCRCEncode(msg,'11'); % CRC polynomial is '6' for uplink when 18<K<25, '24C' for down
    encKfi = [encKfi;repmat([fi(K(ii),0,10,0);btwSamples],length(msg),1);btwFrames];
    encEfi = [encEfi;repmat([fi(E(ii),0,14,0);btwSamples],length(msg),1);btwFrames];
    dataIn{1,ii} = logical(msg);
end

[encSampleIn,encCtrlIn] = whdlFramesToSamples(...
    dataIn,idleCyclesBetweenSamples,idleCyclesBetweenFrames,samplesPerCycle);

sampleTime = 1;
simTime = length(encCtrlIn) + K(numFrames)*2;    %#ok<NASGU>
```

Run Encoder Model

The HDL Algorithm subsystem contains the NR Polar Encoder block. Running the model imports the input signal variables from the workspace and returns a stream of polar-encoded output samples and control signals that indicate the frame boundaries. The NR Polar Encoder block in the model has the **Link direction** parameter set to `Uplink`, and accepts **K** and **E** values from input ports. The model exports variables `sampleOut` and `ctrlOut` to the MATLAB workspace.

```matlab
open_system('NRPolarEncodeHDL');
encOut = sim('NRPolarEncodeHDL');
```
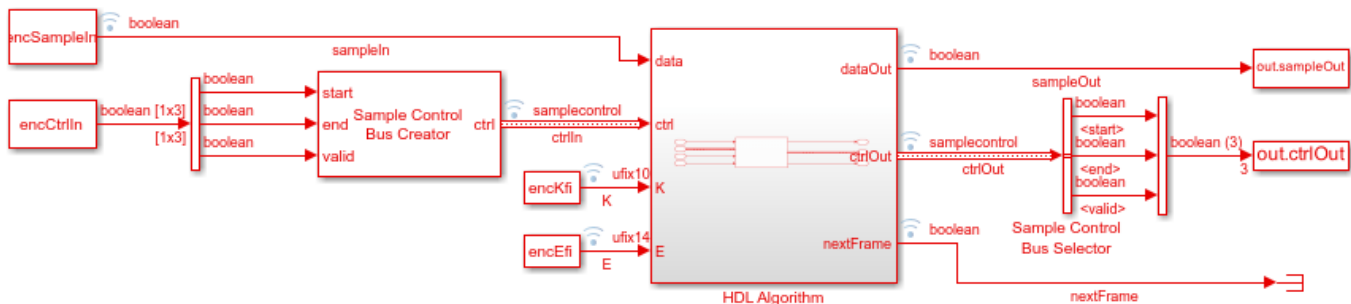

Copyright 2019 The MathWorks, Inc.

Verify Encoder Results

Convert the streaming data back to frames for comparison with the results of the 5G Toolbox™ `nrPolarEncode` function.

```matlab
encHDL = whdlSamplesToFrames(encOut.sampleOut,encOut.ctrlOut);

for ii=1:numFrames
    encRef = nrPolarEncode(double(dataIn{ii}),E(ii),10,false); % last two arguments needed for up
    error = sum(abs(encRef - encHDL{ii}));
    fprintf(['Encoded Frame %d: Behavioral and ' ...
        'HDL simulation differ by %d bits\n'],ii,error);
end
```

```
Maximum frame size computed to be 256 samples.
Encoded Frame 1: Behavioral and HDL simulation differ by 0 bits
Encoded Frame 2: Behavioral and HDL simulation differ by 0 bits
Encoded Frame 3: Behavioral and HDL simulation differ by 0 bits
Encoded Frame 4: Behavioral and HDL simulation differ by 0 bits
```

Generate Input Data for Decoder

Use the encoded data to generate input log-likelihood ratios (LLRs) for the NR Polar Decoder block. Use channel, modulator, and demodulator System objects to add noise to the signal.

Again, create vectors of K and E values, and convert the frames of data to streaming samples with control signals. The example model imports the workspace variables `decSampleIn`, `decCtrlIn`, `decKfi`, `decEfi`, `sampleTime`, and `simTime`.

For this example, the number of invalid cycles between frames is empirically chosen to accommodate the latency of the NR Polar Decoder block for the specified **K** and **E** values. When the values of **K** and

E are larger than in this example, the number of invalid cycles between frames must be longer. Use the **nextFrame** output signal of the block to determine when the block is ready to accept the start of the next input frame.

```
nVar = 0.7;
chan = comm.AWGNChannel('NoiseMethod','Variance','Variance',nVar);
bpskMod = comm.BPSKModulator;
bpskDemod = comm.BPSKDemodulator('DecisionMethod', ...
    'Approximate log-likelihood ratio','Variance',nVar);
% more idle cycles greater list lengths. max 5251 for list 4.
% 1st pkt LL=8 just over 5000, not sure what is max?
% should i make this a more simulink-y example to show how to use the fifo
% with the nextframe signal?
idleCyclesBetweenFrames = 6000;
btwFrames = false(idleCyclesBetweenFrames,1);
decKfi = [];
decEfi = [];
rxLLR = {numFrames};
rxLLRfi = {numFrames};
for ii=1:numFrames
    mod = bpskMod(double(encHDL{ii}));
    rSig = chan(mod);
    rxLLR{1,ii} = bpskDemod(rSig);
    rxLLRfi{1,ii} = fi(rxLLR{1,ii},1,6,0);
    decKfi = [decKfi;repmat([fi(K(ii),0,10,0);btwSamples],length(rSig),1);btwFrames];
    decEfi = [decEfi;repmat([fi(E(ii),0,14,0);btwSamples],length(rSig),1);btwFrames];
end

[decSampleIn,decCtrlIn] = whdlFramesToSamples(...
    rxLLRfi,idleCyclesBetweenSamples,idleCyclesBetweenFrames,samplesPerCycle);

simTime = length(decCtrlIn) + K(numFrames)*2;
```

Run Decoder Model

The HDL Algorithm subsystem contains the NR Polar Decoder block configured to use a list length of eight. The block in the model also has the **Link direction** parameter set to Uplink, and accepts **K** and **E** values from input ports. Running the model imports the input signal variables from the workspace and returns a stream of decoded output samples and control signals that indicate the frame boundaries. The model exports variables sampleOut, ctrlOut, and errOut to the MATLAB workspace. Select the valid values of the errOut signal by using the ctrlOut.valid signal.

```
open_system('NRPolarDecodeHDL');
decOut = sim('NRPolarDecodeHDL');
```
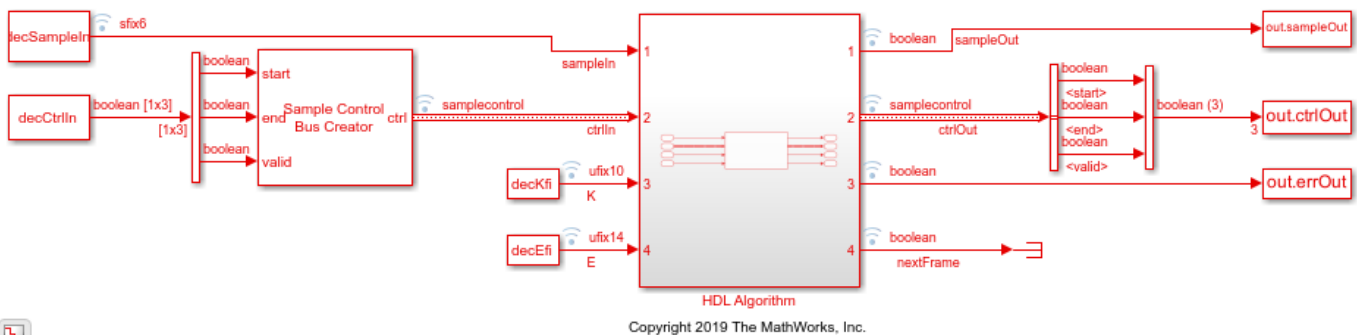

Copyright 2019 The MathWorks, Inc.

Verify Decoder Results

Convert the streaming samples returned from the Simulink model into frames for comparison with the results of the 5G Toolbox™ `nrPolarDecode` function.

The `nrPolarDecode` function returns the decoded message, including 24 recalculated CRC bits. The NR Polar Decoder block returns the decoded message without the CRC bits, and returns the CRC status separately on the **err** port.

The block and function output bits can differ for frames that report a decoding error. The block can return a decoding error in cases when the function successfully decodes the message. The overall decoding performance of the block is very close to that of the function.

```
decHDL = whdlSamplesToFrames(decOut.sampleOut,decOut.ctrlOut);
errHDL = decOut.errOut(decOut.ctrlOut(:,2));

L = 8;
for ii = 1:numFrames
    decRef = nrPolarDecode(rxLLR{1,ii},K(ii),E(ii),L,10,false,numCRCBits); % last three arguments
    [decRef,errRef] = nrCRCDecode(decRef,'11'); % CRC polynomial is '6' for uplink when 18<K<25,
    error = sum(abs(decRef - decHDL{1,ii}));
    fprintf(['Decoded Frame %d: Behavioral and ' ...
        'HDL simulation differ by %d bits\n'],ii,error);
    msg = dataIn{1,ii}(1:(length(dataIn{ii})-numCRCBits));
    loopErr = sum(abs(msg - decHDL{1,ii}));
    fprintf(['The decoded output message from the HDL simulation',...
        ' differs from the input message by %d bits \n'],loopErr);
    errRef = any(errRef);
    if ~errHDL(ii) && ~errRef
        fprintf('HDL and behavioral simulations successfully decoded the message. \n');
    elseif errHDL(ii) && ~errRef
        fprintf(['Behavioral simulation successfully decoded the message,',...
            ' but HDL sim reported a decode error\n']);
    elseif ~errHDL(ii) && errRef
        fprintf(['HDL simulation successfully decoded the message,',...
            ' but behavioral simulation reported a decode error\n']);
    else
        fprintf('HDL and behavioral simulations both reported a decode error. \n');
    end
end
```

```
Maximum frame size computed to be 121 samples.
Decoded Frame 1: Behavioral and HDL simulation differ by 0 bits
The decoded output message from the HDL simulation differs from the input message by 0 bits
HDL and behavioral simulations successfully decoded the message.
Decoded Frame 2: Behavioral and HDL simulation differ by 0 bits
The decoded output message from the HDL simulation differs from the input message by 0 bits
HDL and behavioral simulations successfully decoded the message.
Decoded Frame 3: Behavioral and HDL simulation differ by 0 bits
The decoded output message from the HDL simulation differs from the input message by 0 bits
HDL and behavioral simulations successfully decoded the message.
Decoded Frame 4: Behavioral and HDL simulation differ by 0 bits
```

The decoded output message from the HDL simulation differs from the input message by 0 bits
HDL and behavioral simulations successfully decoded the message.

See Also

NR Polar Encoder | NR Polar Decoder | nrPolarEncode | nrPolarDecode

NR CRC Encode and Decode Streaming Data

This example shows how to use the NR CRC Encoder and NR CRC Decoder Simulink® blocks and compare the hardware-optimized results with the results from the 5G Toolbox™ functions `nrCRCEncode` (5G Toolbox) and `nrCRCDecode` (5G Toolbox), respectively. These blocks support scalar and vector inputs. The NR CRC Encoder and NR CRC Decoder blocks support hardware code generation.

Generate Input Data

Generate random frames of input data and a control signal that indicates the frame boundaries. The frame gap accommodates the latency of the NR CRC Encoder block.

```matlab
CRCType = 'CRC24A';
numFrames = 4;
scalar = true;                                          % true for scalar inputs and false

parallel = false;                                       % true for parallel architecture an
                                                        % serial architecture
msg = {numFrames};
dataIn = [];
encStartIn = [];
encEndIn = [];
encValidIn = [];
[poly,crcLen] = NRCRCEncodeAndDecoderHDLInitScript(CRCType);
if parallel
    listN = divisors(crcLen);                           % Factors of length of CRC polynomi
    dataWidth = randsrc(1,1,listN(2:end));
else
    dataWidth = 1;
end
frameGap = 120;                                         % Frame gap selected based on CRCTy
for ii = 1:numFrames
    len = randsrc(1,1,1:1000);
    frameLen = len*dataWidth;
    msg{ii} = randi([0 1],1,frameLen);

    % Generate data based on the selected dataWidth
    if scalar
        data = reshape(msg{ii},dataWidth,len);
        encIn = zeros(1,size(data,2));
        for i = 1:size(data,2)
            encIn(i) = bit2int(data(:,i).',length(data(:,i))).'; %#ok<*SAGROW>
        end
        dataIn  = fi([dataIn encIn zeros(size(encIn,1),frameGap)],0,dataWidth,0);
    else
        encIn = reshape(msg{ii},dataWidth,len); %#ok<*UNRCH>
        dataIn = logical([dataIn encIn zeros(size(encIn,1),frameGap)]);
    end

    encStartIn = logical([encStartIn 1 zeros(1,len-1) zeros(1,frameGap)]);
    encEndIn   = logical([encEndIn zeros(1,len-1) 1 zeros(1,frameGap)]);
    encValidIn = logical([encValidIn ones(1,len) zeros(1,frameGap)]);
end

encSampleIn = timeseries(dataIn');
```

```
sampleTime = 1;
simTime = length(encValidIn);
```

Run the Model

The `HDLNRCRCEncodeDecode` subsystem contains `HDL NR CRC Encoder` and `HDL NR CRC Decoder` subsystems that contain NR CRC Encoder and NR CRC Decoder blocks, respectively. Running the model imports the input signal variables `encSampleIn`, `encStartIn`, `encEndIn`, and `encValidIn` and exports variables `encSampleOut` and `encCtrlOut` to the MATLAB® workspace.

```
open_system('NRCRCEncodeAndDecodeHDLModel');
set_param('NRCRCEncodeAndDecodeHDLModel/HDLNRCRCEncodeDecode/HDL NR CRC Encoder/NR CRC Encoder',
set_param('NRCRCEncodeAndDecodeHDLModel/HDLNRCRCEncodeDecode/HDL NR CRC Decoder/NR CRC Decoder',
modelOut = sim('NRCRCEncodeAndDecodeHDLModel');
```
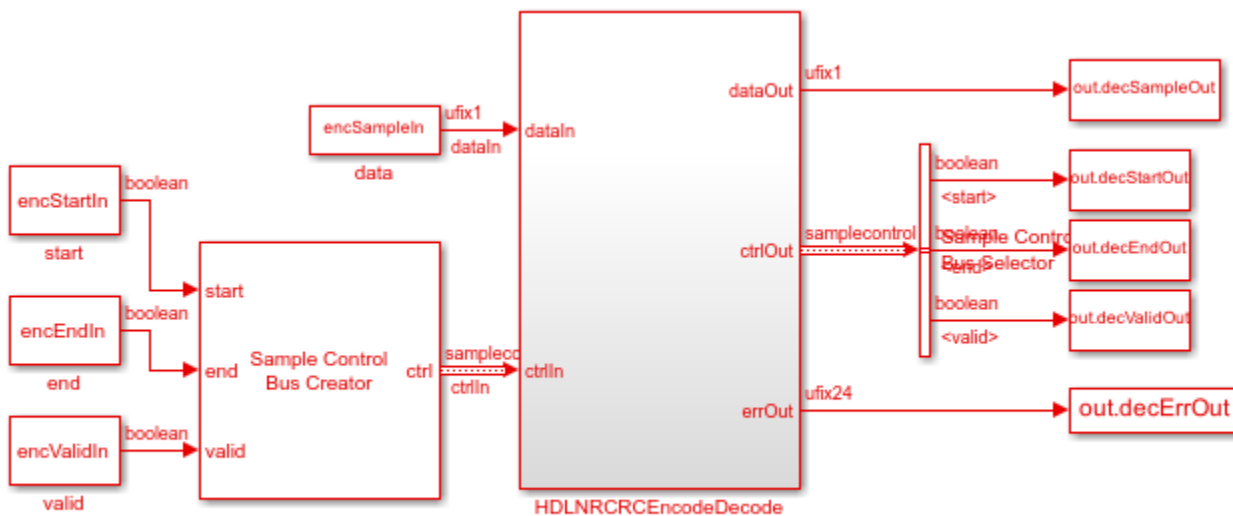

Copyright 2021 The MathWorks, Inc.

Verify Encoder Results

The `HDL NR CRC Encoder` subsystem contains the NR CRC Encoder block. Convert the streaming data output of the NR CRC Encoder block to frames, and then compare the output frames with the output of the `nrCRCEncode` 5G Toolbox function.

```
encOut = squeeze(modelOut.encSampleOut.Data);
startIdx = find(modelOut.encCtrlOut.start.Data);
endIdx = find(modelOut.encCtrlOut.end.Data);
encValidOut = squeeze(modelOut.encCtrlOut.valid.Data);
vector = ~scalar && parallel;

for ii = 1:numFrames
    refEncBits{ii} = nrCRCEncode(msg{ii}',poly);
    % Extract actual encoded bits from output
    idx = startIdx(ii):endIdx(ii);
    if (vector) % For vector inputs
        encBits = encOut(:,idx);
        encBits = encBits(:,encValidOut(idx));
        actEncBits{ii} = encBits(:);
```

```
    else
        encBits = encOut(idx);
        encBits = encBits(encValidOut(idx));
        encBits = dec2bin(encBits,dataWidth)-'0';
        actEncBits{ii} = reshape(encBits',length(refEncBits{ii}),1);
    end
    error = sum(abs(refEncBits{ii}-double(actEncBits{ii}))));
    fprintf(['CRC-encoded frame %d: Behavioral and ' ...
        'HDL simulation differ by %d bits\n'],ii,error);
end
```

```
CRC-encoded frame 1: Behavioral and HDL simulation differ by 0 bits
CRC-encoded frame 2: Behavioral and HDL simulation differ by 0 bits
CRC-encoded frame 3: Behavioral and HDL simulation differ by 0 bits
CRC-encoded frame 4: Behavioral and HDL simulation differ by 0 bits
```

Verify Decoder Results

The `HDL NR CRC Decoder` subsystem contains the NR CRC Decoder block. The `HDL NR CRC Encoder` subsystem outputs are provided as an input to the `HDL NR CRC Decoder` subsystem. The `HDL NR CRC Decoder` subsystem exports a stream of decoded output samples *decSampleOut* and *decErrOut* along with a control signal *decCtrlOut* to the MATLAB workspace. Compare them with the output of the `nrCRCDecode` function.

```
dataOut = squeeze(modelOut.decSampleOut.Data);
errOut = squeeze(modelOut.decErrOut.Data);
startIdx = find(modelOut.decStartOut.Data);
endIdx = find(modelOut.decEndOut.Data);
validOut = squeeze(modelOut.decValidOut.Data);
```

```
for ii = 1:numFrames
    [refDecBits{ii},refErr{ii}] = nrCRCDecode(double(actEncBits{ii}),poly);
    % Extract actual decoded bits from output
    idx = startIdx(ii):endIdx(ii);
    if (vector) % For vector inputs
        dataOutTmp = dataOut(:,idx);
        validOutTmp = validOut(:,idx);
        decBits = dataOutTmp(:,validOutTmp);
        actDecBits{ii} = decBits(:);
    else
        dataOutTmp = dataOut(idx);
        validOutTmp = validOut(idx);
        decBits = dataOutTmp(validOutTmp);
        decBits = dec2bin(decBits,dataWidth) - '0';
        actDecBits{ii} = reshape(decBits',length(refDecBits{ii}),1);
    end
    actErr{ii} = errOut(endIdx(ii));
    error_data = sum(abs(refDecBits{ii} - double(actDecBits{ii}))));
    error_err = double(refErr{ii}) - double(actErr{ii});
    fprintf(['CRC-decoded frame %d: Behavioral and ' ...
        'HDL simulation differ by %d bits and %d errors\n'],ii,error_data,error_err);
end
```

```
CRC-decoded frame 1: Behavioral and HDL simulation differ by 0 bits and 0 errors
CRC-decoded frame 2: Behavioral and HDL simulation differ by 0 bits and 0 errors
```

```
CRC-decoded frame 3: Behavioral and HDL simulation differ by 0 bits and 0 errors
CRC-decoded frame 4: Behavioral and HDL simulation differ by 0 bits and 0 errors
```

See Also

Blocks
NR CRC Encoder | NR CRC Decoder

Functions
nrCRCEncode | nrCRCDecode

Equalize OFDM Data Using Channel Estimates

This example shows how to use the OFDM Equalizer block to equalize data subcarriers using channel estimates. In this example, the model uses the first frame to estimate the channel, stores the estimates, and equalizes the remaining frames using the stored channel estimates. The `HDL Algorithm` subsystem in this example supports HDL code generation.

Set Input Data Parameters

Set up workspace variables for the model to use. You can modify these values according to your requirements.

```
rng('default');
numFrames = 6;                                      % Number of frames
numOFDMSymPerFrame = 140;                            % Number of OFDM symbols per frame
maxLenChEstiPerSym = 14400;                          % Maximum length of channel estimates per sym
numSubCarPerSym = 72;                                % Number of subcarriers per OFDM symbol
hEstLen = numSubCarPerSym * numOFDMSymPerFrame;      % Channel estimate length
totNumOFDMSymbols = numFrames * numOFDMSymPerFrame;  % Total number of OFDM symbols
```

Generate Sinusoidal Input Data Subcarriers

Use the `hEstLen` and `numOFDMSym` variables to generate complex sinusoidal input data subcarriers with their real and imaginary parts generated separately. Plot the input as a real part and an imaginary part using separate plots.

```
dataInGrid = zeros(numSubCarPerSym,totNumOFDMSymbols);
for subCarCount = 0:numSubCarPerSym-1
    for numOFDMSymCount = 0:totNumOFDMSymbols-1
        realXgain = 1 + .2*sin(2*pi*subCarCount/numSubCarPerSym);
        realYgain = 1 + .5*sin(2*pi*numOFDMSymCount/numOFDMSymPerFrame);
        imagXgain = 1 + .3*sin(2*pi*subCarCount/numSubCarPerSym);
        imagYgain = 1 + .4*sin(2*pi*numOFDMSymCount/numOFDMSymPerFrame);
        dataInGrid(subCarCount+1,numOFDMSymCount+1) = realXgain*realYgain + 1i*(imagXgain*imagYga
    end
end
validIn = true(1,length(dataInGrid(:)));

% Normalize data subcarriers to make signal power unity
dataInGrid = dataInGrid./sqrt(mean(abs(dataInGrid).^2,'all'));

figure(1);
surf(real(dataInGrid))
xlabel('OFDM Symbols')
ylabel('Subcarriers')
zlabel('Magnitude')
title('Input Data Grid (Real Part)')

figure(2);
surf(imag(dataInGrid))
xlabel('OFDM Symbols')
ylabel('Subcarriers')
zlabel('Magnitude')
title('Input Data Grid (Imaginary Part)')
```

Input Data Grid (Real Part)

Input Data Grid (Imaginary Part)
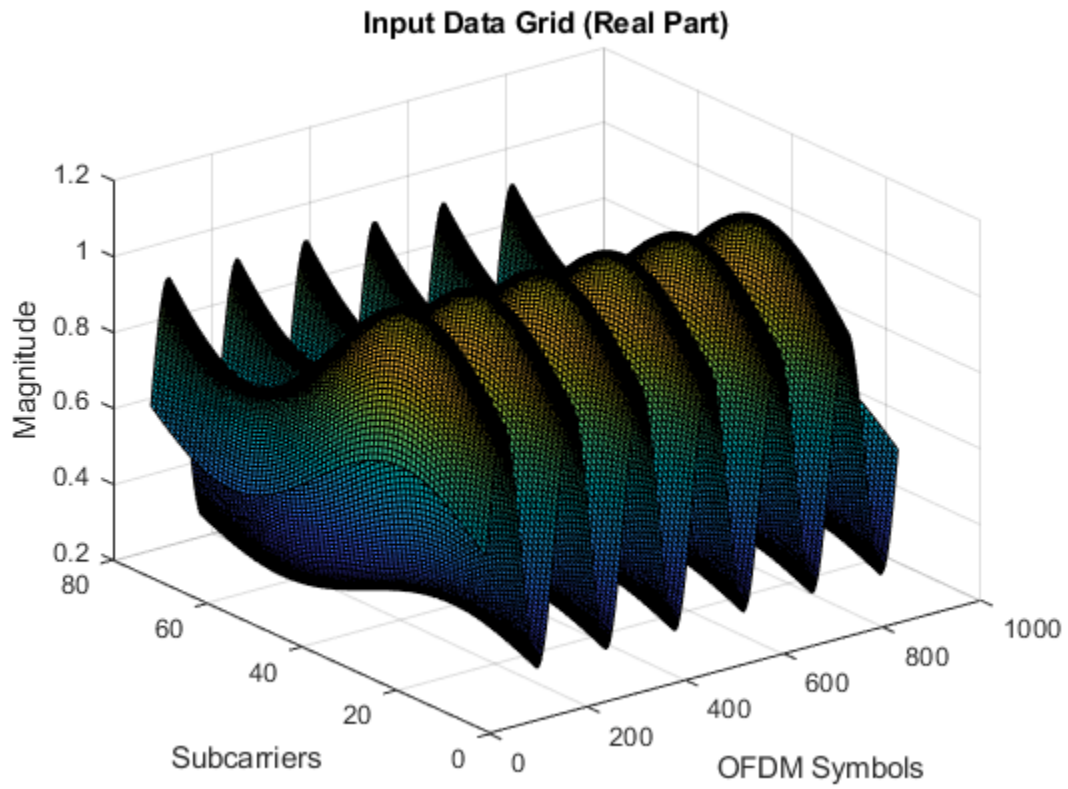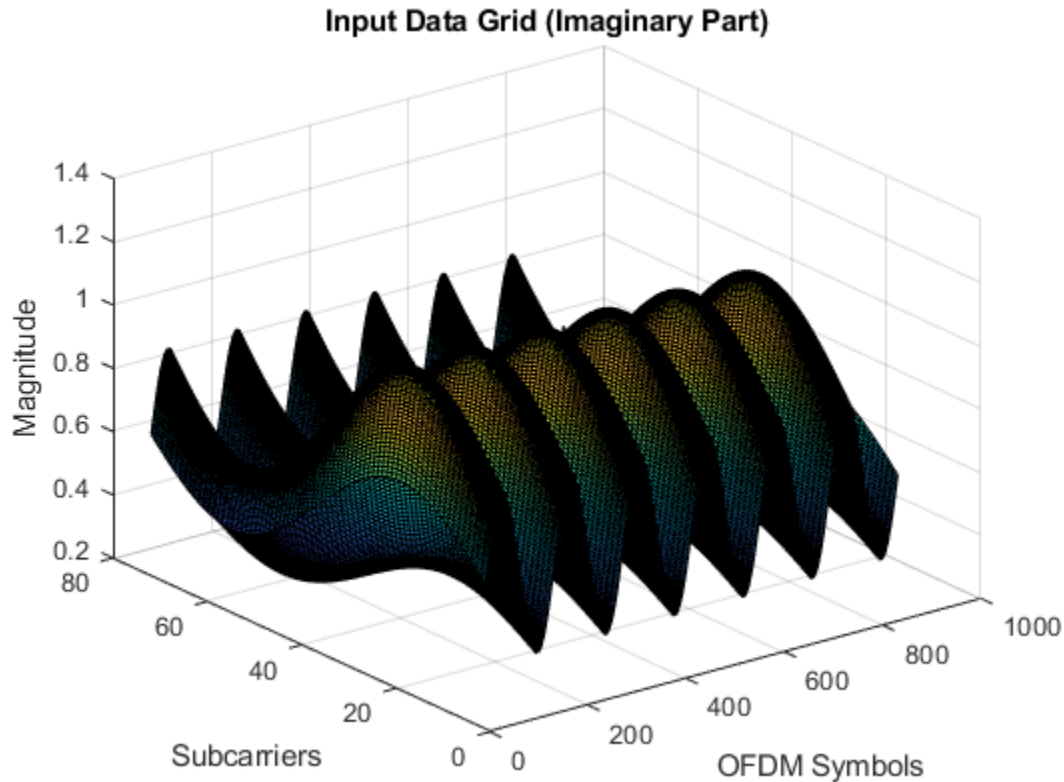
Generate Channel Estimates using MATLAB® Function

Generate the reference data subcarriers using the variables numOFDMSymToBeAvg, interpolFac, and numScPerSym. Use the channelEstReferenceForEqualizer function to generate the channel estimates hEstIn.

```
numOFDMSymToBeAvg = 1;                                                      % Number of OFDM symbols to be
interpolFac = 1;                                                           % Interpolation factor
dataInForChannelEsti = dataInGrid(:,1:numOFDMSymPerFrame);
validInForChanEsti = validIn(1:numSubCarPerSym*numOFDMSymPerFrame);

numScPerSymIn = numSubCarPerSym*true(1,length(dataInForChannelEsti(:)));

refDataIn = randsrc(size(dataInForChannelEsti(:),1),size(dataInForChannelEsti(:),2),[1 1]);
refValidIn = boolean(zeros(1,numOFDMSymPerFrame*numSubCarPerSym));
startRefValidIndex = randi(interpolFac,1,1);
for numOFDMSymCount = 1:numOFDMSymPerFrame
    refValidIn(startRefValidIndex+(numOFDMSymCount-1)*numSubCarPerSym:interpolFac:numSubCarPerSym
end

dataOut1 = channelEstReferenceForEqualizer( ...
    numOFDMSymToBeAvg,interpolFac,numSubCarPerSym,numOFDMSymPerFrame, ...
    dataInForChannelEsti(:),validInForChanEsti,refDataIn,refValidIn,numScPerSymIn);
matlabOut = dataOut1(:);
hEstIn = zeros(numel(matlabOut)*numSubCarPerSym*numOFDMSymToBeAvg,1);
for ii= 1:numel(matlabOut)
    loadArray = [matlabOut(ii).dataOut; repmat(matlabOut(ii).dataOut,[numOFDMSymToBeAvg-1 1]); ze
```

```
        shiftArray = circshift(loadArray,(ii-1)*numSubCarPerSym*numOFDMSymToBeAvg);
        hEstIn =  hEstIn + shiftArray;
    end

    % Repeat hEstIn for dataIn generation
    hEstInForDataIn = repmat(hEstIn,numFrames,1);

    % Normalize channel estimates to make signal power unity
    hEstIn = hEstIn./sqrt(mean(abs(hEstIn).^2,'all'));
    hEstIn = [hEstIn; hEstIn(end)*ones((hEstLen*((totNumOFDMSymbols/numOFDMSymPerFrame)-1)),1)];

    % Generate noise samples
    n = (1/sqrt(2))*(randn(length(dataInGrid(:)),1)+1i*randn(length(dataInGrid(:)),1)); % white gauss

    SNR = 40;
    % Calculate noise variance
    nVar = (10^(-SNR/10));
    noiseVarIn = (10^(-SNR/10))*ones(1,length(dataInGrid(:)));

    modelname = 'genhdlOFDMEqualizerModel';
    open_system(modelname);

    EqMdUsed = get_param('genhdlOFDMEqualizerModel/HDL Algorithm/OFDM Equalizer','EqualizationMethod
    if strcmp(EqMdUsed,'ZF')
        % ZF equalization
        dataIn = hEstInForDataIn.*dataInGrid(:);
    else
        % MMSE equalization
        dataIn = hEstInForDataIn.*dataInGrid(:) + (n.*(sqrt(nVar)))./(sqrt(var(n)));
    end
    % Generate signal with channel estimate length per symbol
    hEstLenIn = hEstLen*true(1,length(dataInGrid(:)));
    loadhEst = logical([1 zeros(1,length(dataInGrid(:))-1)]);
    resetSig = false(1,length(dataInGrid(:)));
```
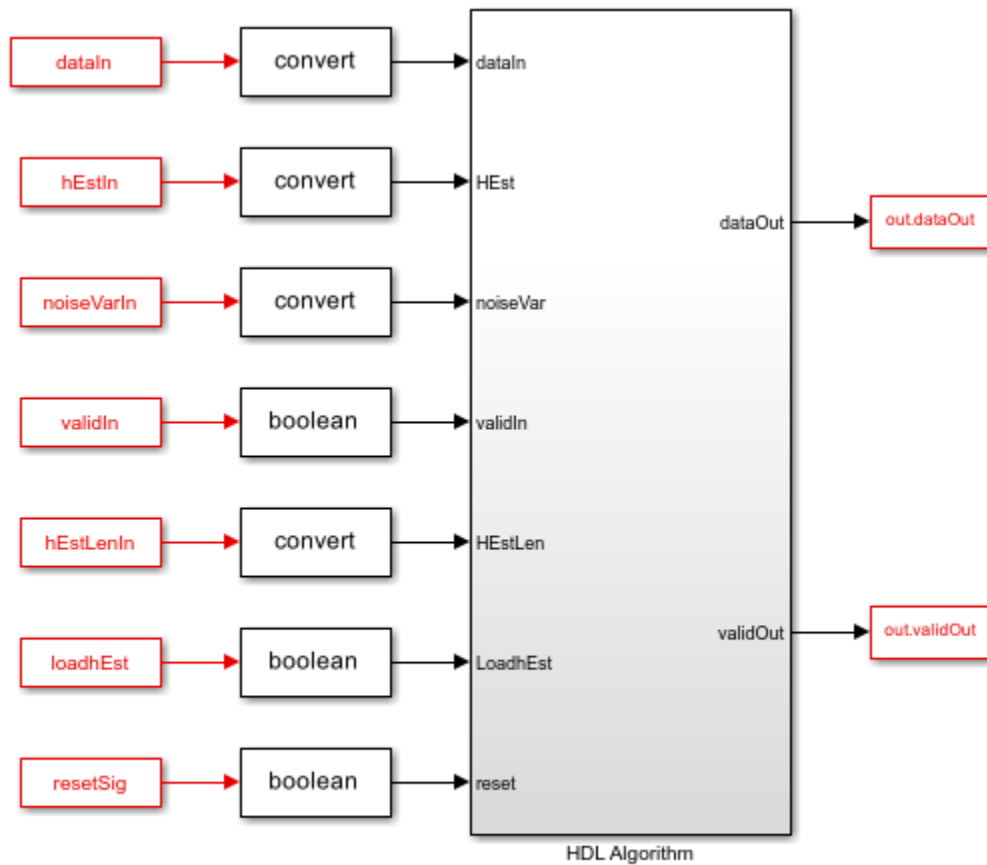
Copyright 2021 The MathWorks, Inc.

Run Simulink® Model

Running the model imports the input signal variables from the MATLAB workspace to the OFDM Equalizer block in the model.

```
out = sim(modelname);
```

Export Stream of Equalized Data from Simulink to MATLAB Workspace

Export the output of the OFDM Equalizer block to the MATLAB workspace. Plot the real part and imaginary part of the exported block output.

```
simOut = out.dataOut.Data(out.validOut.Data);
N = length(simOut)-mod(length(simOut),numSubCarPerSym);
temp = simOut(1:N);
EqualizerSimOut = reshape(temp,numSubCarPerSym,length(temp)/numSubCarPerSym);

figure(3);
surf(real(EqualizerSimOut))
xlabel('OFDM Symbols')
ylabel('Subcarriers')
zlabel('Magnitude')
```

```
title('OFDM Equalizer Output (Real Part)')

figure(4);
surf(imag(EqualizerSimOut))
xlabel('OFDM Symbols')
ylabel('Subcarriers')
zlabel('Magnitude')
title('OFDM Equalizer Output (Imaginary Part)')
```


OFDM Equalizer Output (Real Part)

OFDM Equalizer Output (Imaginary Part)
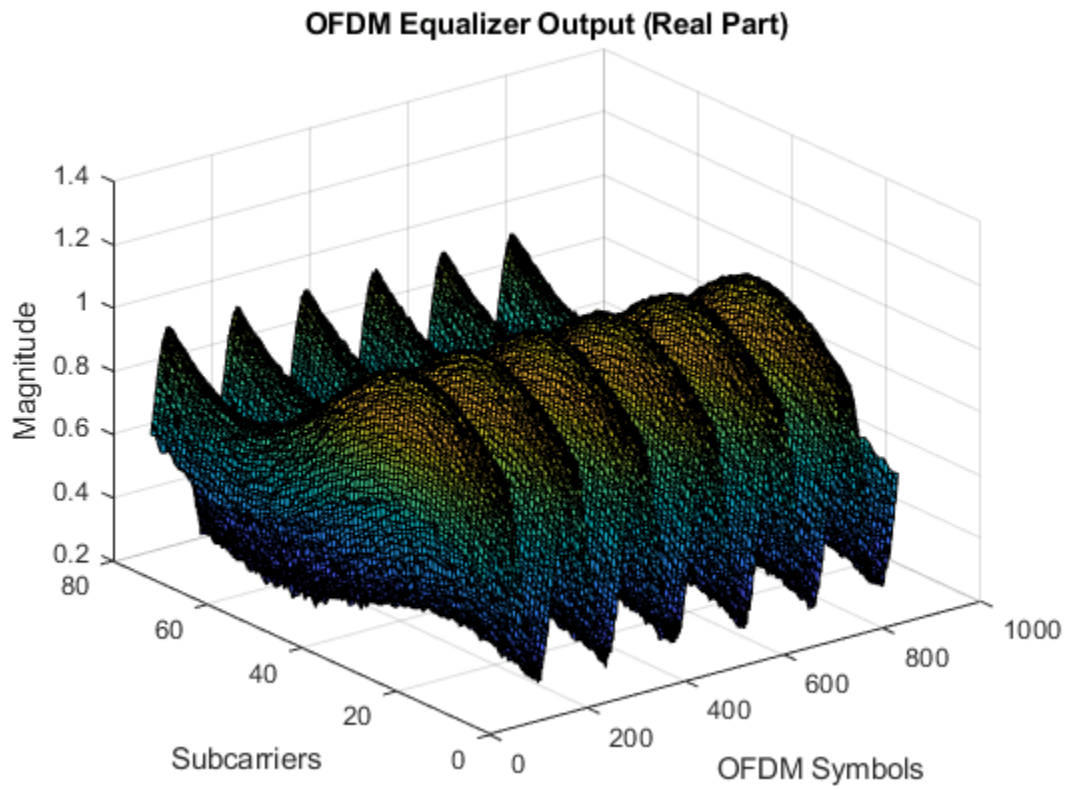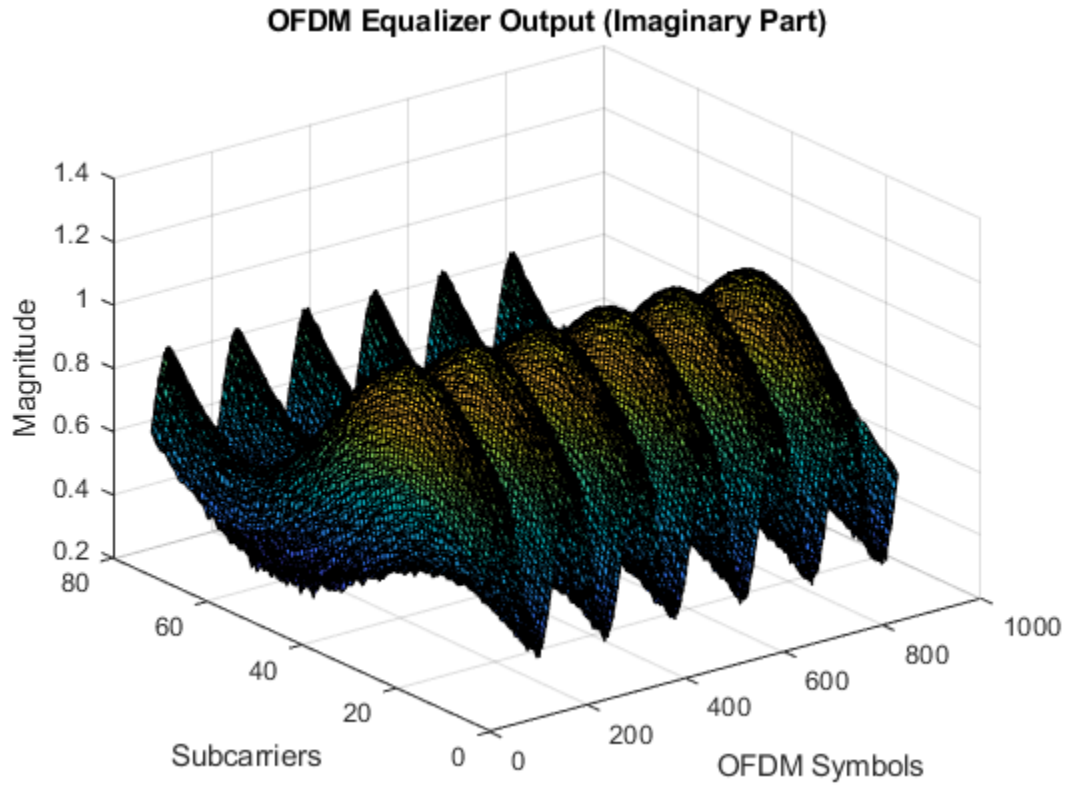
Perform Equalization Using MATLAB

Equalize the channel with equalization equations by using MATLAB.

```matlab
if strcmp(EqMdUsed, 'ZF')
    % ZF equalization
    matOut = dataIn./hEstInForDataIn;
else
    % MMSE equalization
    matOut = (1./(conj(hEstInForDataIn).*hEstInForDataIn+nVar)).*(conj(hEstInForDataIn)).*dataIn
end
```

Compare Simulink Block Output with MATLAB Output

Compare the OFDM Equalizer block output with the MATLAB output. Plot the output comparison as a real part and an imaginary part using separate plots.

```matlab
figure('units','normalized','outerposition',[0 0 1 1])
subplot(2,1,1)
plot(real(matOut(:)));
hold on;
plot(real(simOut(:)));
grid on
legend('MATLAB reference output','Simulink block output')
xlabel('Sample Index')
ylabel('Magnitude')
title('Comparison of Simulink Block and MATLAB Function (Real Part)')
```

```
subplot(2,1,2)
plot(imag(matOut(:)));
hold on;
plot(imag(simOut(:)));
grid on
legend('MATLAB reference output','Simulink block output')
xlabel('Sample Index')
ylabel('Magnitude')
title('Comparison of Simulink Block and MATLAB Function (Imaginary Part)')

sqnrRealdB = 10*log10(double(var(real(simOut(:)))/abs(var(real(simOut(:)))-var(real(matOut(:)))))
sqnrImagdB = 10*log10(double(var(imag(simOut(:)))/abs(var(imag(simOut(:)))-var(imag(matOut(:)))))

fprintf('\n OFDM Equalizer \n SQNR of real part: %.2f dB',sqnrRealdB);
fprintf('\n SQNR of imaginary part: %.2f dB\n',sqnrImagdB);


 OFDM Equalizer
 SQNR of real part: 36.56 dB
 SQNR of imaginary part: 42.16 dB
```

Comparison of Simulink Block and MATLAB Function (Real Part)

Comparison of Simulink Block and MATLAB Function (Imaginary Part)
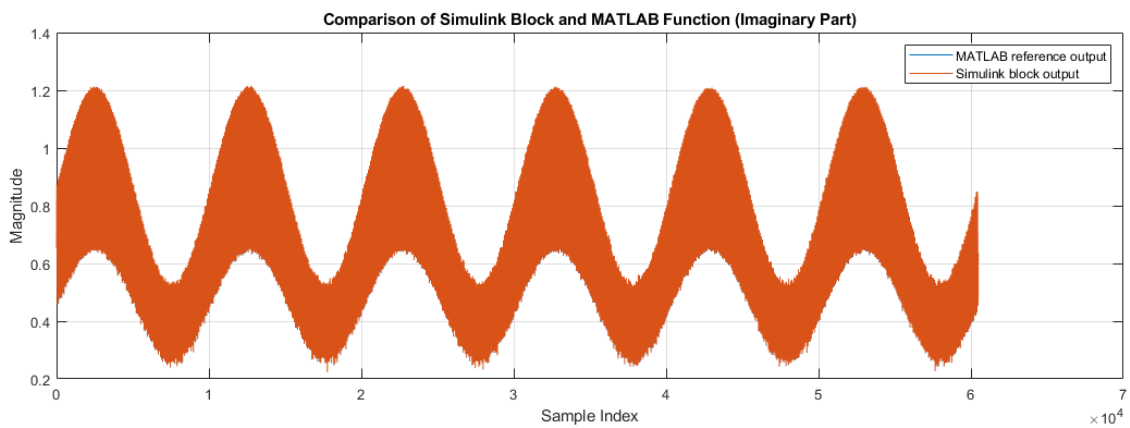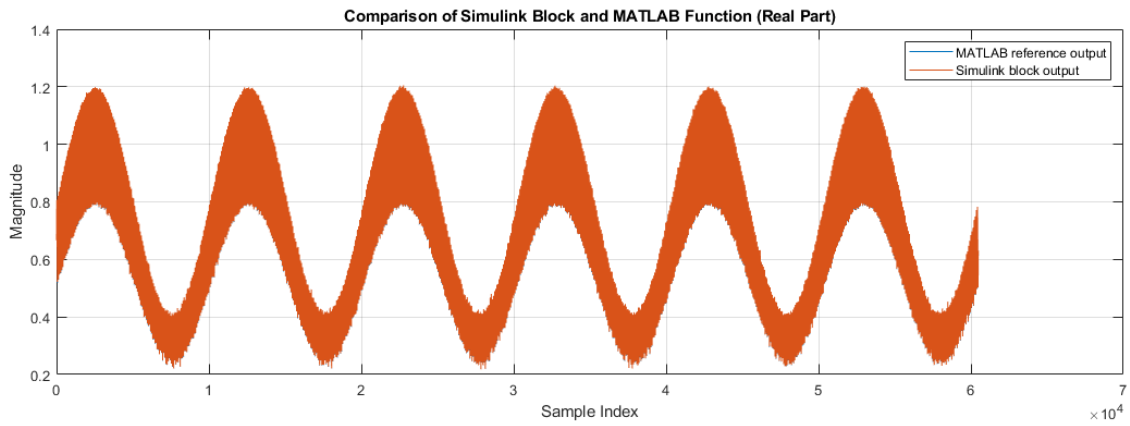
See Also

Blocks
OFDM Channel Estimator

Functions
nrEqualizeMMSE | lteEqualizeMMSE | lteEqualizeZF

3-95

LDPC Decode 5G NR Streaming Data for Multiple Code Rates with Early Termination

This example shows how to use multiple code rates and early termination criteria features in the NR LDPC Decoder Simulink® block. The input to the block is generated using the nrLDPCEncode (5G Toolbox) MATLAB® function and the output of the block is compared with the input of the function. In this example, you can select either the min-sum or normalized min-sum algorithm for the decoding operation.

Generate Input Data

Choose a series of input values for the base graph number (bgn) and liftingSize according to the 5G new radio (NR) standard and generate the corresponding input vectors for those values. Use the encoded data from the nrLDPCEncode function to generate input log-likelihood ratio (LLR) values for the NR LDPC Decoder block. Use channel, modulator, and demodulator System objects to add noise to the signal. Again, create vectors of bgn and liftingSize, and then convert the frames of data to LLRs with a control signal that indicates the frame boundaries. The decFrameGap accommodates the latency of the NR LDPC Decoder block for base graph number, liftingSize, and number of iterations. Use the **nextFrame** output signal to determine when the block is ready to accept the start of the next input frame.

```matlab
bgn = [1; 0; 0; 1];
liftingSize = [4; 384; 144; 208];
numRows = [6; 38; 24; 10];
numFrames = 4;
serial = false;  % true for serial inputs and false for parallel inputs

msg = {numFrames};
K = [];
N = [];
for ii = 1:numFrames
    if bgn(ii) == 0
        K(ii) = 22;
    else
        K(ii) = 10;
    end
    N(ii) = numRows(ii) + K(ii)-2;
    frameLen = liftingSize(ii)*K(ii);
    msg{ii} = randi([0 1],frameLen,1);

    encTmp = nrLDPCEncode(msg{ii},bgn(ii)+1);
    encOut{ii} = encTmp(1:N(ii)*liftingSize(ii));
end

nVar = 0.5;
chan = comm.AWGNChannel('NoiseMethod','Variance','Variance',nVar);
bpskMod = comm.BPSKModulator;
bpskDemod = comm.BPSKDemodulator('DecisionMethod', ...
    'Approximate log-likelihood ratio','Variance',nVar);

algo = 'Min-sum'; % 'Min-sum' or 'Normalized min-sum'
if strcmpi(algo,'Min-sum')
    alpha = 1;
else
```

```matlab
        alpha = 0.75;
end

numIter = 8;
decbgnIn = [];
decliftingSizeIn = [];
rxLLR = {numFrames};
decSampleIn = [];
decStartIn = [];
decEndIn = [];
decValidIn = [];
decnumRows = [];

for ii=1:numFrames
    mod = bpskMod(double(encOut{ii}));
    rSig = chan(mod);
    rxLLR{ii} = fi(bpskDemod(rSig),1,4,0);

    if serial
        len = N(ii)*liftingSize(ii); %#ok<*UNRCH>
        decFrameGap = numIter*7000 + liftingSize(ii)*K(ii);
    else
        len = N(ii)*ceil(liftingSize(ii)/64);
        decFrameGap = numIter*1200;
    end

    decIn = ldpc_dataFormation(rxLLR{ii}',liftingSize(ii),N(ii),serial);

    decSampleIn = [decSampleIn decIn zeros(size(decIn,1),decFrameGap)]; %#ok<*AGROW>
    decStartIn = logical([decStartIn 1 zeros(1,len-1) zeros(1,decFrameGap)]);
    decEndIn = logical([decEndIn zeros(1,len-1) 1 zeros(1,decFrameGap)]);
    decValidIn = logical([decValidIn ones(1,len) zeros(1,decFrameGap)]);
    decbgnIn = logical([decbgnIn repmat(bgn(ii),1,len) zeros(1,decFrameGap)]);
    decliftingSizeIn = uint16([decliftingSizeIn repmat(liftingSize(ii),1,len) zeros(1,decFrameGap
    decnumRows = fi([decnumRows repmat(numRows(ii),1,len) zeros(1,decFrameGap)],0,6,0);
end

decSampleIn = timeseries(fi(decSampleIn',1,4,0));
sampleTime = 1;

simTime = length(decValidIn);
```

Run Simulink Model

The HDL Algorithm subsystem contains the NR LDPC Decoder block. Running the model imports the input signal variables decSampleIn, decStartIn, decEndIn, decValidIn, decbgnIn, decliftingSizeIn, numIter, sampleTime, and simTime and exports a stream of decoded output samples sampleOut along with a control signal ctrlOut to the MATLAB workspace.

```matlab
open_system('NRLDPCDecoderCodeRateHDL');
if alpha ~= 1
    set_param('NRLDPCDecoderCodeRateHDL/HDL Algorithm/NR LDPC Decoder','Algorithm','Normalized m
else
    set_param('NRLDPCDecoderCodeRateHDL/HDL Algorithm/NR LDPC Decoder','Algorithm','Min-sum');
end
decOut = sim('NRLDPCDecoderCodeRateHDL');
```
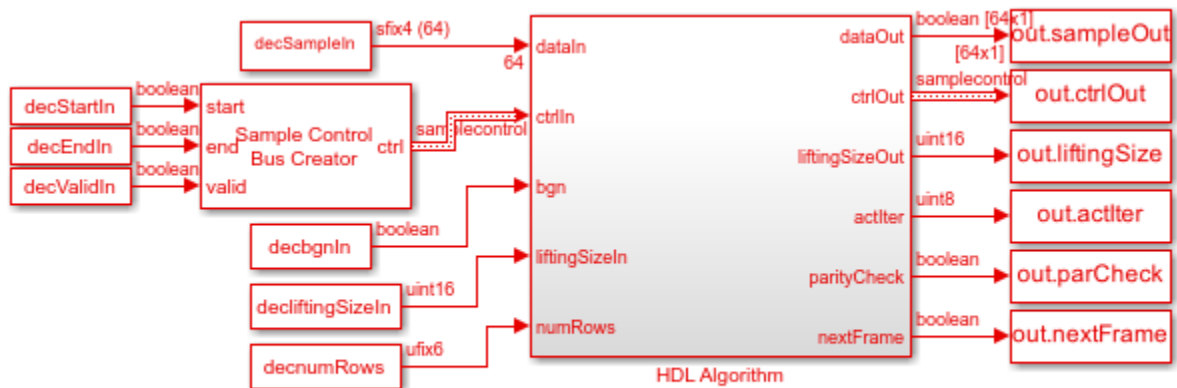
Copyright 2020 - 2022 The MathWorks, Inc.

Compare Simulink Block Output with MATLAB Function Input

Convert the streaming data output of the NR LDPC Decoder block to frames and then compare the frames with the input of the nrLDPCEncode function.

```
startIdx = find(decOut.ctrlOut.start.Data);
endIdx = find(decOut.ctrlOut.end.Data);

for ii = 1:numFrames
    decHDL{ii} = ldpc_dataExtraction(decOut.sampleOut.Data,liftingSize(ii),startIdx(ii),endIdx(ii
    error = sum(abs(double(msg{ii})-decHDL{ii}));
    fprintf(['Decoded frame %d: Output data differs by %d bits\n'],ii,error);
    iter_tmp = squeeze(decOut.actIter.Data);
    actIter{ii} = iter_tmp(startIdx(ii));
    fprintf(['Actual iterations taken to decode the frame: %d \n'],actIter{ii});
end
```

```
Decoded frame 1: Output data differs by 0 bits
Actual iterations taken to decode the frame: 2
Decoded frame 2: Output data differs by 0 bits
Actual iterations taken to decode the frame: 2
Decoded frame 3: Output data differs by 0 bits
Actual iterations taken to decode the frame: 2
Decoded frame 4: Output data differs by 0 bits
Actual iterations taken to decode the frame: 3
```

See Also

Blocks
NR LDPC Decoder

Functions
nrLDPCEncode

Decode and Recover Message from RS Codeword Using CCSDS Standard

This example shows how to use the CCSDS RS Decoder block to decode and recover a message from a Reed-Solomon (RS) codeword according to the Consultative Committee for Space Data Systems (CCSDS) standard. Generate and encode a set of random inputs and then provide them as input to the `ccsdsRSDecode` (Satellite Communications Toolbox) function and the CCSDS RS Decoder block by adding errors. Compare the output of the CCSDS RS Decoder block with the output of the `ccsdsRSDecode` function. The example model supports HDL code generation for the `HDL CCSDS RS Decoder` subsystem.

Set Up Input Data Parameters

Set up workspace variables for the model to use. You can modify these variable values according to your requirements. The block supports a fixed codeword length of 255.

```
k = 239;          % Message length 223 or 239
s = k;            % Shortened message length ranges from 1 to k
i = 4;            % Interleaving depth 1, 2, 3, 4, 5, or 8
numFrames = 3;    % Number of input frames
numErrors = 16;   % Maximum number of correctable errors allowed in the input frame is (255-k)*i/2
```

Generate Random Input Samples

Generate random samples using the specified message length, shortened message length, and interleaving depth. Encode the random samples using the `ccsdsRSEncode` function, and then insert `numErrors` number of errors at random locations in the encoded samples.

```
% Generate random message samples
msg = randi([0 255],s*i,1);

% Encode message samples
encoderOut = ccsdsRSEncode(msg,k,i,s);

% Insert errors in encoded output
errorLoc = randi([1 (255-k+s)*i],numErrors,1);
errorVal = randi([1 255],numErrors,1);
chOut = encoderOut;
chOut(errorLoc) = errorVal;
```

Decode Encoded Data Using MATLAB® Function

Decode the encoded data containing errors using the `ccsdsRSDecode` function.

```
[refOutput,refNErr] = ccsdsRSDecode(chOut,k,i,s);
refOutput = repmat(refOutput,numFrames,1);
refNErr = repmat(refNErr,numFrames,1);
```

Decode Encoded Data Using Simulink® Block

Decode the encoded data containing errors using the CCSDS RS Decoder block. Running the model imports the input signal variables from the MATLAB workspace to the CCSDS RS Decoder block in the model.

```
% Set frame gap between input frames
if(k == 223 && (i == 1 || i == 2))
```

```
    frameGap = 602-(255*i);
else
    frameGap = 0;
end

% Assign inputs to model
dataIn = repmat([chOut; zeros((k-s)*i,1); zeros(frameGap,1)],numFrames,1);
startIn = repmat([true; false(255*i -1,1); false(frameGap,1)],numFrames,1);
endIn = repmat([false((255-k+s)*i -1,1); true; false((k-s)*i,1); ...
    false(frameGap,1)],numFrames,1);
validIn = repmat([true((255-k+s)*i,1); false((k-s)*i,1); ...
    false(frameGap,1)],numFrames,1);

numOutputSamples = k*i;
stopTime = (3065 + numOutputSamples)*numFrames; % Maximum latency of the
                                                % block is 3065 clock cycles

% Run the Simulink model
model_name = 'HDLCCSDSRSDecoder';
open_system(model_name);
set_param([model_name '/HDL CCSDS RS Decoder/CCSDS RS Decoder'], ...
    'MessageLength',num2str(k),'InterleavingDepth',num2str(i));
sim(model_name);
```
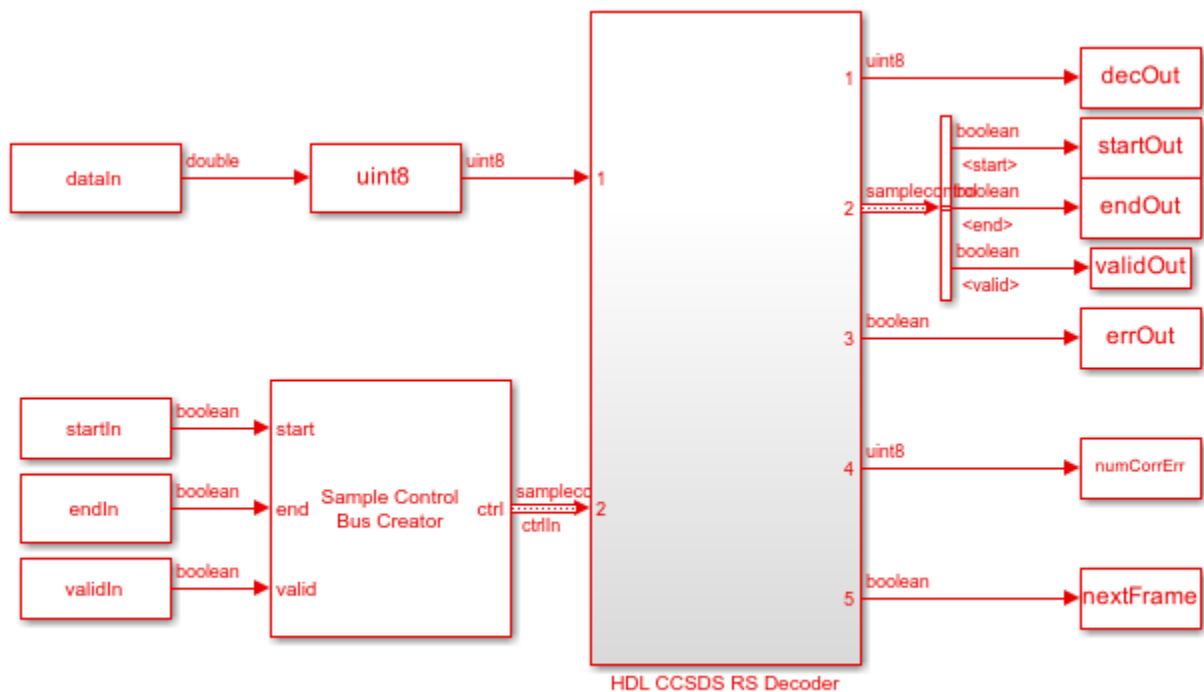

Copyright 2021 The MathWorks, Inc.

Compare Simulink Block Output with MATLAB Function Output

Compare the CCSDS RS Decoder block output with the `ccsdsRSDecode` function output.

```
dataOut = squeeze(decOut);
validOut = squeeze(validOut);
```

```
endOut = squeeze(endOut);
numCorrErrOut = squeeze(numCorrErr);
simOutput = dataOut(validOut);
fprintf('\nHDL CCSDS RS Decoder\n');
difference = double(simOutput) - double(refOutput);
fprintf(['\nTotal number of samples that differ between Simulink block output ' ...
    'and MATLAB function output is: %d \n'],sum(difference));
```

HDL CCSDS RS Decoder

Total number of samples that differ between Simulink block output and MATLAB function output is:

See Also

Blocks
CCSDS RS Decoder

Functions
ccsdsRSDecode

Decode CCSDS Reed-Solomon and Convolutional Concatenated Code

This example shows how to use the CCSDS RS Decoder block with the Viterbi Decoder block to decode a Reed-Solomon (RS) and convolutional concatenated code according to the Consultative Committee for Space Data Systems (CCSDS) standard. The synchronization and channel coding sublayer of the CCSDS TM standard includes concatenated coding scheme with Reed-Solomon code as the outer code and convolutional code as the inner code. The example supports HDL code generation for the `HDL CCSDS Concatenated Decoder` subsystem.

Set Up Concatenated Code Parameters

Specify input variables. You can change only k and i variable values in this section based on your requirements.

```
% Reed-Solomon code parameters
n = 255;  % Codeword length
k = 223;  % Message length
i = 1;    % Interleaving depth

% Convolutional code parameters
convRate = '1/2';      % Convolutional code rate
K = 7;                 % Constraint length
codePoly = [171 133];  % Code generator polynomial
trBackDepth = 32;      % Traceback depth
```

Generate Transmitter Waveform

Generate transmitter waveform using the `ccsdsTMWaveformGenerator` (Satellite Communications Toolbox) System object™ in Satellite Communications Toolbox. The System object performs RS encoding, convolutional encoding, and QPSK modulation on the input data and generates a transmitter waveform.

```
% Generate random input data
dataBits = randi([0,1],k*i*8,1);

% Configure |ccsdsTMWaveformGenerator| System object
obj = ccsdsTMWaveformGenerator('WaveformSource','synchronization and channel coding',...
    'ChannelCoding','concatenated',...
    'ConvolutionalCodeRate',convRate,...
    'RSInterleavingDepth',i,...
    'RSMessageLength',223,...
    'HasRandomizer',false,...
    'HasASM',false,...
    'PulseShapingFilter','none',...
    'Modulation','QPSK');

% Call System object to generate RS and convolutional encoded and QPSK
% modulated transmitter waveform
tmWaveform = obj(dataBits);
```

Add AWGN Channel

Add white Gaussian noise to the transmitter waveform.

```matlab
snrdB = 5; % SNR of noise in dB
snr = 10^(snrdB/10);
noiseVar = 1/snr;

% Generate noise with unit power
awgnUnitPow = (1/sqrt(2))*(randn(length(tmWaveform),1) ...
              +1i*randn(length(tmWaveform),1));

% Add noise to the transmitter waveform
chOut = tmWaveform + sqrt(noiseVar)*awgnUnitPow;
```

Demodulate Receiver Waveform

Demodulate the received AWGN channel output waveform using the `comm.PSKDemodulator` System object and prepare input for the Simulink® model.

```matlab
% Configure the |comm.PSKDemodulator| System object for QPSK demodulation
qpskDemod = comm.PSKDemodulator('ModulationOrder',4,...
    'PhaseOffset',pi/4,...
    'SymbolMapping','Custom',...
    'CustomSymbolMapping',[0 2 3 1],...  % Mapping as per the CCSDS standard
    'BitOutput',true,...
    'DecisionMethod','Approximate log-likelihood ratio',...
    'Variance',noiseVar);

% Call the System object to demodulate the received waveform and output the
% LLR values
demodOut = qpskDemod(chOut);

% Invert every alternate LLR value (starting from second LLR) to remove
% symbol inversion, according to the CCSDS standard
demodOut(2:2:end) = -demodOut(2:2:end);

% Normalize all LLR values with required soft wordlength
llrWL = 4;
maxDemodOut = max(abs(demodOut));
vitInput = fi(-demodOut*(2^(llrWL-1))/maxDemodOut,1,llrWL,0);
```

Decode Demodulated Waveform Using Simulink Model

To decode the demodulated waveform, simulate the `CCSDSConcatenateDecoder.slx` model. The model contains Viterbi Decoder and CCSDS RS Decoder blocks.

```matlab
% Input signals for the Simulink model
dataIn = vitInput;
startIn = true;
endIn = [false(length(dataIn)/2 -1,1); true];
validIn = true(length(dataIn)/2,1);

% Set mask parameters of CCSDS RS Decoder block
modelName = 'CCSDSConcatenateDecoder';
subsystem = 'HDL CCSDS Concatenated Decoder';
open_system(modelName);
set_param([modelName '/' subsystem '/CCSDS RS Decoder'], ...
        'MessageLength',num2str(k), ...
        'InterleavingDepth',num2str(i));

% Stop time
```

```
vitLatency = 148;
upsampleFac = 8;
rsLatency = 3065; % Maximum latency of the CCSDS RS Decoder block
rsOutLen = k*i;
pipelineDelay = 17;
stopTime = vitLatency + (rsLatency+rsOutLen)*upsampleFac + pipelineDelay;

% Simulate the model
sim(modelName);
```
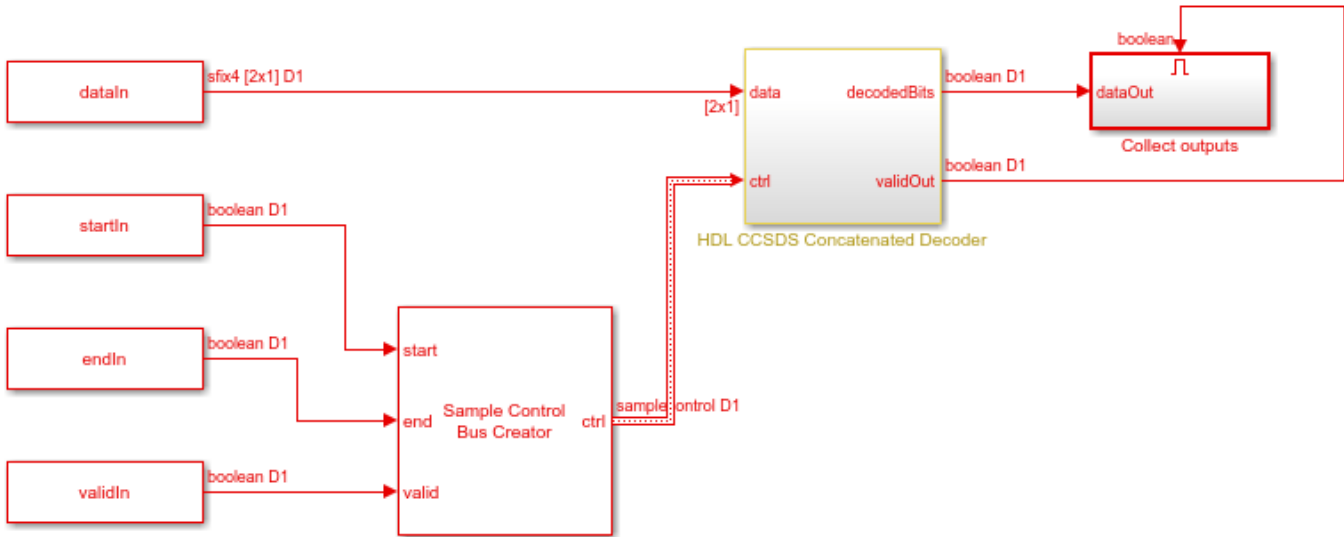

Copyright 2021 The MathWorks, Inc.

Compare Simulink Block Output with MATLAB System Object Input

Compare the output of the CCSDS RS Decoder block with the input of the `ccsdsTMWaveformGenerator` System object.

```
fprintf('\nHDL CCSDS RS Decoder\n');
fprintf('Number of bits mismatched between decoded block output and System object input: %d',nnz
```

```
HDL CCSDS RS Decoder
Number of bits mismatched between decoded block output and System object input: 0
```

See Also

Blocks
CCSDS RS Decoder | Viterbi Decoder

Functions
`ccsdsRSDecode` | `ccsdsTMWaveformGenerator`

Encode Message into RS Codeword Using CCSDS Standard

This example shows how to use the CCSDS RS Encoder block to encode a message into a Reed-Solomon (RS) codeword according to the Consultative Committee for Space Data Systems (CCSDS) standard. Generate a set of random input message symbols and provide them as input to the ccsdsRSEncode (Satellite Communications Toolbox) function and the CCSDS RS Encoder block. Compare the output of the CCSDS RS Encoder block with the output of the ccsdsRSEncode function. The Simulink® model in this example supports HDL code generation for the HDL CCSDS RS Encoder subsystem.

Set Up Input Data Parameters

Set up workspace variables for the model to use. You can modify these variable values according to your requirements. The block supports a fixed codeword length of 255.

```
k = 239;            % Message length 223 or 239
s = k;              % Shortened message length ranges from 1 to k
i = 4;              % Interleaving depth 1, 2, 3, 4, 5, or 8
numFrames = 3;      % Number of input frames
frameGap = (255-k)*i; % Minimum gap required between input frames
                    % If a new input frame is given without this frame
                    % gap, the block discards the previous frame and
                    % processes the new frame.
```

Generate Random Input Samples and Encode Using MATLAB® Function

Generate random samples using the specified message length and interleaving depth. Encode the random samples using the ccsdsRSEncode function.

```
% Generate random message symbols
msg = randi([0 255],s*i,1);

% Encode message samples
encOut = ccsdsRSEncode(msg,k,i,s);
```

Encode Input Samples Using Simulink Block

Encode the random samples using the CCSDS RS Encoder block. Running the model imports the input signal variables from the MATLAB workspace to the CCSDS RS Encoder block in the model.

```
% Assign inputs to model
data = repmat(msg,numFrames,1);
start = repmat([true; false(s*i-1,1); false((k-s)*i,1); false(frameGap,1)],numFrames,1);
endIn = repmat([false(s*i-1,1); true; false((k-s)*i,1); false(frameGap,1)],numFrames,1);
valid = repmat([true(s*i,1); false((k-s)*i,1); false(frameGap,1)],numFrames,1);

% Run Simulink model
model = 'HDLCCSDSRSEncoder';
open_system(model);
set_param([model '/HDL CCSDS RS Encoder/CCSDS RS Encoder'],'MessageLength',num2str(k),'Interleav
latency = 3; % fixed block latency
stopTime = latency + ((s*i) + frameGap)*numFrames;
sim(model);
```

Copyright 2021 The MathWorks, Inc.
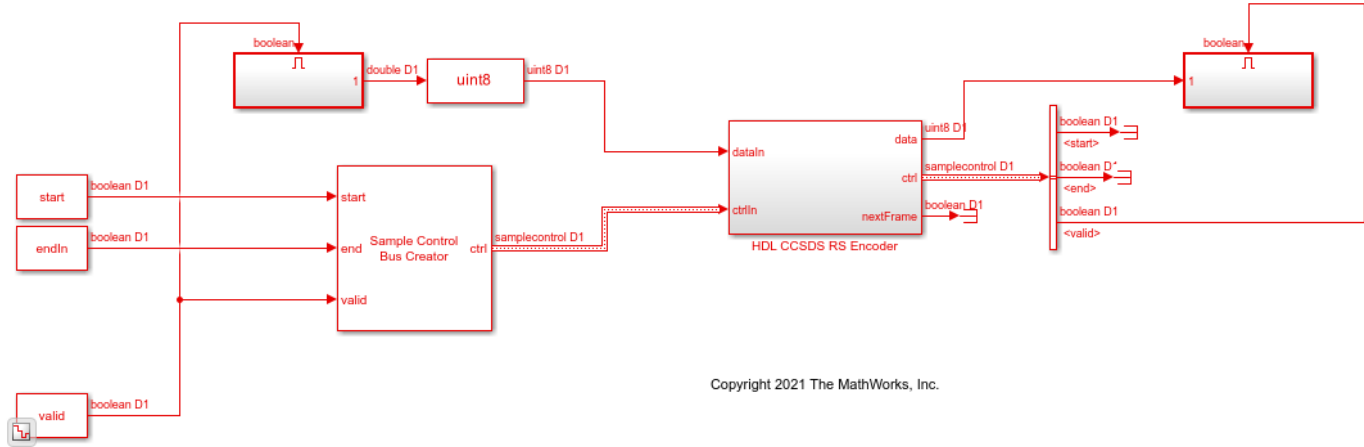
Compare Simulink Block Output with MATLAB Function Output

Compare the CCSDS RS Encoder block output with the `ccsdsRSEncode` function output.

```
simOutput = dataOut;
refOutput = repmat(encOut,numFrames,1);
fprintf('\nHDL CCSDS RS Encoder\n');
difference = double(simOutput) - double(refOutput);
fprintf(['\nTotal number of samples that differ between Simulink block output ' ...
    'and MATLAB function output is: %d \n'],sum(difference));
```

```
HDL CCSDS RS Encoder
```

```
Total number of samples that differ between Simulink block output and MATLAB function output is:
```

See Also

Blocks
CCSDS RS Encoder

Functions
ccsdsRSEncode

Encode and Decode Message with RS Code Using CCSDS Standard

This example shows how to encode and decode a message with Reed-Solomon (RS) code according to the Consultative Committee for Space Data Systems (CCSDS) standard.

The Simulink® model in this example contains CCSDS RS Encoder and CCSDS RS Decoder blocks connected back-to-back and are combined under `CCSDS RS Encode Decode` subsystem. You can generate HDL code only for this subsystem.

Set Up Input Data Parameters

Set up workspace variables for the model to use. You can modify the variable values according to your requirement. The block supports a fixed codeword length of 255.

```
numFrames = 2;  % Number of input frames
k = 239;        % Message length 223 or 239
s = k;          % Shortened message length in the range 1 to k
i = 5;          % Interleaving depth 1, 2, 3, 4, 5, or 8
```

Generate Input Samples for Simulink® Model

Generate input samples for the CCSDS RS Encoder block. Generate error samples to be introduced along with the encoder output, to provide as an input to the CCSDS RS Decoder block. Define the input frame gaps required for the blocks.

The CCSDS RS Decoder block does not support back-to-back input frames for shortened lengths (s < k) and when k = 223 and i = 1 or 2.

```
decFrGap = 0;
if((s<k) || (k==223 && i<3))
    decFrGap = 602-(k*i);    % Minimum gap required between input frames for CCSDS RS Decoder bl
end
encFrGap = (255-k)*i;        % Minimum gap required between input frames for CCSDS RS Encoder blo
frameGap = encFrGap+decFrGap;

% Generate random input samples
data = uint8(randi([0,255],s*i*numFrames,1));
valid = repmat([true(s*i,1); false((k-s)*i,1); false(frameGap,1)],numFrames,1);
start = repmat([true; false(s*i-1,1); false((k-s)*i,1); false(frameGap,1)],numFrames,1);
endIn = repmat([false(s*i-1,1); true; false((k-s)*i,1); false(frameGap,1)],numFrames,1);

% Generate errors based on the error correction capability of the CCSDS RS
% code
errSymPerFrame = (255-k)/2; % Maximum number of correctable errors per interleaving depth
noise1 = uint8(zeros((255-k+s),i*numFrames));
loc = zeros(errSymPerFrame,i*numFrames);
values = zeros(errSymPerFrame,i*numFrames);
index = 0;
for ii = 1:numFrames
    for jj = 1:i
        index = index+1;
        % Select the error locations such that there are |errSymPerFrame|
        % number of errors per interleaving depth in the decoder input
        loc(:,index) = 255*i*(ii-1)+255*(jj-1)+randperm(255-k+s,errSymPerFrame);
```

```
            % Generate random error values
            values(:,index) = randi([1 255],1,errSymPerFrame);
        end
end
noise1(loc) = values;
noise1 = reshape(noise1,[],i,numFrames);
noise = [];
for ii = 1:numFrames
    noise = [noise; reshape(noise1(:,:,ii)',[],1)]; %#ok<AGROW>
end
```

Run Simulink Model

Running the model imports the input samples to the CCSDS RS Encoder block and encodes the random input samples. It also introduces errors to the encoded output, provides them as input to the CCSDS RS Decoder block, and decodes the erroneous samples.

```
model = 'ccsdsRSEncoderDecoder';
open_system(model);
set_param([model '/CCSDS RS Encode Decode/CCSDS RS Encoder'],'MessageLength',num2str(k),'Interlea
set_param([model '/CCSDS RS Encode Decode/CCSDS RS Decoder'],'MessageLength',num2str(k),'Interlea
encLat = 3;
decLat = 3065; % Maximum latency of the CCSDS RS Decoder block
latency = encLat+decLat;
stopTime = (latency + (s*i) + frameGap)*numFrames -1;
sim(model);
```
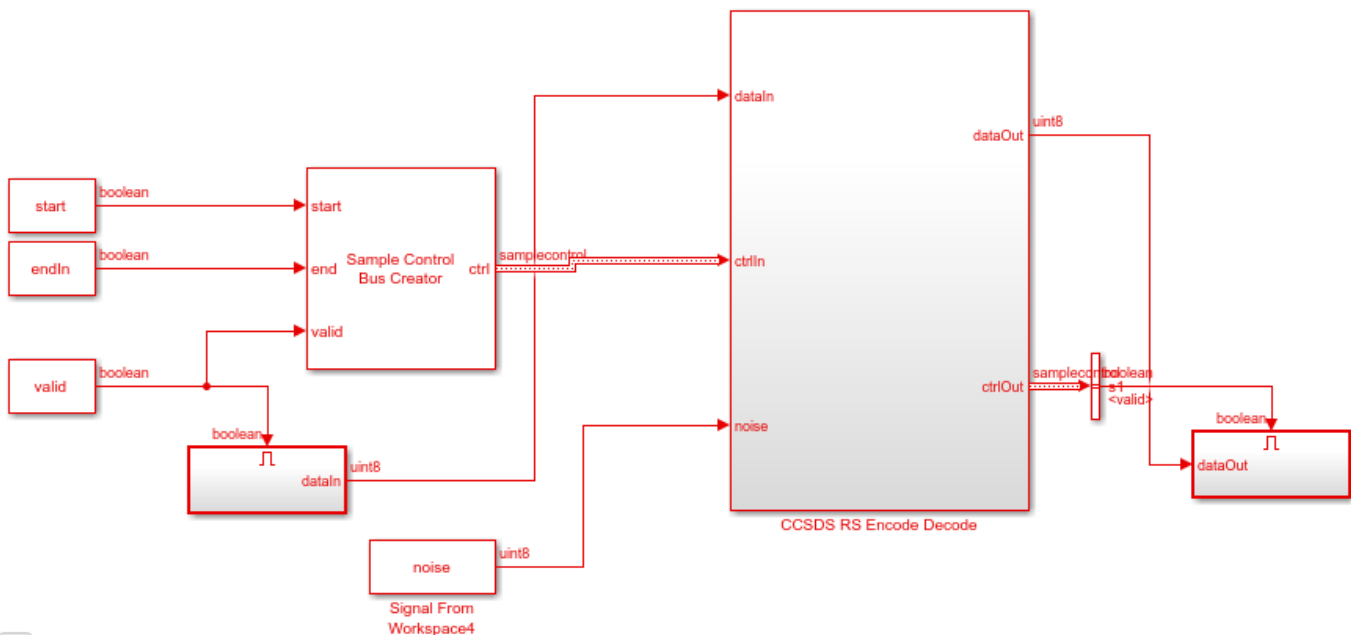

Copyright 2021 The MathWorks, Inc

Compare CCSDS RS Decoder Block Output with CCSDS RS Encoder Input

Compare the CCSDS RS Decoder block output with the CCSDS RS Encoder input.

```
decOutput = squeeze(dataOut);
encInput  = data;
```

```
fprintf('\n Compare CCSDS RS Decoder Output with CCSDS RS Encoder Input\n');
difference = double(decOutput) - double(encInput);
fprintf(['\nTotal number of samples that differ between CCSDS RS Decoder output ' ...
    'and CCSDS RS Encoder input is: %d \n'],sum(difference));
```

```
 Compare CCSDS RS Decoder Output with CCSDS RS Encoder Input

Total number of samples that differ between CCSDS RS Decoder output and CCSDS RS Encoder input is
```

See Also

Blocks
CCSDS RS Encoder | CCSDS RS Decoder

Functions
ccsdsRSEncode | ccsdsRSDecode

Decode WLAN LDPC Streaming Data

This example shows how to simulate the WLAN LDPC Decoder block and compare the hardware-optimized results with the results from the Communication Toolbox™ function.

Generate the input to the block using the Communication Toolbox function `ldpcEncode`. Provide the generated data as the input to the WLAN LDPC Decoder block and the Communication Toolbox function `ldpcDecode`. Compare the output of the block with the output of the `ldpcDecode` function. This example contains two Simulink® models. One model is configured to support the WLAN standards IEEE 802.11 n/ac/ax, and other model is configured to support the standard IEEE 802.11 ad. When you run the script, the respective model is selected based on the value that you specify for the variable `standard` mentioned in the script.

Set Up Input Variables

Choose a series of input values for the block length and code rate according to the WLAN standard. You can change the variable values in this section based on your requirements.

```
standard = 'IEEE 802.11 n/ac/ax';  % IEEE 802.11 n/ac/ax or IEEE 802.11 ad
codeRateIdx = [0; 1; 2; 3];        % Code rate index
blkLenIdx = [0; 1; 2; 0];          % Block length index
numFrames = 4;
scalar = false;                    % true for scalar inputs and false
                                   % for vector inputs
algorithm = 'Min-sum';             % Min-sum or Normalized min-sum
niter = 8;                         % Number of iterations

if strcmpi(algorithm,'Min-sum')
    alpha = 1;
else
    alpha = 0.75;                  % Scaling factor, which must be in
                                   % the range [0.5:0.0625:1]
end
```

Generate Input Data

Generate inputs for the `ldpcEncode` function with the specified block length and code rate variables. Use the encoded data from the `ldpcEncode` function, modulate the data using a modulator function, add noise using a channel System object™, and generate log-likelihood ratio (LLR) values using a symbol demodulator function. After that, provide these LLR values as an input to the `ldpcDecode` function.

Create vectors of block length index and code rate index using the `blockLenIdx` and `codeRateIdx` variables, respectively. Convert the frames of LLR values to samples with a control bus signal that indicates the frame boundaries. Provide these vectors and control bus as an input to the WLAN LDPC Decoder block.

The `decFrameGap` variable in the script accommodates the latency of the WLAN LDPC Decoder block for the specified block length, code rate, and number of iterations. Use the **nextFrame** output signal to determine when the block is ready to accept the start of the next input frame.

```
% Initialize inputs
msg = {numFrames};             % Input to |ldpcEncode| function
rxLLR = cell(1,numFrames);     % Input to |ldpcDecode| function
```

```matlab
refOut = cell(1,numFrames);          % Output of |ldpcDecode| function

decSampleIn = [];
decStartIn = [];
decEndIn = [];
decValidIn = [];
decBlkLenIdxIn = [];
decCodeRateIdxIn = [];

for ii = 1:numFrames
    if strcmpi(standard,'IEEE 802.11 n/ac/ax')
        blockLenSet = [648,1296,1944];
        rateSet = {'1/2','2/3','3/4','5/6'};

        blkLen = blockLenSet(blkLenIdx(ii)+1);
        codeRate = rateSet{codeRateIdx(ii)+1};
        modelName = 'HDLWLANLDPCDecoderStd11ac';
    else
        rateSet = {'1/2','5/8','3/4','13/16'};
        blkLen = 672;
        codeRate = rateSet{codeRateIdx(ii)+1};
        modelName = 'HDLWLANLDPCDecoderStd11ad';
    end

    [rxLLR{ii},refOut{ii},msg{ii}] = inputGenForWLANLDPCDec(blkLen,codeRate,niter,alpha);

    if scalar
        decFrameGap = niter*1000 + length(msg{ii}); %#ok
        vecSize = 1;
        len = length(rxLLR{ii});
    else
        len = length(rxLLR{ii})/8;
        vecSize = 8;
        decFrameGap = niter*1000 + ceil(length(msg{ii})/8);
    end

    decIn = reshape(rxLLR{ii},vecSize,[]);

    decSampleIn = [decSampleIn decIn zeros(size(decIn,1),decFrameGap)]; %#ok<*AGROW>
    decStartIn = logical([decStartIn 1 zeros(1,len-1) zeros(1,decFrameGap)]);
    decEndIn = logical([decEndIn zeros(1,len-1) 1 zeros(1,decFrameGap)]);
    decValidIn = logical([decValidIn ones(1,len) zeros(1,decFrameGap)]);
    decBlkLenIdxIn = ([decBlkLenIdxIn repmat(blkLenIdx(ii),1,len) zeros(1,decFrameGap)]);
    decCodeRateIdxIn = ([decCodeRateIdxIn repmat(codeRateIdx(ii),1,len) zeros(1,decFrameGap)]);
end

dataIn = timeseries(fi(decSampleIn',1,4,0));
startIn = timeseries(decStartIn);
endIn = timeseries(decEndIn);
validIn = timeseries(decValidIn);

if strcmpi(standard,'IEEE 802.11 n/ac/ax')
    blockLenIdx = timeseries(fi(decBlkLenIdxIn,0,2,0));
end
codeRateIdx = timeseries(fi(decCodeRateIdxIn,0,2,0)); % For the standard
                                                      % IEEE 802.11 ad

simTime = length(decValidIn);
```

Run Simulink Model

The `HDL WLAN LDPC Decoder` subsystem contains the WLAN LDPC Decoder block. Running the model imports the input signal variables `dataIn`, `startIn`, `endIn`, `validIn`, `blockLenIdx`, `codeRateIdx`, `niter`, and `simTime` to the block from the script and exports a stream of decoded output samples `dataOut` and a control bus containing `startOut`, `endOut`, and `validOut` signals from the block to the MATLAB workspace.

```
open_system(modelName);
if alpha ~= 1
    set_param([modelName '/HDL WLAN LDPC Decoder/WLAN LDPC Decoder'], ...
        'Algorithm','Normalized min-sum');
    set_param([modelName '/HDL WLAN LDPC Decoder/WLAN LDPC Decoder'], ...
        'ScalingFactor',num2str(alpha));
else
    set_param([modelName '/HDL WLAN LDPC Decoder/WLAN LDPC Decoder'], ...
        'Algorithm','Min-sum');
end
sim(modelName);
```
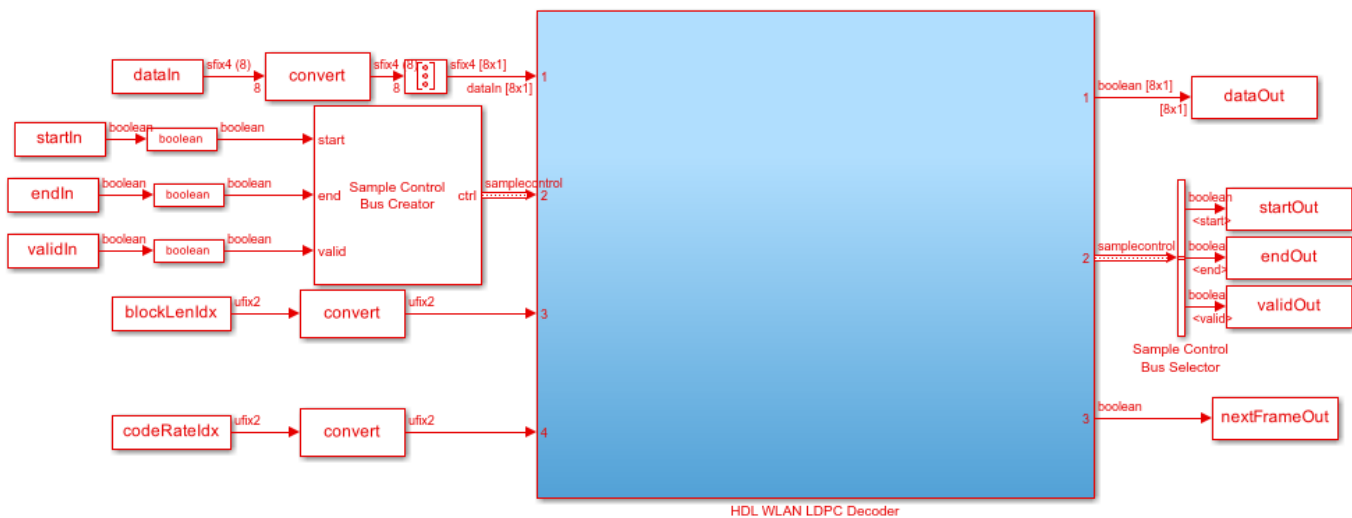

Copyright 2021 The MathWorks, Inc.

Compare Simulink Block Output with MATLAB Function Output

Convert the streaming data output of the WLAN LDPC Decoder block to frames. Compare the frames with the output of the `ldpcDecode` function.

```
startIdx = find(squeeze(startOut));
endIdx = find(squeeze(endOut));
dec = squeeze(dataOut);

decHDL = {numFrames};
for ii = 1:numFrames
    idx = startIdx(ii):endIdx(ii);
    if scalar
        decHDL{ii} = dec(idx);
    else
        decHDL{ii} = dec(:,idx);
```

```
        end
     HDLOutput = decHDL{ii}(1:length(refOut{ii}));
     error = sum(abs(double(refOut{ii})-HDLOutput(:)));
    fprintf(['Decoded frame %d: Output data differs by %d bits\n'],ii,error);
end
```

```
Decoded frame 1: Output data differs by 0 bits
Decoded frame 2: Output data differs by 0 bits
Decoded frame 3: Output data differs by 0 bits
Decoded frame 4: Output data differs by 0 bits
```

See Also

Blocks
WLAN LDPC Decoder

Functions
ldpcDecode | ldpcEncode

DVB-S2 Symbol Demodulation of Complex Data Symbols

This example shows how to use the DVB-S2 Symbol Demodulator block to demodulate complex data symbols to log-likelihood ratio (LLR) values or data bits. Generate a set of complex random inputs and provide them as an input to the block and the MATLAB® function refDVBS2SymDemod. Compare the output of the block with the output of the refDVBS2SymDemod function. This reference function uses the comm.PSKDemodulator object and the dvbsapskdemod function from Communications Toolbox™. To work with scalar and vector output types separately, this example uses two Simulink models. You can generate HDL code from the subsystems in these Simulink models.

Set Up Input Variables

Set up the input variables. You can change the variable values in this section based on your requirements. The example runs the HDLDVBS2SymbolDemodulatorScalar.slx model when you set outputType to 'Scalar' and runs the HDLDVBS2SymbolDemodulatorVector.slx model when you set outputType to 'Vector'.

```
rng(0);
framesize = 8;                     % framesize must be a multiple of 8 when you
                                   % set the 'Output type' variable to 'Vector'
                                   % framesize can be any integer greater than
                                   % 0 when you set the 'Output type' variable
                                   % to 'Scalar'
modIdx = [1;3;0;2;4];              % modIdx must contain 0, 1, 2, 3, and 4,
                                   % which correspond to the modulation schemes
                                   % QPSK, 8-PSK, 16-APSK, 32-APSK, and
                                   % pi/2-BPSK, respectively.
codeRateIdx = [5;10;6;7;9;8];     % codeRateIdx values can be 5, 6, 7, 8, 9, or 10, which
                                   % correspond to the code rates 2/3, 3/4,
                                   % 4/5, 5/6, 8/9, and 9/10, respectively.
UnitAvgCheckBox = 'on';           % on to enable and off to disable unit average power option
outputType = 'Scalar';            % outputType can be 'Scalar' or 'Vector'
decisionType = 'Approximate log-likelihood ratio';  % decisionType can be 'Approximate log-likel:

% Initialize variables
numframes = length(modIdx);
dataSymbols  = cell(1,numframes);
modSelTmp = cell(1,numframes);
modOrder = cell(1,numframes);
codeRateStr = cell(1,numframes);
referenceOutput = cell(1,numframes);
codeRateIndTmp = cell(1,numframes);
```

Generate Frames of Random Samples

Generate frames of complex random samples using the MATLAB function randn.

```
for ii = 1:numframes
    dataSymbols{ii} = complex(randn(framesize,1),randn(framesize,1));
    modSelTmp{ii} = fi(modIdx(ii)*ones(framesize,1),0,3,0);
    codeRateIndTmp{ii} = fi(codeRateIdx(ii)*ones(framesize,1),0,4,0);
end

if strcmp(UnitAvgCheckBox,'on')
 UnitAvgPower = true;
```

```matlab
else
 UnitAvgPower = false;
end
```

Convert Frames to Stream of Random Samples

Convert frames of complex random samples to a stream of complex random samples to provide them as an input to the block.

```matlab
idlecyclesbetweensamples = 0;
idlecyclesbetweenframes = 0;
[dataIn, ctrl] = whdlFramesToSamples(dataSymbols,idlecyclesbetweensamples, ...
    idlecyclesbetweenframes);
[modInd, ~] = whdlFramesToSamples(modSelTmp,idlecyclesbetweensamples, ...
    idlecyclesbetweenframes);
[codeRateInd, ~] = whdlFramesToSamples(codeRateIndTmp,idlecyclesbetweensamples, ...
    idlecyclesbetweenframes);
startIn = logical(ctrl(:,1)');
endIn = logical(ctrl(:,2)');
validIn = logical(ctrl(:,3)');

sampletime = 1;
samplesizeIn = 1;
simTime = size(ctrl,1)*8;
```

Run Simulink® Model

The `HDL DVBS2 Symbol Demodulator` subsystem contains the DVB-S2 Symbol Demodulator block. Running the model imports the input signal variables and control signals into the block from the script and exports a stream of demodulated output samples and control signals from the block to the MATLAB workspace.

```matlab
if strcmp(outputType,'Vector')
    modelname = 'HDLDVBS2SymbolDemodulatorVector';
    open_system(modelname);
    set_param([modelname '/DVBS2SymbolDemod/DVBS2 Symbol Demodulator'],'UnitAveragePower',UnitAvg
    set_param([modelname '/DVBS2SymbolDemod/DVBS2 Symbol Demodulator'],'DecisionType',decisionTyp
    symDemodOut = sim(modelname);

    startIdx = find(symDemodOut.startOut.Data);
    endIdx = find(symDemodOut.endOut.Data);
    actualData = cell(1,numframes);

    for ii = 1:numframes
        idx = startIdx(ii):endIdx(ii);
        tmpDataOut = symDemodOut.dataOut.Data(:,idx);
        dataOutSqueezed = squeeze(tmpDataOut);
        tmpValidOut = symDemodOut.validOut.Data(:,idx);
        demodOut = tmpDataOut(:,tmpValidOut);
        actualData{ii} = double(demodOut(:));
    end

else
    modelname = 'HDLDVBS2SymbolDemodulatorScalar';
    open_system(modelname);
    set_param([modelname '/DVBS2SymbolDemod/DVB-S2 Symbol Demodulator'],'UnitAveragePower',UnitAv
    set_param([modelname '/DVBS2SymbolDemod/DVB-S2 Symbol Demodulator'],'DecisionType',decisionTy
    symDemodOut = sim(modelname);
```

```
        demodOut = symDemodOut.dataOut.Data(symDemodOut.validOut.Data);
    end
```
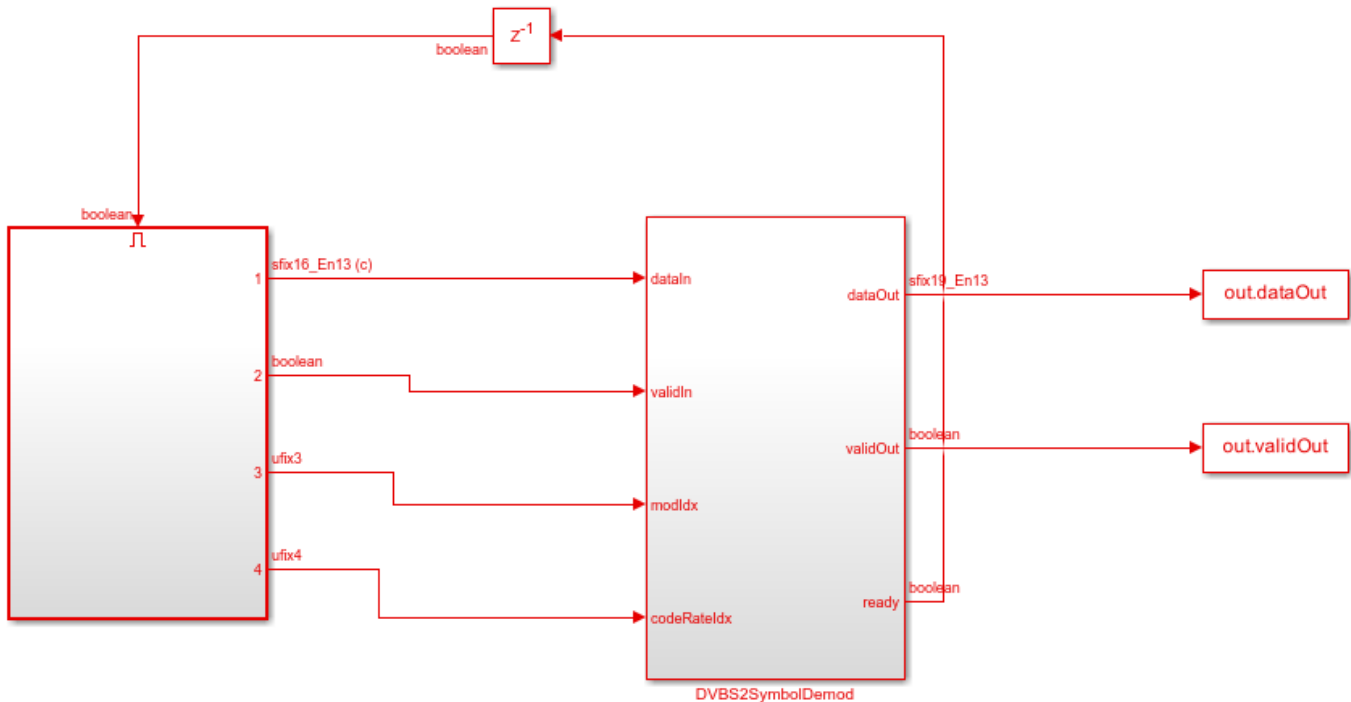

Copyright 2021 The MathWorks, Inc.

Demodulate Stream Samples Using MATLAB Function

To demodulate the stream of random samples, provide them as an input to the `refDVBS2SymDemod` function. You can use the output of this function as a reference to compare the output of the block.

```
for ii = 1:numframes
 inpParamFr.decisionType = decisionType;
 inpParamFr.UnitAvgCheckBox = UnitAvgCheckBox;
 inpParamFr.modIdx = modIdx(ii);
 inpParamFr.codeRateIdx = codeRateIdx(ii);
 referenceOutput{ii} = refDVBS2SymDemod(dataSymbols{ii},inpParamFr);
end
```

Compare Simulink Block Output with MATLAB Function Output

Compare the output of the DVB-S2 Symbol Demodulator block with the output of the `refDVBS2SymDemod` function.

```
referenceOutput = double(cell2mat(referenceOutput.'));
if strcmp(outputType,'Vector')
    actualData = double(cell2mat(actualData.'));
else
    actualData = double(squeeze(demodOut(:)));
end

figure(1)
stem(actualData,'-bo')
```

```
hold on
stem(referenceOutput,'-r*')
grid on
legend('Reference Output','Block Output')
xlabel('Sample Index')
ylabel('Magnitude')
title('Comparison of Simulink Block and MATLAB Function')
fprintf('\nPlotting the comparison results of Simulink block and MATLAB function outputs\n');
```

Plotting the comparison results of Simulink block and MATLAB function outputs
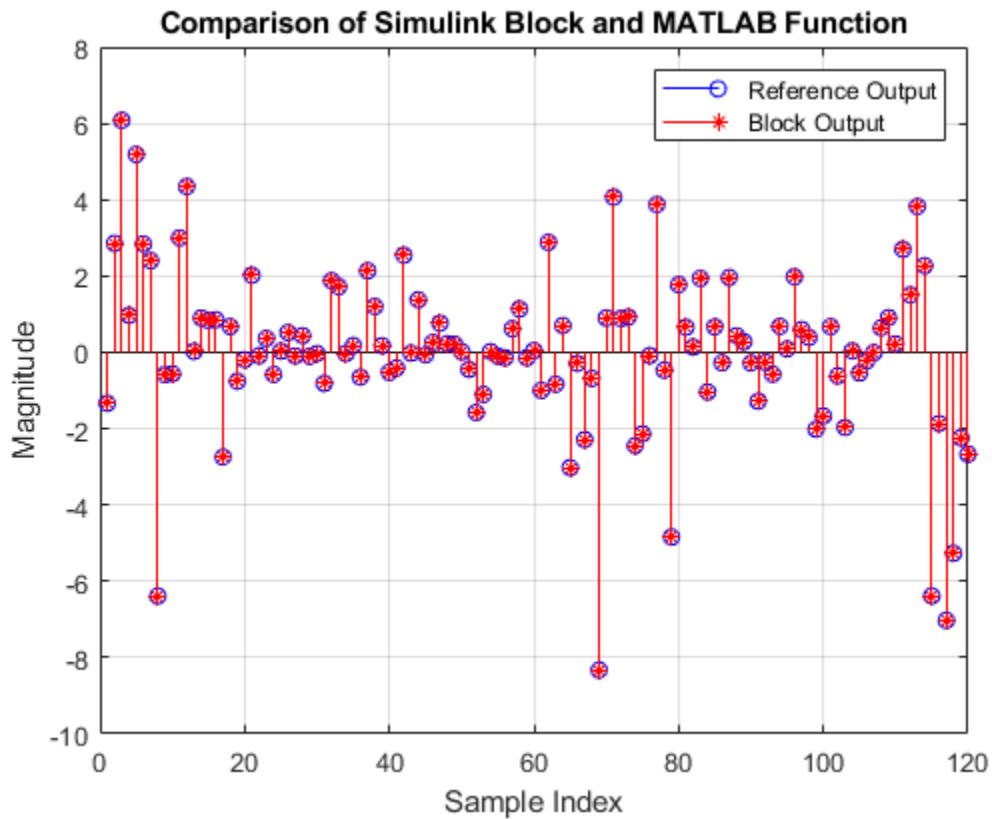
See Also

DVB-S2 Symbol Demodulator | comm.PSKDemodulator | dvbsapskdemod

Decode Convolutionally-Coded LLR Values Using APP Decoder

This example shows how to decode convolutionally-coded log-likelihood ratio (LLR) values using the APP Decoder block. To verify the results, compare the output of the block with the output of the Communication Toolbox™ System object™ `comm.APPDecoder` that is provided with same inputs as the block. This example supports HDL code generation for the `HDL APP Decoder` subsystem.

Set Up Input Variables

Specify the input variables. You can change the variable values in this section based on your requirements. In this example, you must specify the same value for the frame length (`frameLength`) and the window length (`winLen`). The block supports a maximum window length of 128.

```
numFrames = 3;
frameLength = 64;
codeGenerator = '[171 133]';                            % Code generator, specified as a row vector of
codeRate = length(str2num(codeGenerator));             % Decoding rate
winLen = 64;                                            % Window length must be less than or equal to
CodeGenDecimal = oct2dec(str2num(codeGenerator));
K = length(dec2bin(CodeGenDecimal(2)));                % Constraint length derived from code generato
TermMode = 'Truncated';                                % Terminated or Truncated
Algorithm = 'Max Log MAP (max)';                       % Max Log MAP (max) or Log MAP (max*)
```

Generate Frames of Input Data

Generate frames of LLR-coded and LLR-uncoded input data with the specified variables. To generate input data, create random binary bits, convolutionally-encode and symbol-demodulate the random binary bits, add noise to the symbol-demodulated data, and demodulate the noise-added symbol-demodulated data.

```
TrellisStructure = poly2trellis(K,str2num(codeGenerator));

if frameLength == winLen
    FrameGap = 0;
else
    FrameGap = winLen - rem(frameLength,winLen);
end

if strcmpi(TermMode,'Terminated')
    tailLen = K - 1;
else  % 'Truncated'
    tailLen = 0;
end

LLRCodedIn = [];
LLRUncodedIn = [];
startIn = [];
endIn = [];
validIn = [];

for fr=1:numFrames
    % Create binary random inputs to convolution encoder
    inpToConvEnc(:,fr) = [randn(frameLength-tailLen,1)>0; zeros(tailLen,1)];

    % Convolutionaly-encode binary random inputs
    encodedData = convenc(inpToConvEnc(:,fr), TrellisStructure);
```

```
% Modulate convolutionally-encoded data
modData = nrSymbolModulate(encodedData,'QPSK');

% Add AWGN noise to modulated data
snrdB = 8;
noiseVar = 10^-(snrdB/10);
rxSig = awgn(modData,snrdB,'measured');

% Demodulate noise-added modulated data
demod(:,fr) = nrSymbolDemodulate(rxSig,'QPSK',noiseVar);

% Prepare LLR-coded (LLRc) and LLR-uncoded (LLRu) input values to model
LLRc = reshape(demod(:,fr),codeRate,[]).';
    LLRu(:,fr) = LLRc(:,1);

LLRCodedIn = [LLRCodedIn; LLRc];
LLRUncodedIn = [LLRUncodedIn; LLRu(:,fr)];

startSig = [true; false(frameLength-1,1); false(FrameGap,1)];
endSig = [false(frameLength-1,1); true; false(FrameGap,1)];
validSig = [true(frameLength,1); false(FrameGap,1)];

startIn = [startIn; startSig];
endIn = [endIn; endSig];
validIn = [validIn; validSig];
end
stopTime = (numFrames+4)*frameLength;
```

Run Simulink Model

Run the model to import the input signal variables from the MATLAB® workspace to the APP Decoder block in the model.

```
modelName = 'HDLAPPDecoder';
open_system(modelName);
set_param([modelName '/HDL APP Decoder/APP Decoder'],'Algorithm',Algorithm);
set_param([modelName '/HDL APP Decoder/APP Decoder'],'CodeGenerator',codeGenerator);
set_param([modelName '/HDL APP Decoder/APP Decoder'],'TermMode',TermMode);
sim(modelName);
```
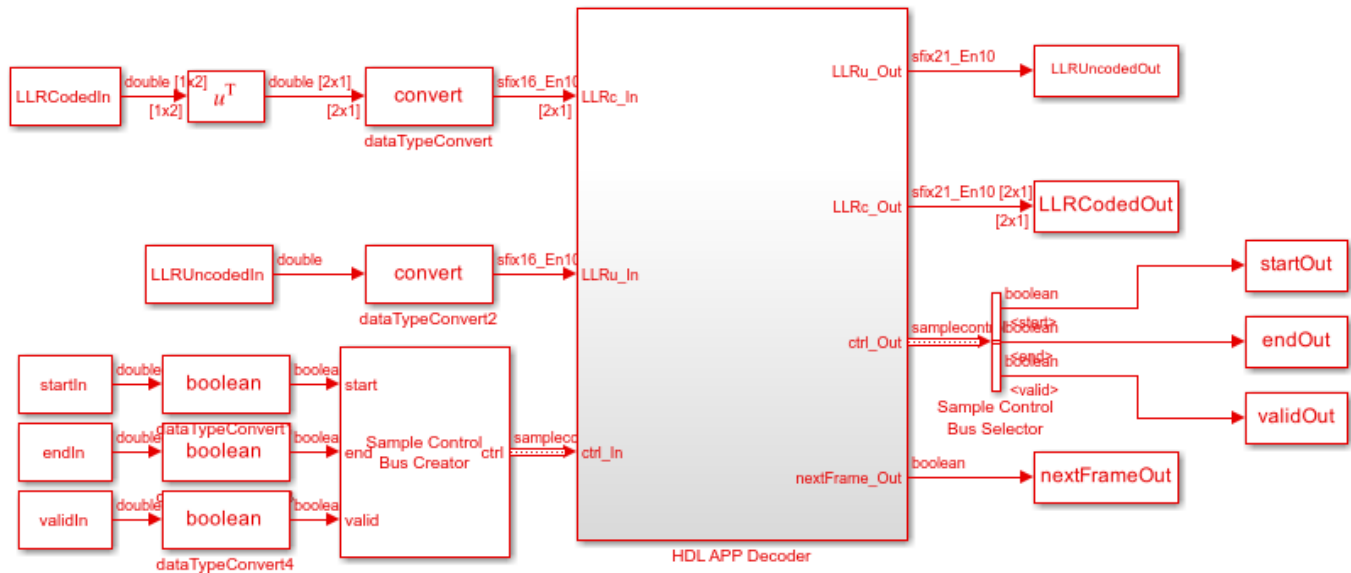
Copyright 2021 The MathWorks, Inc.

Decode Generated Data Using System Object

Create `comm.APPDecoder` System object and provide the same inputs as the block inputs.

```
hAPPDec = comm.APPDecoder;
if strcmpi(Algorithm,'Max Log MAP (max)')
    hAPPDec.Algorithm = 'Max';
else
    hAPPDec.Algorithm = 'Max*';
end
hAPPDec.TrellisStructure = TrellisStructure;
hAPPDec.TerminationMethod = TermMode;

% Generate reference output
LLRu_ref = [];
LLRc_ref = [];
for fr=1:numFrames
    [LLRu_MATLAB,LLRc_MATLAB] = hAPPDec(LLRu(:,fr),demod(:,fr));
    LLRu_ref = [LLRu_ref; LLRu_MATLAB];
    LLRc_ref = [LLRc_ref; LLRc_MATLAB];
end
```

Compare Simulink Block Output with System Object Output

Compare the APP Decoder block output with the `comm.APPDecoder` System object output.

```
LLRu_out_sim = LLRUncodedOut(validOut);
LLRc_out_sim = reshape(LLRCodedOut(:,validOut),[],1);

figure(1);
plot(LLRu_ref,'-bd');
hold on;
plot(LLRu_out_sim,'--r*')
grid on;
```

```matlab
legend('Reference output','Block output');
xlabel('Sample Index');
ylabel('Uncoded LLR Output');
title('Comparison of Block Output with System Object Output');
figure(2);
plot(LLRc_ref,'-bd');
hold on;
plot(LLRc_out_sim,'--r*')
grid on;
legend('Reference output','Block output');
xlabel('Sample Index');
ylabel('Coded LLR Output');
title('Comparison of Block Output with System Object Output');
```

Comparison of Block Output with System Object Output
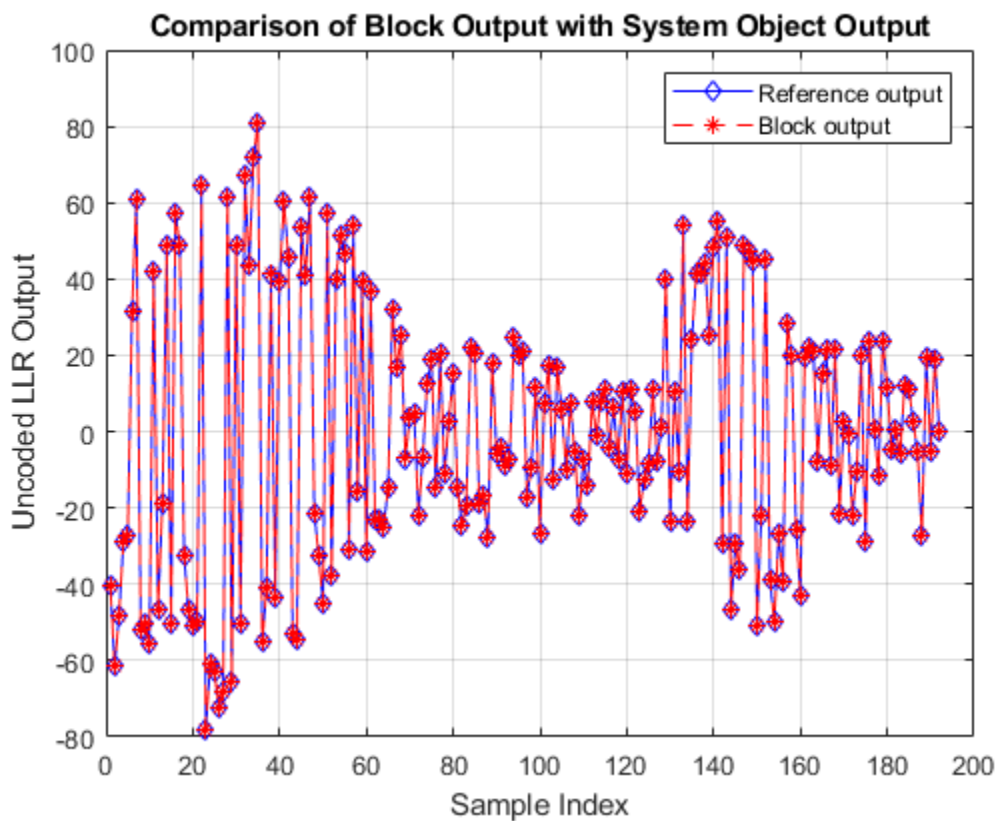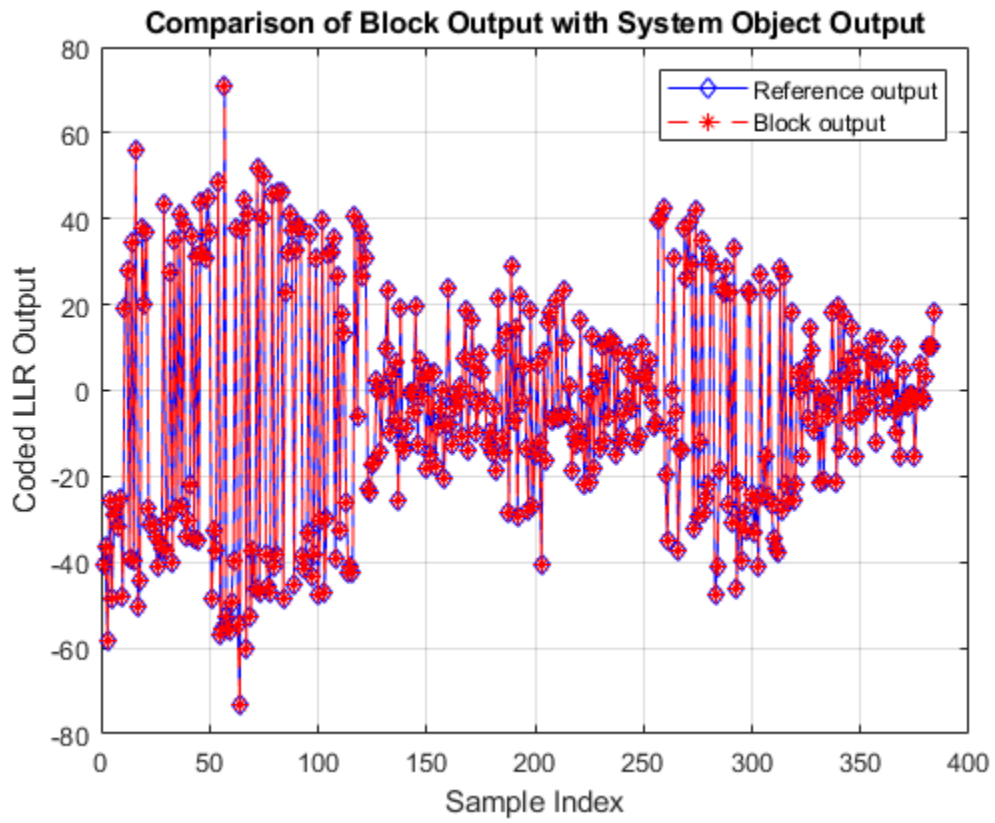
See Also

APP Decoder | `comm.APPDecoder`

Decode and Recover Message Using DVB-S2 Standard FEC Decoder

This example shows how to decode and recover a message from a codeword using a forward error correction (FEC) decoder according to the Digital Video Broadcast Satellite Second Generation (DVB-S2) standard.

The FEC decoder model in this example comprises a DVB-S2 LDPC Decoder block and a DVB-S2 BCH Decoder block connected in sequence. To provide input to the model, an encoded data of DVB-S2 standard is generated using MATLAB® functions and Satellite Communications Toolbox helper functions. After that, to verify the functionality of the blocks the output of the Simulink® model is compared with the input of the functions. The blocks used in this model support HDL code generation.

Set Up Input Variables

Set up workspace variables to generate inputs. These values are tunable and you can modify them according to your requirement.

```
numFrames = 2;          % Number of frames
frameType = [1 1];      % Type of FEC frame. 0 for normal frame and 1 for short frame.
                        % You must specify the same FEC frame type for all
                        % the frames.
codeRateIdx = [3 6];    % Code rate index must be in the range 0 to 10 for normal frame
                        % and in the range 0 to 9 for short frame
nIter = 10;             % Number of iterations in the range 1 to 63
EbNo = 15;              % To avoid bit errors, minimum EbNo must be 4 for code rate index values
                        % less than 5 and 15 for code rate index values
                        % greater than or equal to 5.
```

Download DVB-S2 LDPC Parity Matrices Data Set

To use Satellite Communications Toolbox helper functions, you need a MAT file predefined with DVB-S2 LDPC parity matrices. If the MAT file is not available on the MATLAB path, use these commands to download and unzip the MAT file.

```
if ~exist('dvbs2xLDPCParityMatrices.mat','file')
    if ~exist('s2xLDPCParityMatrices.zip','file')
        url = 'https://ssd.mathworks.com/supportfiles/spc/satcom/DVB/s2xLDPCParityMatrices.zip';
        websave('s2xLDPCParityMatrices.zip',url);
        unzip('s2xLDPCParityMatrices.zip');
    end
    addpath('s2xLDPCParityMatrices');
end

modelName = 'dvbs2LDPCBCHDecode';
open_system(modelName);
```
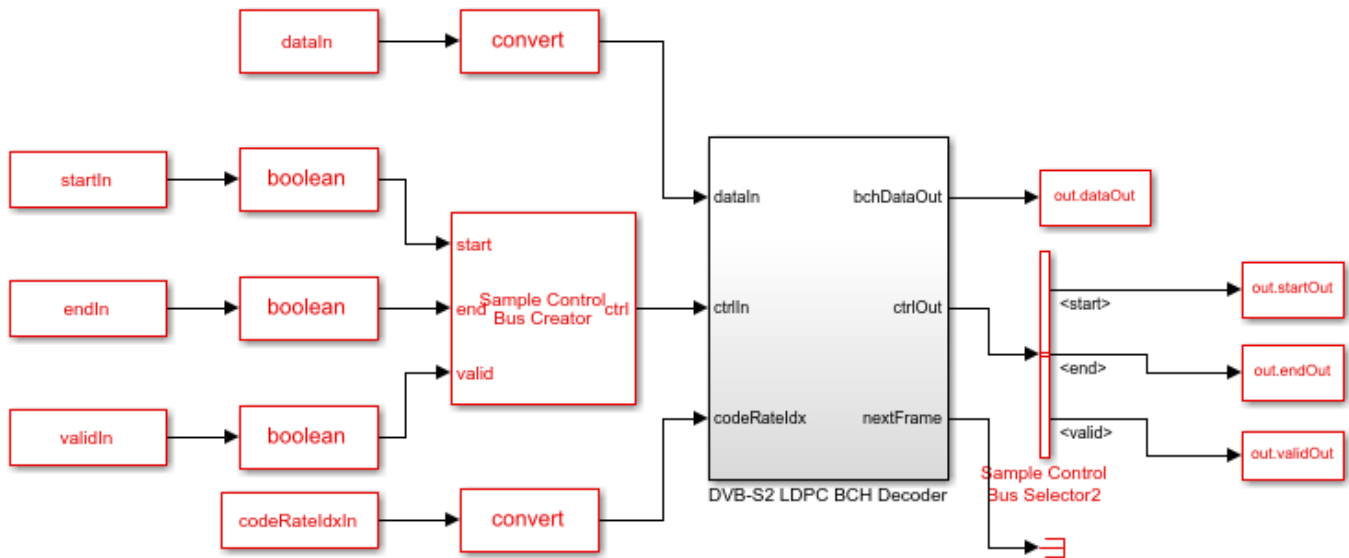
Copyright 2021 The MathWorks, Inc.

Generate Input Data

Generate input data for the Simulink® model and the MATLAB functions used in this example. Generating the input involves multiple stages as mentioned in this section.

```matlab
% Initialize inputs
fecFrameSet = {'Normal','Short'};
codeRateSet = {'1/4','1/3','2/5','1/2','3/5','2/3','3/4',...
    '4/5','5/6','8/9','9/10'};

fecFrameType = fecFrameSet(frameType+1);
codeRate = codeRateSet(codeRateIdx+1);
msg = {numFrames};

encSampleIn = [];
encValidIn = []; encStartIn = []; encEndIn =[];
nVarIn = [];
codeRateIn = [];

for ii = 1:numFrames
    fFrame =  fecFrameType{ii};
    % Input and codeword length calculation
    if strcmpi(fFrame,'Normal')
        cwLen = 64800;
        R = str2num(codeRate{ii}); %#ok<*ST2NM>
        lenList = [16008 21408 25728 32208 38688 43040 48408 51648 53840 57472 58192];
        ldpcDecLat = nIter*25000;
        set_param([modelName '/DVB-S2 LDPC BCH Decoder/DVB-S2 LDPC Decoder'],'FECFrame','Normal']
        set_param([modelName '/DVB-S2 LDPC BCH Decoder/DVB-S2 BCH Decoder'],'FECFrameType','Norma
    else
        cwLen = 16200;
        ReffList = [1/5 1/3 2/5 4/9 3/5 2/3 11/15 7/9 37/45 8/9];
        RactList = [1/4 1/3 2/5 1/2 3/5 2/3 3/4 4/5 5/6 8/9];
        Reff = ReffList(RactList == str2num(codeRate{ii}));
```

```matlab
        R = Reff(1);
        lenList = [3072 5232 6312 7032 9552 10632 11712 12432 13152 14232];
        ldpcDecLat = nIter*6500;
        set_param([modelName '/DVB-S2 LDPC BCH Decoder/DVB-S2 LDPC Decoder'],'FECFrame','Short')
        set_param([modelName '/DVB-S2 LDPC BCH Decoder/DVB-S2 BCH Decoder'],'FECFrameType','Short
    end
    inpLen = lenList(codeRateIdx(ii)+1);

    if (codeRateIdx(ii) < 5 || strcmpi(fFrame,'Short'))
        M = 4; % QPSK
    else
        M = 16; % 16-APSK
    end
    bps = log2(M);

    % Input bits generation
    msg{ii} = (randi([0 1],inpLen,1)); % Input to |bchEncode| function

    % BCH encoding
    bchOut = satcom.internal.dvbs.bchEncode(int8(msg{ii}),inpLen,cwLen);

    % LDPC encoding
    ldpcOut = satcom.internal.dvbs.ldpcEncode(int8(bchOut), codeRate{ii}, cwLen);

    % Symbol mapping
    modOut = satcom.internal.dvbs.mapper(ldpcOut, M, ...
        codeRate{ii}, cwLen, true);

    % Channel addition - AWGN channel
    EsNo = EbNo + 10*log10(bps);
    snrdB = EsNo + 10*log10(R); % in dB
    noiseVar = 1./(10.^(snrdB/10));
    chan = comm.AWGNChannel('NoiseMethod','Variance','Variance',noiseVar);
    rxData = chan(modOut);

    % Symbol demapping
     demodOut = satcom.internal.dvbs.demapper(rxData, M, ...
        codeRate{ii}, cwLen, noiseVar);

    % Latency calculation considering different frame types and code rate
    % configurations

    ldpcLen = length(demodOut);
    encFrameGap = cwLen + ldpcDecLat + 2000;

    encSampleIn = [encSampleIn demodOut.' zeros(1,encFrameGap)]; %#ok<*AGROW>
    encStartIn = logical([encStartIn 1 zeros(1,ldpcLen-1) zeros(1,encFrameGap)]);
    encEndIn = logical([encEndIn zeros(1,ldpcLen-1) 1 zeros(1,encFrameGap)]);
    encValidIn = logical([encValidIn ones(1,ldpcLen) zeros(1,encFrameGap)]);
    codeRateIn = [codeRateIn repmat(codeRateIdx(ii),1,ldpcLen) zeros(1,encFrameGap)];
end

dataIn = ((encSampleIn.'));
validIn = (encValidIn);
startIn = (encStartIn);
endIn = (encEndIn);
codeRateIdxIn = (codeRateIn);
```

```
simTime = length(encValidIn) + encFrameGap;
```

Run Simulink Model

Running the model imports the input signal variables `dataIn`, `startIn`, `endIn`, `validIn`, `frameTypeIn`, `codeRateIdxIn`, and `simTime` to the model from the script and exports a stream of decoded output samples `dataOut` and a control bus containing `startOut`, `endOut`, and `validOut` signals from the model to the MATLAB workspace.

```
out = sim(modelName);
```

Compare Simulink Model Output with MATLAB Function Input

Compare the output of the `dvbs2LDPCBCHDecode.slx` model with the input of the `bchEncode` function.

```
startIdx = find(squeeze(out.startOut));
endIdx = find(squeeze(out.endOut));
validOut = (squeeze(out.validOut));
decData = squeeze(out.dataOut);
fprintf('Decoded data with the following configuration: \n');
for ii = 1:numFrames
    idx = startIdx(ii):endIdx(ii);
    decHDL = decData(idx);
    validHDL = validOut(idx);

    HDLOutput = logical(decHDL(validHDL));
    error = sum(abs(logical(msg{ii})-HDLOutput(:)));
    fprintf('Frame: %d, FEC frame type: %s, and Code rate: %s. The Simulink model output and the
end
h = warning('off','MATLAB:rmpath:DirNotFound');
rmpath('s2xLDPCParityMatrices');
warning(h);clear h;

Decoded data with the following configuration:
Frame: 1, FEC frame type: Short, and Code rate: 1/2. The Simulink model output and the MATLAB fun
Frame: 2, FEC frame type: Short, and Code rate: 3/4. The Simulink model output and the MATLAB fun
```

See Also

Blocks
DVB-S2 BCH Decoder | DVB-S2 LDPC Decoder

Symbol Demodulation of Complex Data Symbols

This example shows how to demodulate complex data symbol using the Symbol Demodulator block. Generate a set of complex random inputs and provide them as an input to the Symbol Demodulator block and the reference functions qamdemod and pskdemod from the Communications Toolbox®. Then, compare the output of the block with the output of these functions based on type of modulation you select. To work with scalar and vector output types separately, this example provides two Simulink® models. You can generate HDL code for these models.

Set Up Input Variables

Set up input variables. You can change the variable values in this section according to your requirement. The example runs the symbolDemodulatorScalar.slx model when you set the outputType variable to 'Scalar' and the symbolDemodulatorVector.slx model when you set to 'Vector'.

```matlab
frameLength = 120;                                  % Frame length
numFrames = 8;                                      % Number of frames
frameGap = 0;                                       % Frame gap
modSel = [7 6 5 4 3 2 1 0];                         % Modulation type
maxModulation = '256-QAM';                          % {'BPSK', 'QPSK', '8-PSK', '16-PSK',
                                                    % '16-QAM', '32-PSK', '64-QAM', '256-QAM'}
phaseOffset = 'pi/2';                               % Phase offset
decisionType = 'Approximate log-likelihood ratio'; % Decision type 'Approximate log-likelihood ra
outputType = 'Vector';                              % Type of output 'Vector' or 'Scalar'
```

Generate Complex Random Inputs

Generate complex random inputs and required control signals.

```matlab
dataIn = []; validIn = []; startIn = []; endIn = []; modSelIn = [];
for frameNo = 1:numFrames
    inpData = complex(randn(1,frameLength),randn(1,frameLength));
    totalSize = frameLength;
    data = zeros(1,totalSize,'like',inpData(1));
    validCtrl = false(1,totalSize);
    modSelCtrl = zeros(1,totalSize);

    idx  = 1:totalSize;
    data(:,idx) = inpData;
    validCtrl(idx) = true;
    modSelCtrl(idx) = modSel(frameNo);

    validIdx = find(validCtrl);
    startCtrl = zeros(size(validCtrl));
    endCtrl = zeros(size(validCtrl));
    startCtrl(validIdx(1)) = 1;
    endCtrl(validIdx(end)) = 1;

    dataIn = [dataIn,zeros(frameGap,1)',data]; %#ok
    startIn = logical([startIn,zeros(frameGap,1)',startCtrl]);
    endIn = logical([endIn,zeros(frameGap,1)',endCtrl]);
    validIn = logical([validIn,zeros(frameGap,1)',validCtrl]);
    modSelIn = [modSelIn,zeros(frameGap,1)',modSelCtrl]; %#ok
end
```

```
if strcmpi(outputType,'Vector')
    stopTime = 2*frameLength*numFrames;
else
    stopTime = 8*frameLength*numFrames;
end
```

Run Simulink Model

Running the model imports the input variables and control signals to the block from the script and exports a stream of demodulated output samples and control signals from the block to the MATLAB® workspace.

```
if strcmpi(outputType,'Vector')
    modelname = 'symbolDemodulatorVector';
else
    modelname = 'symbolDemodulatorScalar';
end

load_system(modelname);
set_param([modelname '/HDL Symbol Demod/Symbol Demodulator'],'MaxModulation',maxModulation);
set_param([modelname '/HDL Symbol Demod/Symbol Demodulator'],'PhaseOffset',phaseOffset);
set_param([modelname '/HDL Symbol Demod/Symbol Demodulator'],'DecisionType',decisionType);
sim(modelname);
```

Demodulate Stream Samples Using MATLAB Function

To demodulate the stream of random samples, provide them as input to the `qamdemod` and `pskdemod` functions. You can use the output of this functions as a reference to compare the output of the block. The parameters in this section are nontunable and they are specified with default configuration values.

```
nbps = [2, 4, 8, 16, 16, 32, 64, 256];
M = nbps(modSel+1);
offset = str2num(phaseOffset); %#ok<ST2NM>
if strcmpi(decisionType,'Approximate log-likelihood ratio')
    decType = 'approxllr';
else
    decType = 'bit';
end

defConstOrder16 = [2 3 1 0 6 7 5 4 14 15 13 12 10 11 9 8];

defConstOrder64 = [4 5 7 6 2 3 1 0 12 13 15 14 10 11 9 8 28 29 31 30 26 27 25 24 20 21 23 22 18 1
             50 51 49 48 60 61    63 62 58 59 57 56 44 45 47    46 42 43 41 40    36 37 39 38 34

defConstOrder256 = [8,9,11,10,14,15,13,12,4,5,7,6,2,3,1,0,24,25,27,26,30,31,29,28,20,21,23,22,18
             19,17,16,56,57,59,58,62,63,61,60,52,53,55,54,50,51,49,48,40,41,43,42,46,47,45,44,36,3
             39,38,34,35,33,32,104,105,107,106,110,111,109,108,100,101,103,102,98,99,97,96,120,12
             122,126,127,125,124,116,117,119,118,114,115,113,112,88,89,91,90,94,95,93,92,84,85,87
             83,81,80,72,73,75,74,78,79,77,76,68,69,71,70,66,67,65,64,200,201,203,202,206,207,205
             196,197,199,198,194,195,193,192,216,217,219,218,222,223,221,220,212,213,215,214,210,2
             208,248,249,251,250,254,255,253,252,244,245,247,246,242,243,241,240,232,233,235,234,
             238,239,237,236,228,229,231,230,226,227,225,224,168,169,171,170,174,175,173,172,164,1
             166,162,163,161,160,184,185,187,186,190,191,189,188,180,181,183,182,178,179,177,176,1
             158,159,157,156,148,149,151,150,146,147,145,144,136,137,139,138,142,143,141,140,132,1


refDemodOut =[];
```

```
startIdx = find(startIn==true);
for ind =1:numFrames
    modSelFr = modSel(ind);
    dataInFr = dataIn(startIdx(ind)+(0:frameLength-1));
    if modSelFr==7 || modSelFr==6 || modSelFr==4
        if modSelFr==7
            defConstOrder = defConstOrder256;
        elseif modSelFr==6
            defConstOrder = defConstOrder64;
        else
            defConstOrder = defConstOrder16;
        end
        qamRefOut = qamdemod(dataInFr, M(ind), defConstOrder,'OutputType', decType,'UnitAveragePo
        refDemodOut = [refDemodOut;qamRefOut(:)]; %#ok
    else
        pskRefOut = pskdemod(dataInFr, M(ind), offset, 'OutputType', decType);
        refDemodOut = [refDemodOut;pskRefOut(:)]; %#ok
    end
end

simOut = (dataOut(:,validOut));
simDemodOut = simOut(:);
```

Compare Simulink Block Output with MATLAB Function Output

Compare the output of Symbol Demodulator block with the MATLAB function output.

```
figure('units','normalized','outerposition',[0 0 1 1])
plot(simDemodOut,'-k*');
hold on;
plot(refDemodOut,'--rd');
grid on;
legend('Block output','Reference function output')
xlabel('Number of Samples');
ylabel('Output Values');
title('Comparison of Simulink Block Output with MATLAB Function Output')

avgErr = mean(simDemodOut-refDemodOut);
sprintf('The average error between block output and reference function output is %d.',avgErr)


ans =

    'The average error between block output and reference function output is 0.'
```

Comparison of Simulink Block Output with MATLAB Function Output
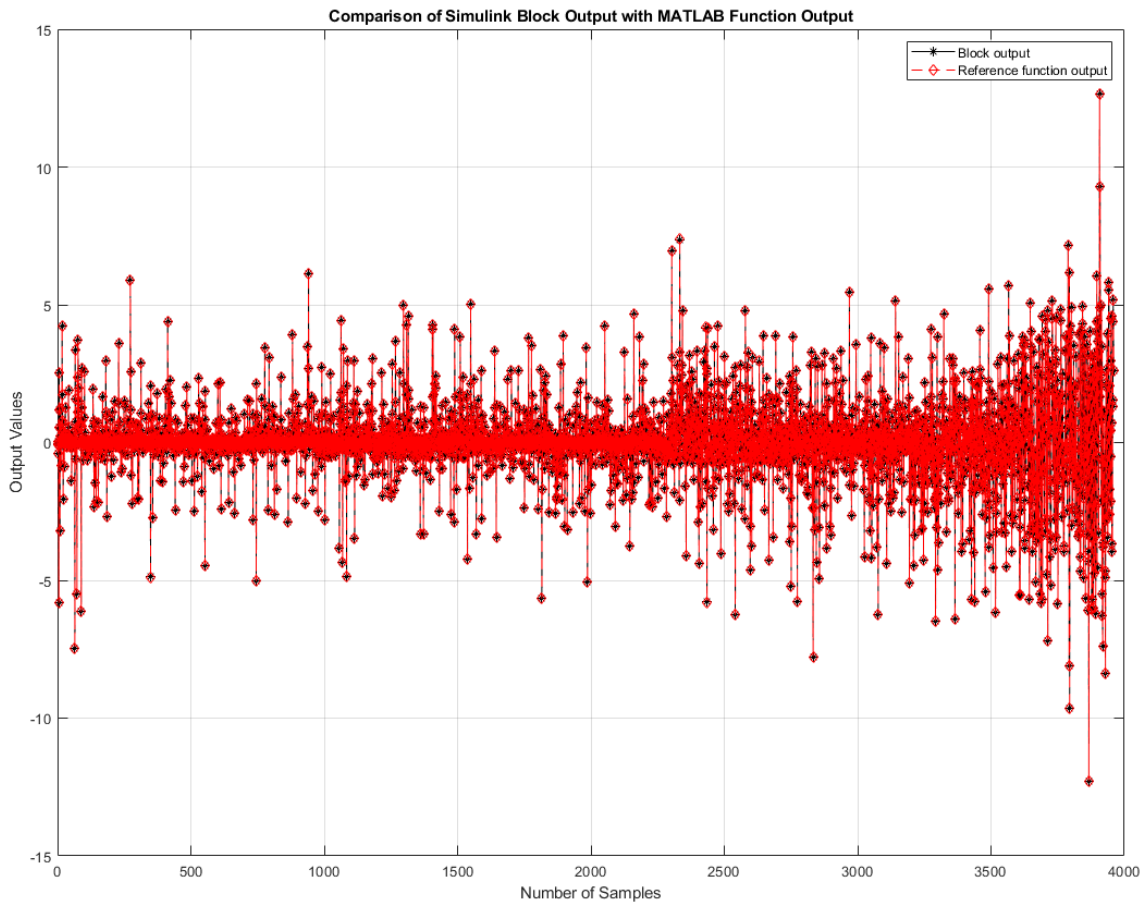
See Also

Blocks
Symbol Demodulator

Functions
qamdemod | pskdemod

Decode and Recover Message Using CCSDS LDPC Decoder

This example shows how to decode and recover a message from a codeword using an LDPC decoder designed according to the Consultative Committee for Space Data Systems (CCSDS) standard. This example comprises two Simulink® models, each containing a CCSDS LDPC Decoder block. Use the `ccsdsLDPCModelBase` model for the `(8160,7136) LDPC` configuration and the `ccsdsLDPCModelAR4JA` model for the `AR4JA LDPC` configuration. To verify the behavior of the block, compare the output of the model with the input provided to the encoder function. The CCSDS LDPC Decoder block used in these models supports HDL code generation.

Set Up Input Variables

Set up workspace variables to generate inputs. These values are tunable and you can modify them according to your requirement. The block length index must be in the range [0, 2] and it is applicable only when `configType` is set to `'AR4JA LDPC'`. The code rate index must be in the range [0, 2] and it is applicable only when `configType` is set to `'AR4JA LDPC'`.

```
numFrames = 2;
configType = 'AR4JA LDPC'; % Select the configuration type as
                           % '(8160,7136) LDPC' or 'AR4JA LDPC'
blkLenIdx = [0 2];
codeRateIdx = [1 0];
nIter = 8;                 % Number of iterations in the range [1, 63]
EbNo = 4;
```

Generate Input Data

Generate input data for the Simulink® model and the MATLAB functions that you use in this example.

```
blkLenSet = [1024 4096 16384];
codeRateSet = {'1/2' '2/3' '4/5'};

msg = {numFrames};

decSampleIn = [];
decValidIn = []; decStartIn = []; decEndIn =[];
blockLenIn = []; codeRateIn = [];

for ii = 1:numFrames

    % Input and output codeword length calculation
    if strcmpi(configType,'AR4JA LDPC')
        blockLen = blkLenSet(blkLenIdx(ii)+1);
        codeRate = codeRateSet{codeRateIdx(ii)+1};
        R = str2num(codeRate); %#ok<*ST2NM>
    else
        blockLen = 7136;
        R = 7/8;
    end

    bps = 2;

    % Input bits generation
    msg{ii} = (randi([0 1],blockLen,1)); % Data before encoding
```

```matlab
    % LDPC encoding
    G = generatorMatrix(configType,blockLen,R);
    encData = satcom.internal.ccsds.tmldpcEncode(msg{ii},G); % Encoded data

    % Symbol mapping
    symOut = pskmod(double(encData),bps);

    % Channel addition - AWGN channel
    EsNo = EbNo + 10*log10(bps);
    snrdB = EsNo + 10*log10(R); % Noise in dB
    noiseVar = 1./(10.^(snrdB/10));
    chan = comm.AWGNChannel('NoiseMethod','Variance','Variance',noiseVar);
    rxData = chan(symOut);

    % Symbol demapping
    demodOut = pskdemod(rxData,bps,'OutputType','llr', ...
                'NoiseVariance',noiseVar);

    ldpcLen = length(demodOut);
    decFrameGap = nIter*5000; % Maximum frame gap considering all block lengths and code rates

    decSampleIn = fi([decSampleIn demodOut.' zeros(1,decFrameGap)],1,4,0); %#ok<*AGROW>
    decStartIn = logical([decStartIn 1 zeros(1,ldpcLen-1) zeros(1,decFrameGap)]);
    decEndIn = logical([decEndIn zeros(1,ldpcLen-1) 1 zeros(1,decFrameGap)]);
    decValidIn = logical([decValidIn ones(1,ldpcLen) zeros(1,decFrameGap)]);
    blockLenIn = fi([blockLenIn repmat(blkLenIdx(ii),1,ldpcLen) zeros(1,decFrameGap)],0,2,0);
    codeRateIn = fi([codeRateIn repmat(codeRateIdx(ii),1,ldpcLen) zeros(1,decFrameGap)],0,2,0);
end

dataIn = decSampleIn.';
validIn = decValidIn;
startIn = decStartIn;
endIn = decEndIn;
blkLenIdxIn = blockLenIn;
codeRateIdxIn = codeRateIn;

simTime = length(decValidIn) + decFrameGap;

if strcmpi(configType,'AR4JA LDPC')
    modelName = 'ccsdsLDPCModelAR4JA.slx';
else
    modelName = 'ccsdsLDPCModelBase.slx';
end
```
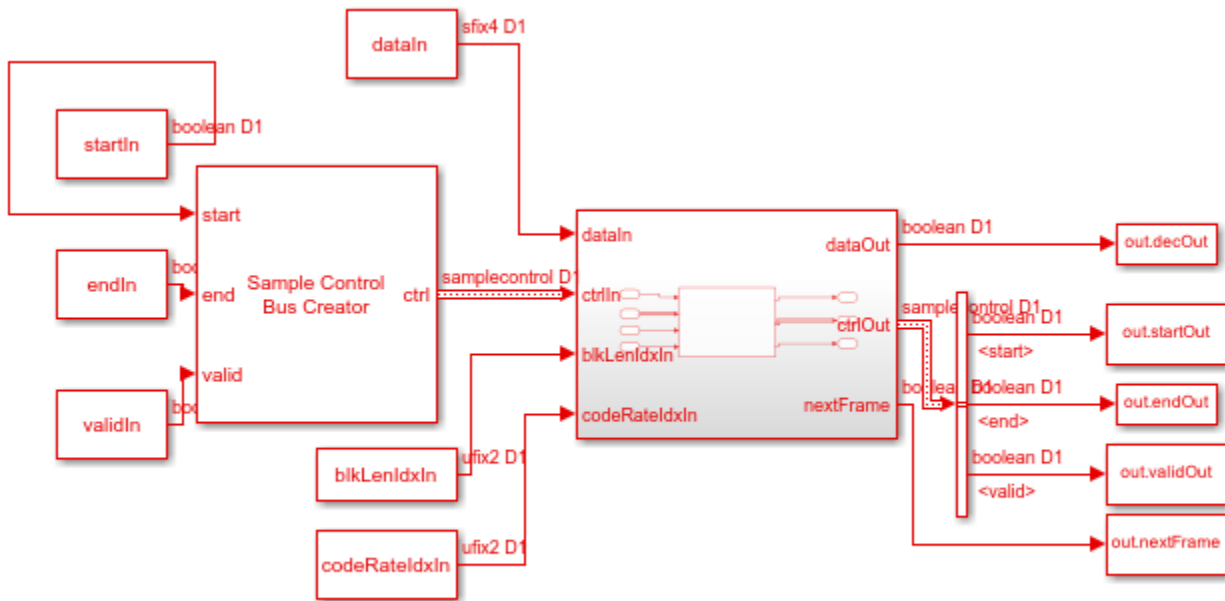
Run Simulink Model

Running the model imports the input signal variables `dataIn`, `startIn`, `endIn`, `validIn`, `blkLenIdxIn`, `codeRateIdxIn`, and `simTime` from the script. The model exports a stream of decoded output samples `dataOut` and a control bus containing `startOut`, `endOut`, and `validOut` signals to the MATLAB workspace.

```matlab
open_system(modelName);
out = sim(modelName);
```

Copyright 2022 The MathWorks, Inc.

Compare Simulink Model Output with MATLAB Function Input

Compare the output of the model with the encoder function input data.

```
startIdx = find(squeeze(out.startOut));
endIdx = find(squeeze(out.endOut));
validOut = (squeeze(out.validOut));
decData = squeeze(out.decOut);
fprintf('Decoded data with the following configuration: \n');
for ii = 1:numFrames
    idx = startIdx(ii):endIdx(ii);
    decHDL = decData(idx);
    validHDL = validOut(idx);

    HDLOutput = logical(decHDL(validHDL));
    error = sum(abs(logical(msg{ii})-HDLOutput(:)));
    fprintf('Frame: %d, The Simulink model output and the MATLAB function input differs by %d bi
end
close_system(modelName,0);

Decoded data with the following configuration:
Frame: 1, The Simulink model output and the MATLAB function input differs by 0 bits
Frame: 2, The Simulink model output and the MATLAB function input differs by 0 bits
```

See Also

Blocks
CCSDS LDPC Decoder

Functions
pskdemod | pskmod

3-133

DVB-S2 Symbol Modulation of Data Bits

This example shows how to modulate data bits to complex data symbols using the DVB-S2 Symbol Modulator block. Generate a set of random inputs and provide them as an input to the block and the MATLAB® function `refDVBS2SymMod`. Compare the output of the block with the output of the `refDVBS2SymMod` function. This reference function uses a `comm.PSKModulator` object and the `dvbsapskmod` function from Communications Toolbox™.

Set Up Input Variables

Set up input variables. You can change the variable values in this section based on your requirements. The `modIdx` values 0, 1, 2, 3, and 4 correspond to the modulation schemes QPSK, 8-PSK, 16-APSK, 32-APSK and pi/2-BPSK, respectively. The `codeRateIdx` values 5, 6, 7, 8, 9, and 10 correspond to the code rates 2/3, 3/4, 4/5, 5/6, 8/9, and 9/10, respectively.

```
rng(0);
numBits = 60;
modIdx = [1; 3; 0; 2; 4];
codeRateIdx = [5; 10; 6; 7; 9; 8];
unitAvgPower = 'on';              % 'on' to enable and 'off' to disable
outputDataType = 'Custom';        % Output data type can be 'double', 'single', or 'Custom'
wordLength = 16;                  % Output word length can be in the range [3, 32]

numframes = length(modIdx);
dataSymbols  = cell(1,numframes);
modSelTmp = cell(1,numframes);
codeRateIndTmp = cell(1,numframes);
loadTmp = cell(1,numframes);
referenceOutput = cell(1,numframes);
modOrder = cell(1,numframes);
codeRateStr = cell(1,numframes);
```

Generate Frames of Random Samples

Generate bits using the `randi` function.

```
for ii = 1:numframes
    dataSymbols{ii} = randi([0 1],numBits,1);
    modSelTmp{ii} = fi(modIdx(ii)*ones(numBits,1),0,3,0);
    codeRateIndTmp{ii} = fi(codeRateIdx(ii)*ones(numBits,1),0,4,0);
    loadTmp{ii} = boolean([1;zeros(numBits-1,1)]);
end
```

Convert Frames to Stream of Random Samples

Convert frames of complex random samples to a stream of complex random samples that the block can accept as input.

```
idlecyclesbetweensamples = 0;
idlecyclesbetweenframes = 0;
[dataIn,ctrl] = whdlFramesToSamples(dataSymbols,idlecyclesbetweensamples, ...
    idlecyclesbetweenframes);
[modInd,~] = whdlFramesToSamples(modSelTmp,idlecyclesbetweensamples, ...
    idlecyclesbetweenframes);
[codeRateInd,~] = whdlFramesToSamples(codeRateIndTmp,idlecyclesbetweensamples, ...
    idlecyclesbetweenframes);
```

```
[loadIn,~] = whdlFramesToSamples(loadTmp,idlecyclesbetweensamples, ...
    idlecyclesbetweenframes);
validIn = logical(ctrl(:,3)');

sampletime = 1;
samplesizeIn = 1;
simTime = size(ctrl,1)*2;
```

Run Simulink Model

The `HDL DVB-S2 Symbol Modulator` subsystem contains the DVB-S2 Symbol Modulator block. Running the model imports the input signal variables `dataIn`, `validIn`, `modIdx`, `codeRateIdx`, and `loadIn` to the block from the script. The model exports a stream of modulated output samples `dataOut` and `validOut` signals from the block to the MATLAB workspace.

```
modelname = 'HDLDVBS2SymbolModulator';
open_system(modelname);
set_param([modelname '/DVBS2SymbolMod/DVB-S2 Symbol Modulator'],'UnitAveragePower',unitAvgPower)
set_param([modelname '/DVBS2SymbolMod/DVB-S2 Symbol Modulator'],'OutputDataType',outputDataType)
symModOut = sim(modelname);
modOut =  symModOut.dataOut.Data(symModOut.validOut.Data);
```


Copyright 2022 The MathWorks, Inc.
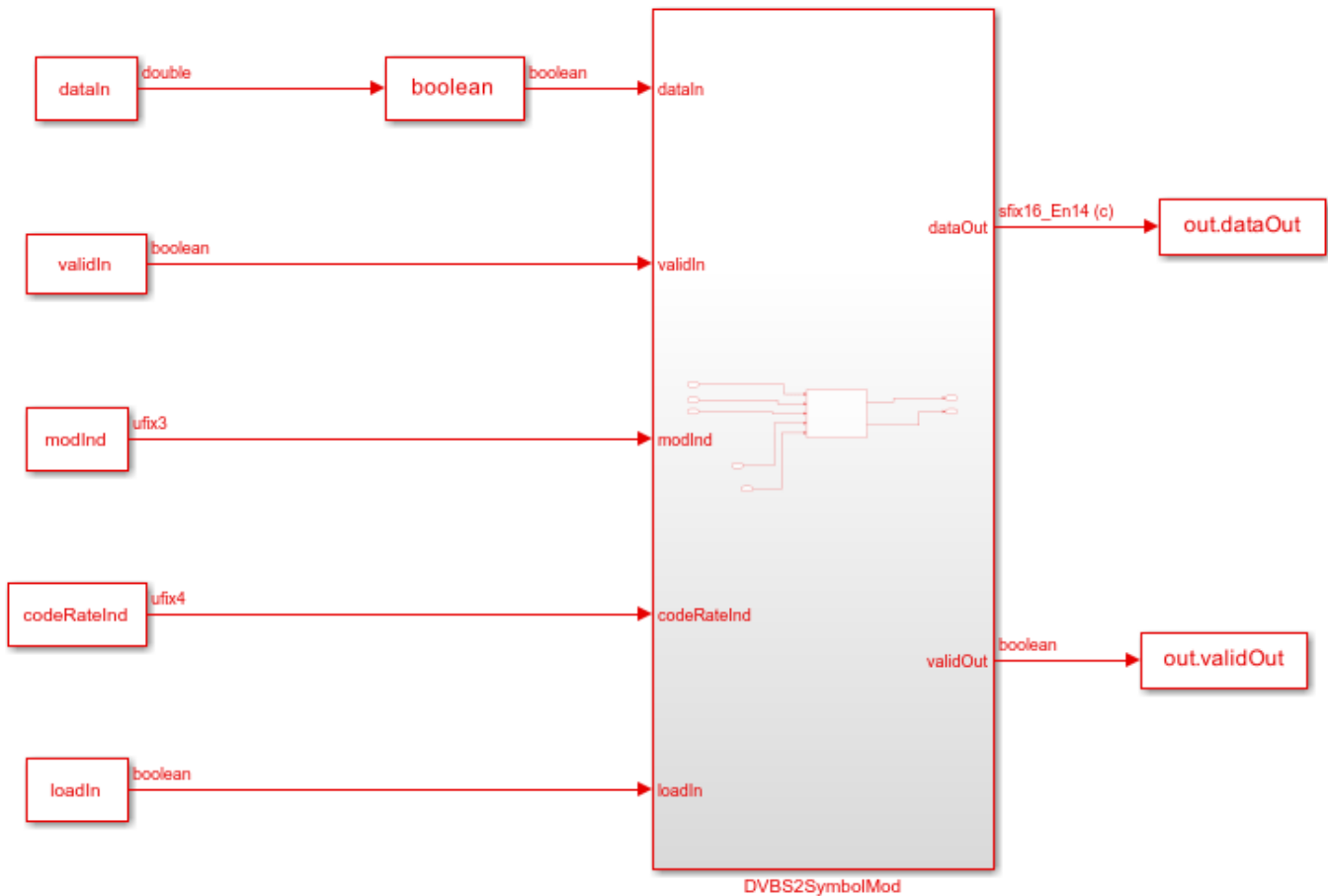
Modulate Stream Samples Using MATLAB Function

To symbol modulate the stream of random bits, provide them as input to the `refDVBS2SymMod` function. You can use the output of this function as a reference against which to compare the output of the block.

```matlab
for ii = 1:numframes
 inpParamFr.UnitAvgPowerRef = unitAvgPower;
 inpParamFr.modIdx = modIdx(ii);
 inpParamFr.codeRateIdx = codeRateIdx(ii);
 referenceOutput{ii} = refDVBS2SymMod(dataSymbols{ii},inpParamFr);
end
```

Compare Simulink Block Output with MATLAB Function Output

Compare the output of the DVB-S2 Symbol Modulator block with the output of the `refDVBS2SymMod` function.

```matlab
referenceOutput = double(cell2mat(referenceOutput.'));
actualData = double(squeeze(modOut(:)));

figure('units','normalized','outerposition',[0 0 1 1])
subplot(2,1,1)
plot(real(referenceOutput(:)));
hold on;
plot(real(actualData(:)));
grid on
legend('MATLAB Reference Output','Simulink Block Output')
xlabel('Sample Index')
ylabel('Magnitude')
title('Comparison of Simulink Block and MATLAB Function (Real Part)')

subplot(2,1,2)
plot(imag(referenceOutput(:)));
hold on;
plot(imag(actualData(:)));
grid on
legend('MATLAB Reference Output','Simulink Block Output')
xlabel('Sample Index')
ylabel('Magnitude')
title('Comparison of Simulink Block and MATLAB Function (Imaginary Part)')

sqnrRealdB = 10*log10(double(var(real(actualData(:)))/abs(var(real(actualData(:)))-var(real(refer
sqnrImagdB = 10*log10(double(var(imag(actualData(:)))/abs(var(imag(actualData(:)))-var(imag(refer

fprintf('\n DVB-S2 Symbol Modulator \n SQNR of real part: %.2f dB',sqnrRealdB);
fprintf('\n SQNR of imaginary part: %.2f dB\n',sqnrImagdB);
```
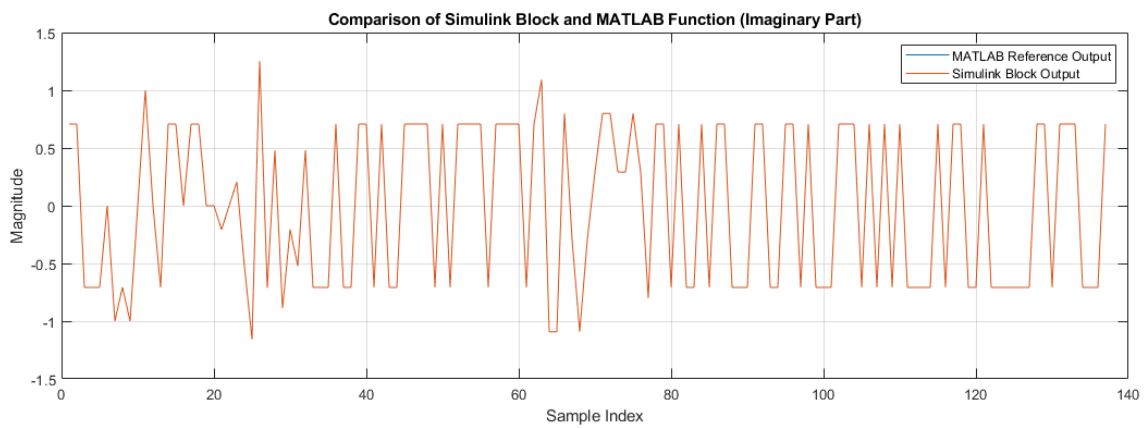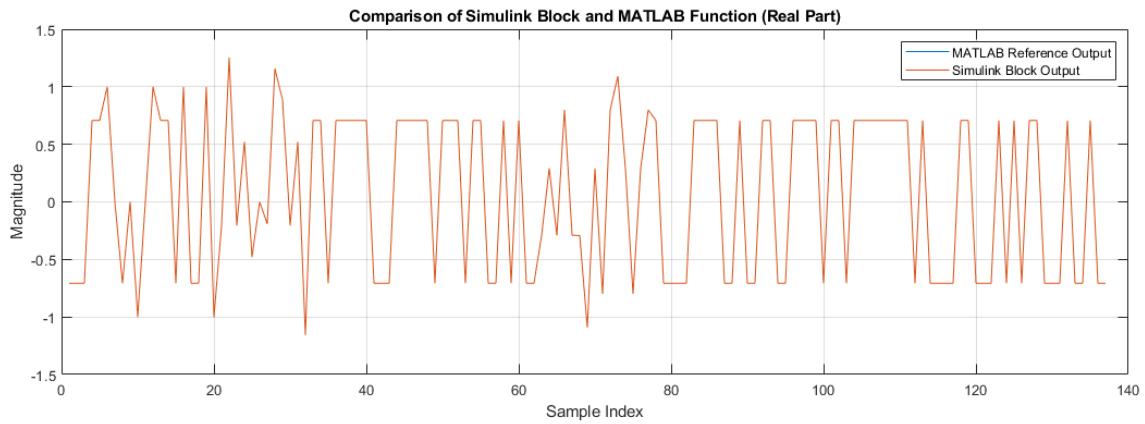
```
 DVB-S2 Symbol Modulator
 SQNR of real part: 44.25 dB
 SQNR of imaginary part: 44.20 dB
```

Comparison of Simulink Block and MATLAB Function (Real Part)

Comparison of Simulink Block and MATLAB Function (Imaginary Part)

See Also

Blocks
DVB-S2 Symbol Modulator | DVB-S2 Symbol Demodulator

Functions
`dvbsapskmod`

Objects
`comm.PSKModulator`

# Featured Examples

Sample Rate Conversion for an LTE Receiver

This example shows how to design and implement sample rate conversion for an LTE receiver front end. The model is compatible with the Wireless HDL Toolbox™ receiver reference applications, and supports HDL code generation with HDL Coder™.

Introduction

The "LTE HDL Cell Search" on page 5-106, "LTE HDL MIB Recovery" on page 5-141, and "LTE HDL SIB1 Recovery" on page 5-123 reference applications require an input sampling rate of 30.72 Msps. In practice, the sampling rate presented to hardware may differ from this, for example due to choice of components or system design decisions. Therefore, sample rate conversion may be required to integrate these reference applications into a larger system. The model shown in this example converts from 125, 140, or 150 Msps to 30.72 Msps using two FIR Decimation filters and a programmable Farrow rate converter. The rate changes from 125, 140 or 150 Msps to 30.72 Msps were deliberately chosen because they are not trivial to implement yet represent an example of the type of rate change often required in a radio receiver.

Sample Rate Converter Design Overview

At the top level, you can program the rate converter using the FarrowSelect input. This allows you to select between three predefined input sampling rates - 125, 140 or 150 Msps. You can program the Farrow Rate Converter to have any rate change in practice. The default rates are just chosen as an example. The default conversion from 150 Msps to 30.72 Msps corresponds to a rate change factor of 0.2048. This is implemented with the filter chain shown. First, a Farrow rate converter is used to make a fine adjustment to the sample rate by a factor of $150/30.72*4 = 1.2207$. Next, the signal is decimated by two (i.e. a rate change of 1/2) using a halfband filter. Last, a decimating FIR filter implements the final decimate-by-two stage.
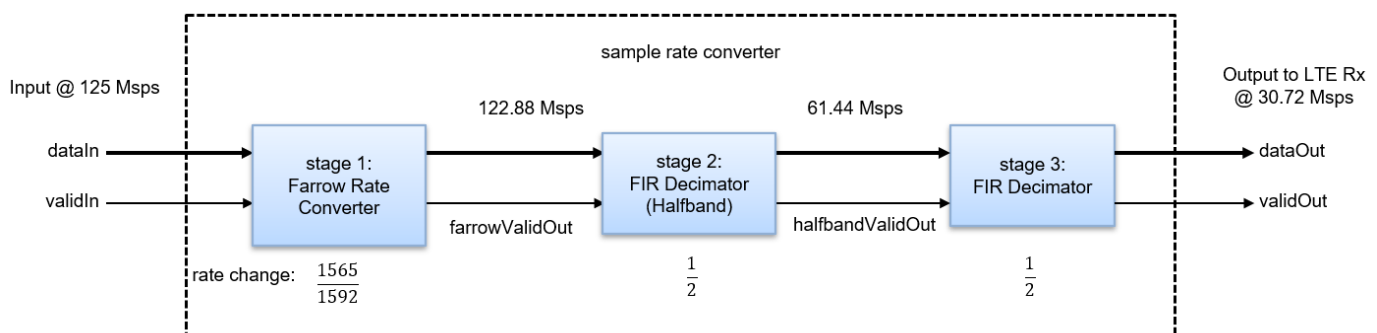
Choice of Filters

The reasons for using this set of filters are as follows:

1 A Farrow rate converter was chosen to implement the fine adjustment stage due to the high rate change resolution achievable with this approach. This leads to a flexible design which can be readily modified to implement other rate changes.

2 While Farrow filters achieve high rate change resolution, aliasing can be an issue. A good design practice is to place the Farrow Rate Converter as far away from Nyquist bandwidth as possible, and to keep the rate change close to 1. Both of these design practices are met in the design.

3 The intermediate filter stage can be done efficiently with a halfband filter. The subsequent filter then has two cycles available per input sample to implement resource sharing.

4 It then follows that the last stage is a decimating FIR filter, which can use resource sharing by a factor of two.

Clock Rate and Valid Signals

In this example, the default clock rate is 150 MHz and the default input sampling rate is 150 Msps. Sampling rates are conveyed by the duty cycle of the valid signals (the percentage of time that valid is `true`) at each stage. For example, the duty cycle of `validIn` is 100% and the duty cycle of `farrowValidOut` is is 81.92% and has an irregular, non-periodic pattern. It follows the `true`, `false`, `true`, `false` ... pattern most of the time, however it will occasionally miss a `true` cycle to represent the rate correctly. The FIR Decimator halves the sampling rate again, however it also has an irregular, non-periodic valid output pattern because it is driven by the Farrow Rate Converter. `validOut` has a duty cycle of 20.48%.

While the simulation shown in this example uses a `validIn` duty cycle of 100%, the the sample rate converter can accept any valid input pattern with any duty cycle. This is useful in scenarios where the hardware clock rate is greater than the input sampling rate.

Top Level Parameters

Configure the top level parameters of the sample rate converter. `FsADC` is the input rate, while `FsLTERx` is the output rate; that is, the input to the LTE receiver. You can modify `FsADC` to be 100e6, 125e6, or 150e6, so long as you drive the FarrowSelect switch with the correct value. `Fpass` is the passband cut-off frequency and is set to 10 MHz to accommodate the maximum possible LTE bandwidth of 20 MHz. `Fstop` is set to the Nyquist rate, however can be adjusted if more out-of-band signal rejection is required. `Ast` is the stopband attenuation in dBs, and `Ap` is the desired amount of passband ripple.

```
FsADC   = 150e6;
FsLTERx = 30.72e6;
Fpass   = 10e6;
Fstop   = FsLTERx/2;
Ast     = 60;
Ap      = 0.1;
```

Farrow Rate Converter

In this example, the Farrow Rate Converter uses the default 3rd order LaGrange coefficients. These are derived from a closed form solution and in general work for any rate change. The Farrow filter structure is the same as that used in the `dsp.VariableIntegerDelay` and `dsp.FarrowRateConverter` System objects.

Define the key parameters of the Farrow rate converter. `FsIn` and `FsOut` are the input and output rates respectively.

```
farrow.FsIn    = FsADC;
farrow.FsOut   = 4*FsLTERx;
```

Now, we can evaluate the Farrow Rate Converter.

```
% Generate an impulse input with a length of Lx samples.
Lx = 10;
x = zeros(Lx,1);
```

```matlab
x(1) = 1;

% Evaluate the oversampled impulse response of the
% Farrow-based Variable Fractional Delay.
% Instantiate a variable fractional delay object.
% Pass the impulse through it at N different fractional
% delays from 0 to 1-(1/N) in steps of 1/N and store
% the results in the oversampled response vector p.

vfd = dsp.VariableFractionalDelay( ...
    'InterpolationMethod','Farrow');

N = 4;
Lp = N * Lx;
p = zeros(Lp,1);


for n=1:N
    p(n:N:end) = vfd(x,4+(N-n)/N);
end

% Plot the impulse response
figure(1); clf;
t = (0:length(p)-1)/N;
plot(t,p,'-o');
title("Impulse response");
xlabel("Time index, k" + newline + "(in samples at original rate)");
ylabel("p(k)");
print ImpulseResponse.png -dpng

% Plot the magnitude response
figure(2); clf;
Lfft = 1024;
Pmag = 20*log(abs(fft(p/N,1024)));
f = (0:Lfft-1) * N / Lfft;
plot(f-N/2,fftshift(Pmag)); hold on;
ax = axis;
plot([1/2 1/2],[ax(3) ax(4)],'--');
plot([1/4 1/4],[ax(3) ax(4)],'--');
plot([1/8 1/8],[ax(3) ax(4)],'--');
axis([ax(1) ax(2) -100 20]);
grid on;
title("Magnitude");
xlabel("Frequency" + newline + "(normalized by original sampling rate)");
ylabel("Gain (dBs)");
legend("Filter response","Fs/2","Fs/4","Fs/8");
print MagnitudeResponse.png -dpng
```
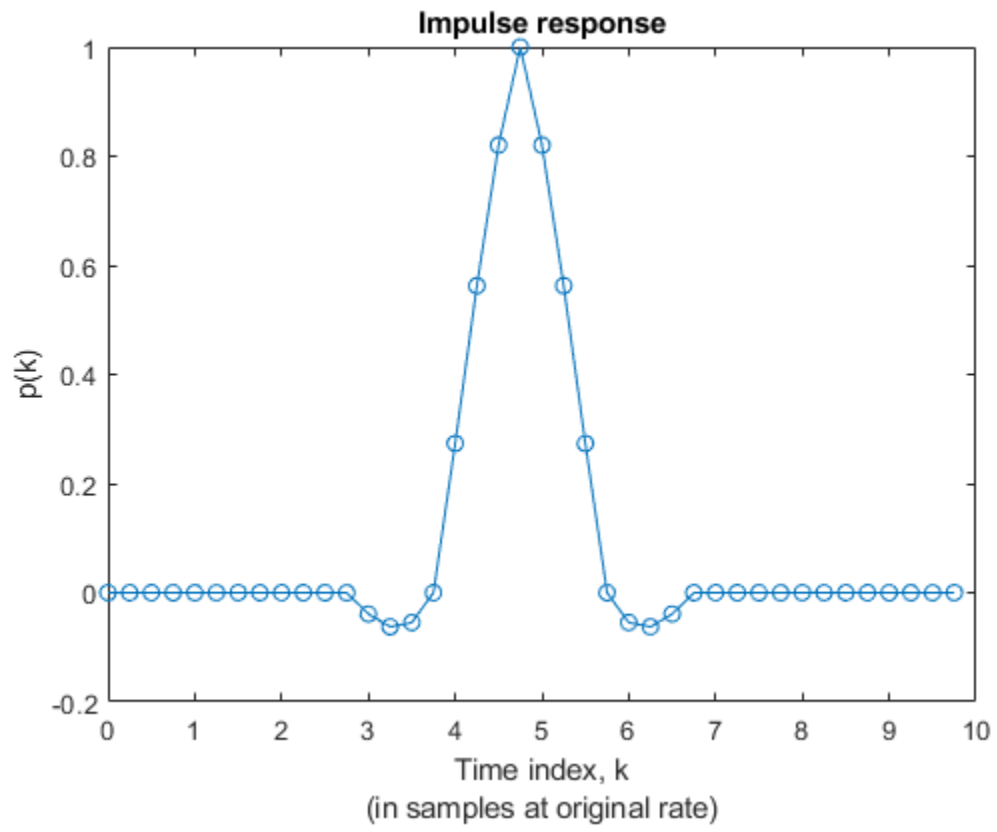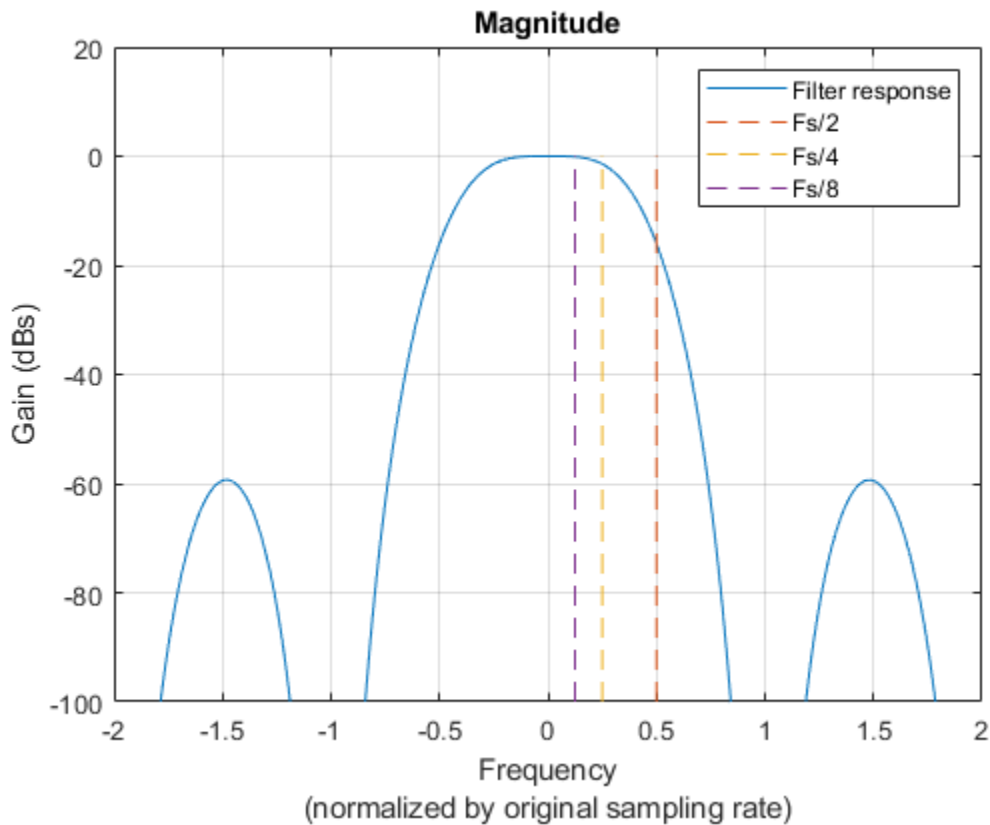
Decimating FIR Filters

Design the intermediate and final FIR filter stages. Both filters use 16-bit coefficients. For convenience, the coefficients data type is defined.

```
FIRCoeffsDT = numerictype(1,16,15);
```

Halfband Decimator

Design a halfband filter to efficiently decimate the input by 2.

```
hbParams.FsIn               = farrow.FsOut;
hbParams.FsOut              = farrow.FsOut/2;
hbParams.TransitionWidth    = hbParams.FsOut - 2*Fpass;
hbParams.StopbandAttenuation = Ast + 10;

hbSpec = fdesign.decimator(2,'halfband',...
    'Tw,Ast',...
    hbParams.TransitionWidth, ...
    hbParams.StopbandAttenuation,...
    hbParams.FsIn);

halfband = design(hbSpec,'SystemObject',true);

halfband.FullPrecisionOverride = false;
halfband.CoefficientsDataType  = 'Custom';
halfband.CustomCoefficientsDataType = numerictype([],...
    FIRCoeffsDT.WordLength,FIRCoeffsDT.FractionLength);
```
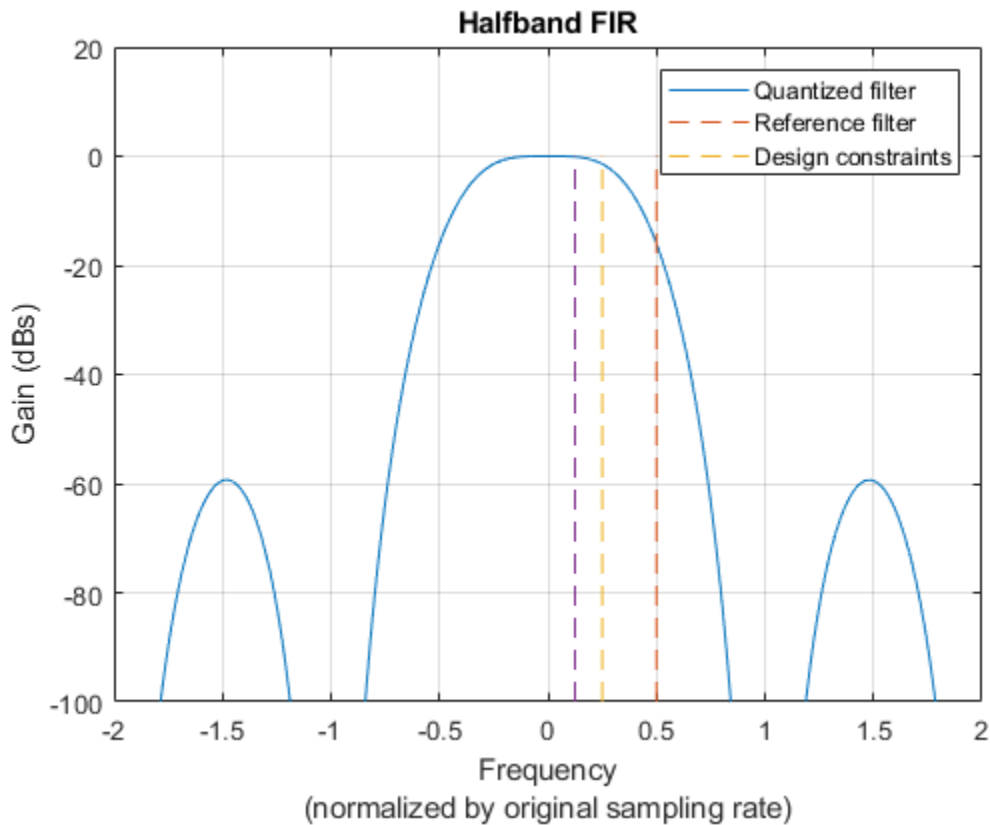
Plot the frequency response of the filter, including the quantized response.

```
srcPlots.halfband = fvtool(halfband,'arithmetic','fixed');
SRCTestUtils.setPlotNameAndTitle('Halfband FIR');
legend('Quantized filter','Reference filter','Design constraints');
```
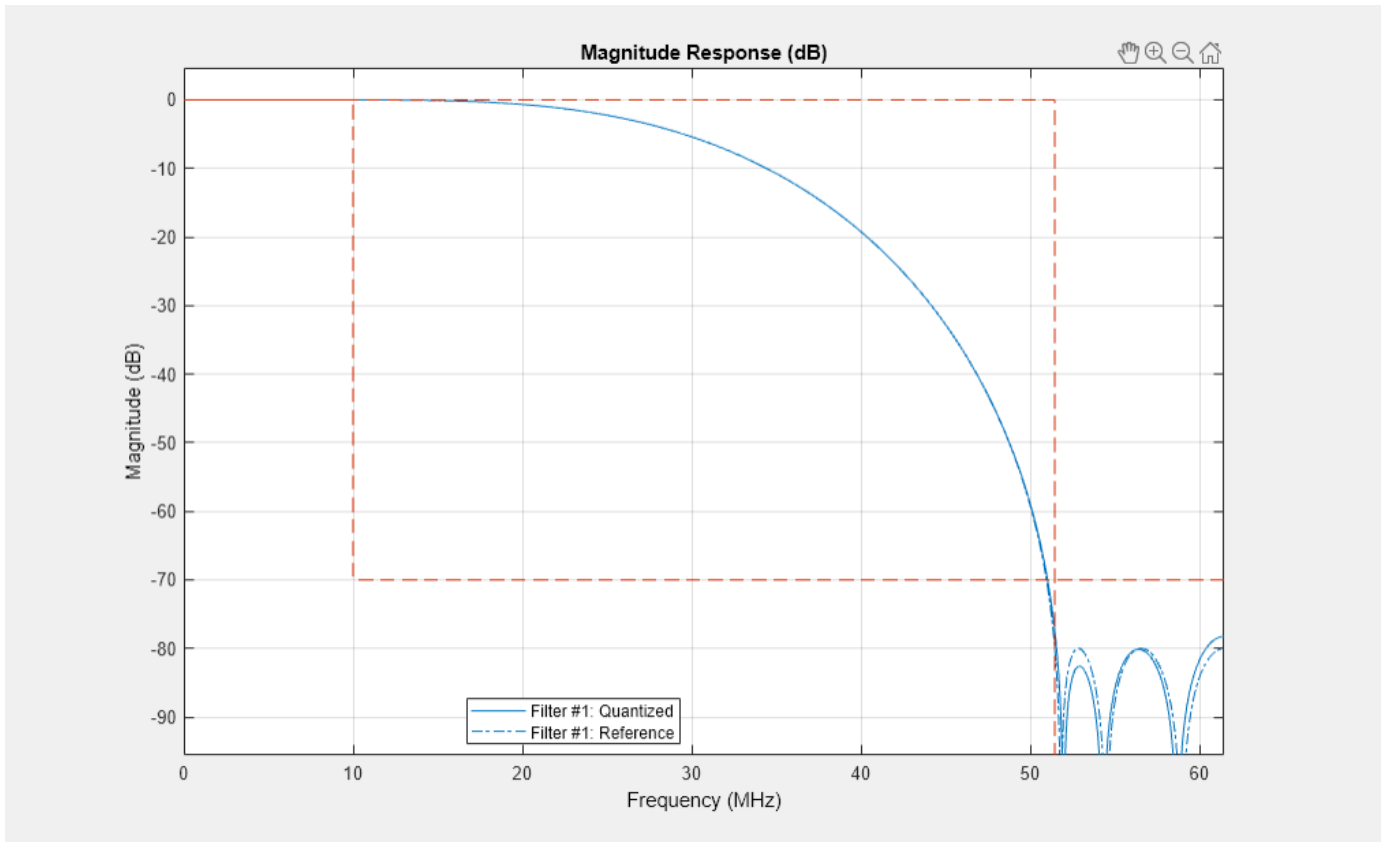
Final FIR Decimator

Design the final decimate-by-2 FIR filtering stage.

```
finalSpec = fdesign.decimator(2,'lowpass',...
    'Fp,Fst,Ap,Ast',Fpass,Fstop,Ap,Ast,hbParams.FsOut);

finalFilt = design(finalSpec,'equiripple','SystemObject',true);

finalFilt.FullPrecisionOverride = false;
finalFilt.CoefficientsDataType  = 'Custom';
finalFilt.CustomCoefficientsDataType = numerictype([],...
    FIRCoeffsDT.WordLength,FIRCoeffsDT.FractionLength);
```
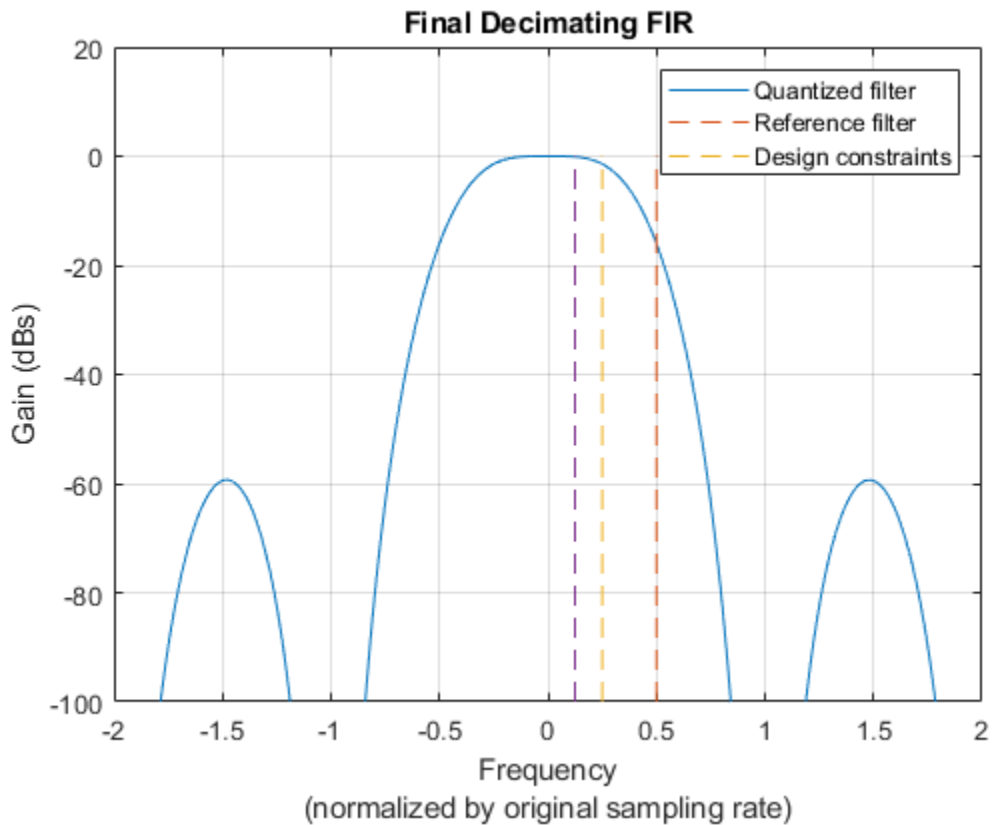
Plot the frequency response of the filter, including the quantized response.

```
srcPlots.finalFilt = fvtool(finalFilt,'arithmetic','fixed');
SRCTestUtils.setPlotNameAndTitle('Final Decimating FIR');
legend('Quantized filter','Reference filter','Design constraints');
```
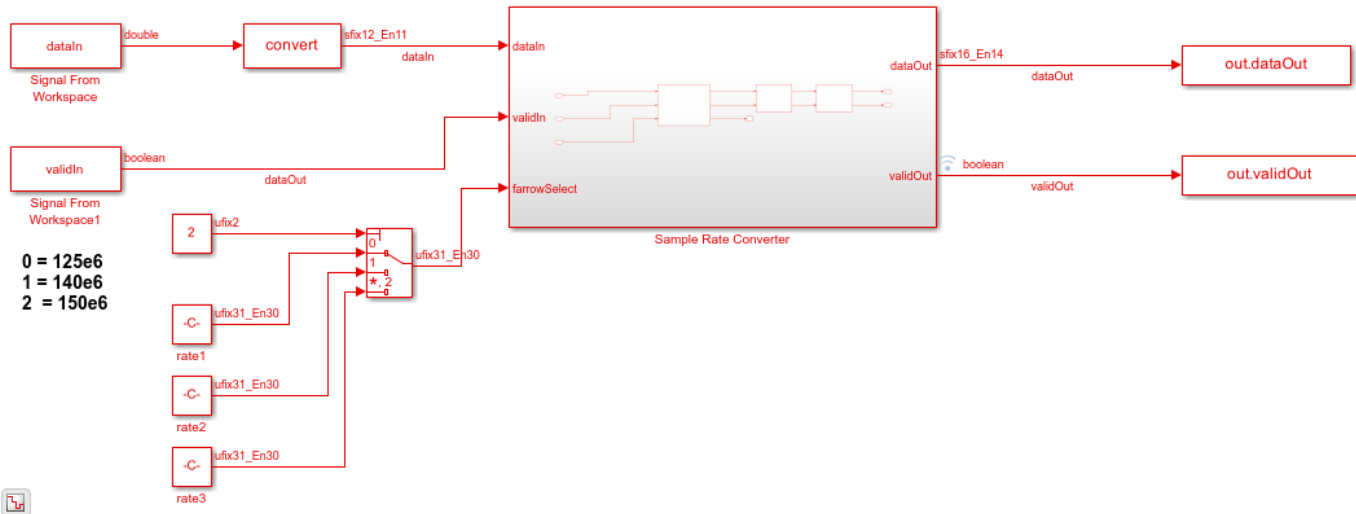
Final Decimating FIR
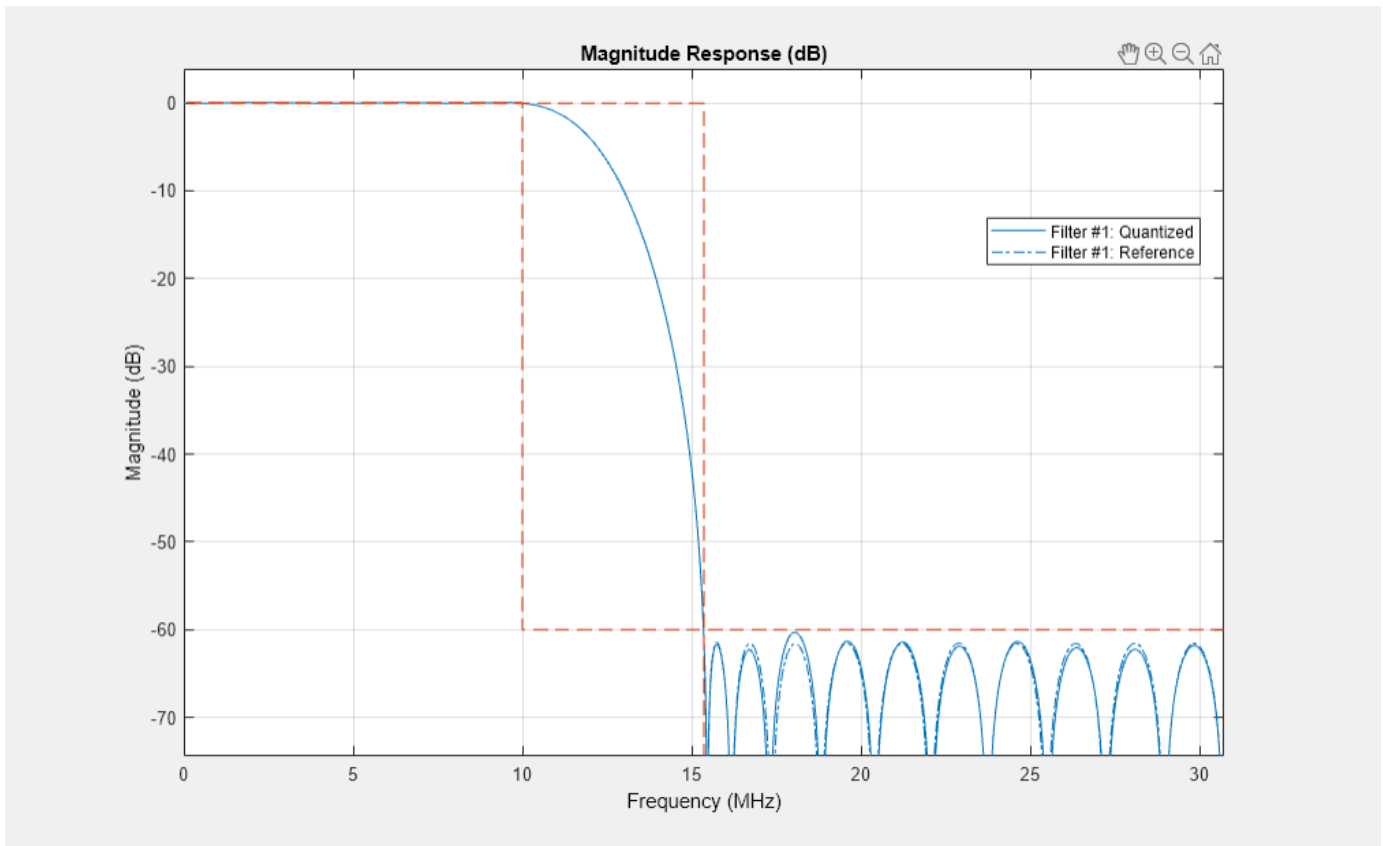
Simulink HDL Implementation

Open the model and update the diagram. The top level of the model is shown. HDL code can be generated for the **Sample Rate Converter** subsystem.

```
stopTime  = 0;
dataIn    = 0;
validIn   = false;
modelName = 'SampleRateConversionHDL';
open_system(modelName);
set_param(modelName,'SimulationCommand','Update');
set_param(modelName, 'Open','on');
```

As discussed, the sample rate converter contains a Farrow rate converter, a halfband filter, and a final FIR decimation stage. The Farrow has a ready signal, which is not used in decimation and therefore is terminated. When the overall rate change corresponds to interpolation, the ready is useful for pacing the input. The two FIR Decimator blocks are configured to use the coefficients previously designed and stored in FIR System objects. As mentioned previously, the final FIR Decimator can use resource sharing, seeing as the input is only valid one every 2 cycles. This is configured by setting "Minimum number of cycles between valid input" to 2.

```
set_param([modelName '/Sample Rate Converter'],'Open','on');
```
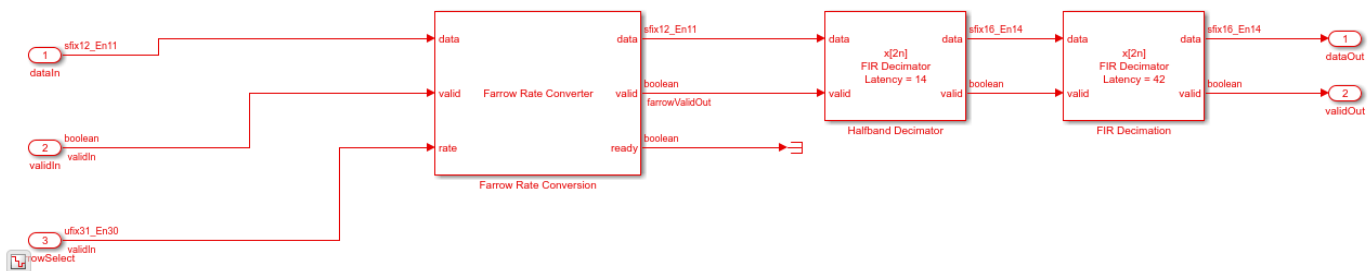

Validation and Verification

An LTE test signal is generated at 150 Msps and passed through the rate converter. An Error Vector Magnitude (EVM) measurement is then performed, confirming that the resampler is suitable for use in an LTE receiver. For reference, three different methods are used to resample the signal to 30.72 Msps and their EVM results compared. The three methods are:

1. The MATLAB resample function.
2. A MATLAB model of the rate converter.
3. The Simulink HDL model of the rate converter.

In addition, to confirm correct operation of the HDL implementation, the root-mean-square error between the outputs of the MATLAB and Simulink rate converter models is computed.

Generate a 20 MHz LTE test signal sampled at 150 Msps.

```
rng(0);
enb               = lteRMCDL('R.9');
enb.TotSubframes = 2;
[tx, ~, sigInfo] = lteRMCDLTool(enb,randi([0 1],1000,1));
dataIn = resample(tx,FsADC,sigInfo.SamplingRate);
dataIn  = 0.95 * dataIn / max(abs(dataIn));
validIn = true(size(dataIn));
```

Use the `resample` function to resample the received signal from the ADC rate to 30.72 Msps. This provides a good quality reference to compare to the rate converter.

```
resampleOut = resample(dataIn,FsLTERx,FsADC);
```

Pass the signal through a MATLAB model of the rate converter.

```
farrowFilt        = dsp.FarrowRateConverter(farrow.FsIn,farrow.FsOut);
farrowOut = step(farrowFilt,dataIn);
halfbandOut       = halfband(farrowOut);
floatResamplerOut = finalFilt(halfbandOut);
```

Pass the signal through the fixed-point Simulink HDL implementation model.

```
stopTime        = (length(dataIn)+1000)/FsADC;
simOut          = sim(modelName);
fiResamplerOut = simOut.dataOut(simOut.validOut);
fiResamplerOut = fiResamplerOut(1:length(floatResamplerOut));
```

Plot `validIn` and `validOut` to show the overall rate change of the sample rate converter. `validIn` is always HIGH, whereas `validOut` is HIGH about a quarter (0.24576%) of the time.

```
srcPlots.validSignals = figure;
Ns = 300;
validInSlice = validIn(1:Ns);
validOutSlice = simOut.validOut(1:Ns);
subplot(2,1,1);
plot((0:Ns-1)/FsADC,validInSlice);
axis([0 (Ns-1)/FsADC -0.1 1.2]);
ylabel('validIn');
xlabel('time');
subplot(2,1,2);
plot((0:Ns-1)/FsADC,validOutSlice);
axis([0 (Ns-1)/FsADC -0.1 1.2]);
ylabel('validOut');
xlabel('time');
```
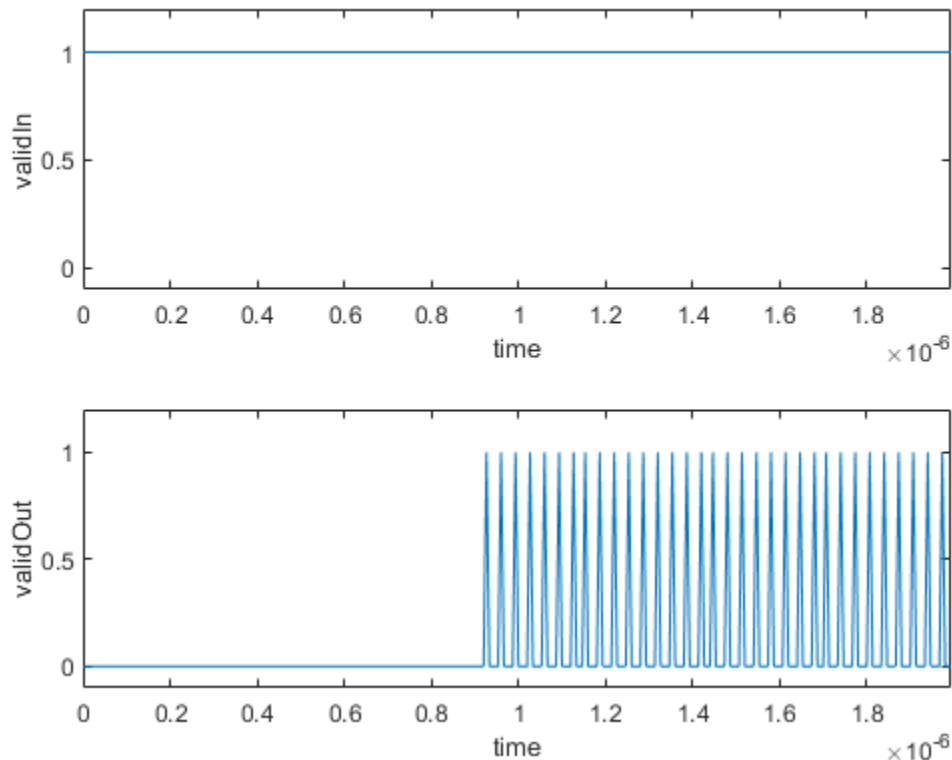
Compute the root mean square error between the outputs of the MATLAB and Simulink models of the rate converter

```
e = floatResamplerOut-fiResamplerOut;
rootMeanSquareError = sqrt((e' * e)/length(e));
disp(['Root-mean-square error: ' num2str(rootMeanSquareError)]);
```

```
Root-mean-square error: 0.26432
```

Measure the EVM of all three resampling methods.

```
results.resampleEVM                    = SRCTestUtils.MeasureEVM(sigInfo,resampleOut,FsLTERx);
results.floatPointSRCEVM               = SRCTestUtils.MeasureEVM(sigInfo,floatResamplerOut,FsLTEF
[results.fixedPointSRCEVM,fiEqSymbols] = SRCTestUtils.MeasureEVM(sigInfo,fiResamplerOut,FsLTERx)

disp('LTE Error Vector Magnitude (EVM) Measurements');
disp(['     resample function RMS EVM: '  num2str(results.resampleEVM.RMS*100,3) ' %']);
disp(['    resample function Peak EVM: ' num2str(results.resampleEVM.Peak*100,3) ' %']);
disp(['    floating point SRC RMS EVM: '  num2str(results.floatPointSRCEVM.RMS*100,3) ' %']);
disp(['   floating point SRC Peak EVM: ' num2str(results.floatPointSRCEVM.Peak*100,3) ' %']);
disp(['    fixed point HDL SRC RMS EVM: '  num2str(results.fixedPointSRCEVM.RMS*100,3) ' %']);
disp(['   fixed point HDL SRC Peak EVM: ' num2str(results.fixedPointSRCEVM.Peak*100,3) ' %']);
```

```
LTE Error Vector Magnitude (EVM) Measurements
     resample function RMS EVM: 0.0138 %
    resample function Peak EVM: 0.0243 %
    floating point SRC RMS EVM: 0.0265 %
   floating point SRC Peak EVM: 0.0645 %
```

```
  fixed point HDL SRC RMS EVM: 0.0518 %
 fixed point HDL SRC Peak EVM: 0.246 %
```

Confirm that the signal quality is high by plotting the equalized pilot symbols from the EVM measurement of the HDL implementation. Note that almost no blurring of the constellation points is visible.

```
srcPlots.scatterPlot = scatterplot(fiEqSymbols);
SRCTestUtils.setPlotNameAndTitle('Equalized Cell RS');
```
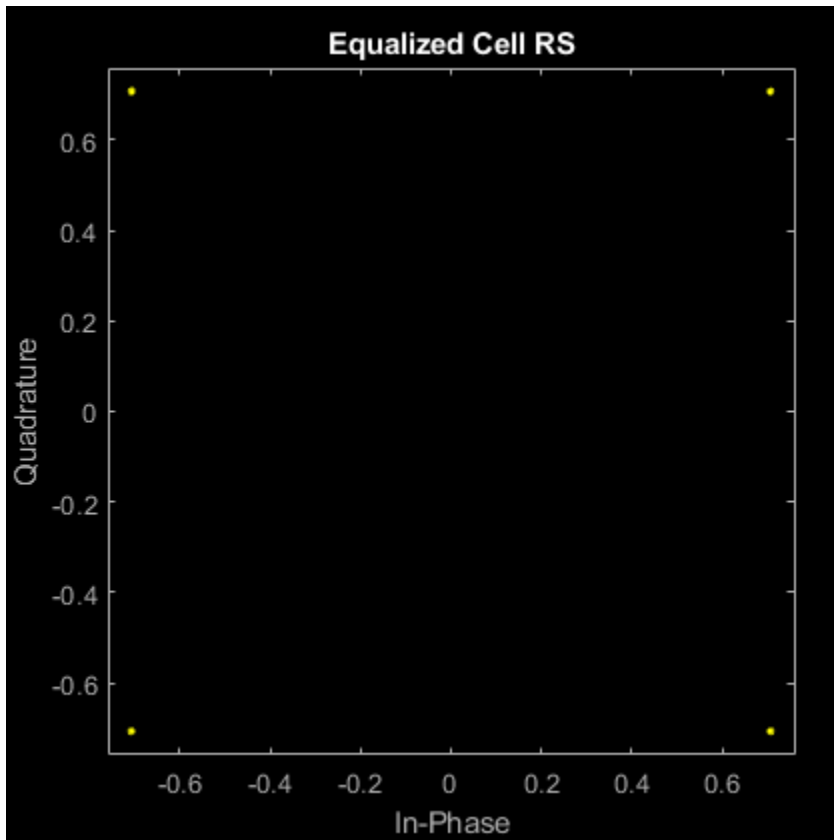

HDL Code Generation and FPGA Implementation

To generate the HDL code for this example you must have an HDL Coder™ license. Use the `makehdl` and `makehdltb` commands to generate HDL code and an HDL testbench for the **Sample Rate Converter** subsystem. The resulting HDL code was synthesized on a Xilinx® Zynq®-7000 ZC706 evaluation board. The post place and route resource utilization results are shown in the table. The design met timing with a clock frequency of 170 MHz. The critical path is the accumulator in the Farrow Rate Converter. This is implemented in fabric, and has a large wordlength of 32 bits, in order to achieve a high precision rate conversion. To improve timing, either reduce the accumulator wordlength, or map the accumulator to a DSP slice on FPGA.

```
disp(table(...
    categorical({'LUT'; 'LUTRAM'; 'FF'; 'BRAM'; 'DSP'}),...
    categorical({'1489'; '139'; '7165'; '0'; '64'}),...
    'VariableNames',{'Resource','Usage'}));

    Resource    Usage
    _____    _____
```

```
LUT        1489
LUTRAM     139
FF         7165
BRAM       0
DSP        64
```

HDL Code Generation for Filtered OFDM (F-OFDM) Transmitter

Filtered OFDM (F-OFDM) applies a filter to the symbols after the IFFT in the transmitter to improve bandwidth while maintaining the orthogonality of the complex symbols. This example implements a transmitter F-OFDM for HDL code generation. The example shows how to go from a MATLAB® reference model to an HDL-optimized Simulink® model. It includes converting from double to fixed-point types, and minimizing the resource use of the design on an FPGA.

Refer to "F-OFDM vs. OFDM Modulation" for comparison between OFDM and F-OFDM waveforms.

System Parameters

Set the desired F-OFDM properties.

```
NDLRB              = 108;
WaveformType       = 'F-OFDM';
SubcarrierSpacing  = 60*1e3; %Hz
CellRefP           = 1;
CyclicPrefix       = 'Normal';
FilterLength       = 513;
ToneOffset         = 2.5000;
CyclicExtension    = 'off';
```

Call the `h5gOFDMInfo` function to calculate F-OFDM parameters. The method calculates FFT length, cyclic prefix lengths and number of subcarriers.

```
genb = struct('NDLRB', NDLRB,...
              'WaveformType', WaveformType,....
              'SubcarrierSpacing', SubcarrierSpacing*1e-3,...
              'FilterLength', FilterLength,...
              'ToneOffset', ToneOffset,...
              'CellRefP', CellRefP,...
              'CyclicPrefix', CyclicPrefix,...
              'CyclicExtension', CyclicExtension);
info = h5gOFDMInfo(genb);
```

Generate a Grid of Input Data

```
QAMModulation   = '64QAM';
TotSubframes    = 5;
[txgrid, bitsIn] = generateOFDMGrid(genb,info,QAMModulation,TotSubframes);
```

Reference MATLAB Model

The reference model runs a floating-point F-OFDM system and plots the spectrum. Use the reference model to compare against the fixed-point model that supports HDL code generation.

```
[txSig_ref,txinfo] = h5gOFDMModulate(genb,txgrid);
```

Model the channel by adding noise to the signal.

```
snrdB = 18;
S = RandStream('mt19937ar','Seed',1);
rxSig_ref = awgn(double(txSig_ref),snrdB,'measured',S);
```

The received signal must be synchronized and aligned. In real situations, the receiver includes symbol synchronization. In this example, the receiver corrects for the shift of the frame by the transmitter filter by $\frac{FilterLength}{2}$.

```
rxSig_ref_sync = circshift(rxSig_ref,-floor(FilterLength/2));
```

Recover data, calculate BER, and display constellation.

```
[constDiagRx, ber, rxgrid_ref] = FOFDM_Receiver(rxSig_ref_sync, bitsIn, genb,...
                                 QAMModulation, 'F-OFDM Reception (REF)');
disp(['F-OFDM Reception (REF)', ' BER = ' num2str(ber(1)) ' at SNR = ' num2str(snrdB) ' dB']);
constDiagRx(rxgrid_ref(:));
```

```
F-OFDM Reception (REF) BER = 0.0094568 at SNR = 18 dB
```
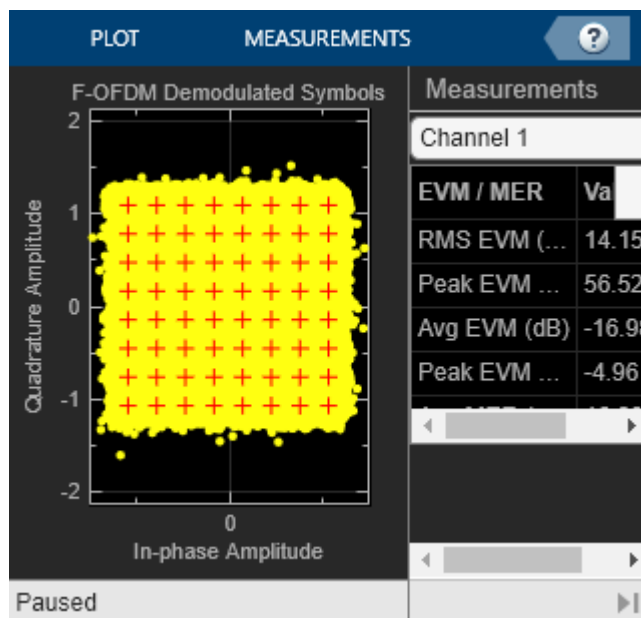

The spectrum shows clear improvement of out-of-band radiation of the subband signal, and increase in effective bandwidth.

```
FOFDMTransmitterHDLSpectrum(txSig_ref,txinfo,genb,'F-OFDM Spectrum (REF)');
```

108 RBs. Subcarrier spacing: 60kHz.
Sample rate: 122.88 Msps

F-OFDM Spectrum (REF)

BW: 77.8 MHz
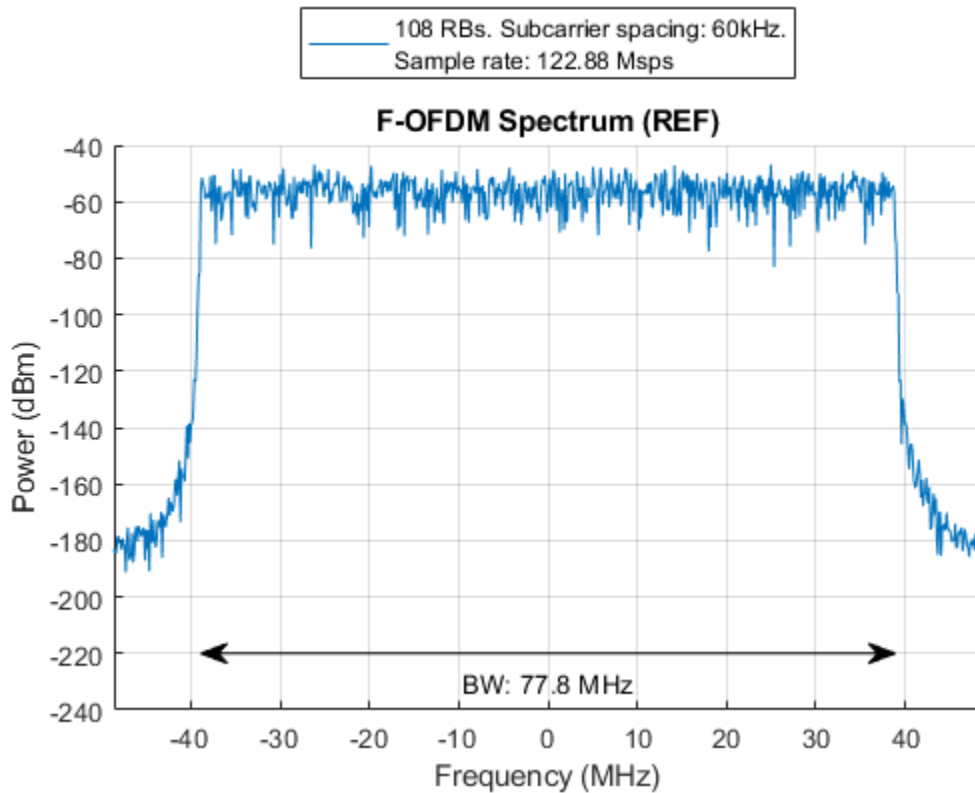
Simulink Fixed-point Model

```
model = 'FOFDMTransmitterHDLExample_FixPt';
load_system(model);
open_system([model, '/F-OFDM']);
```
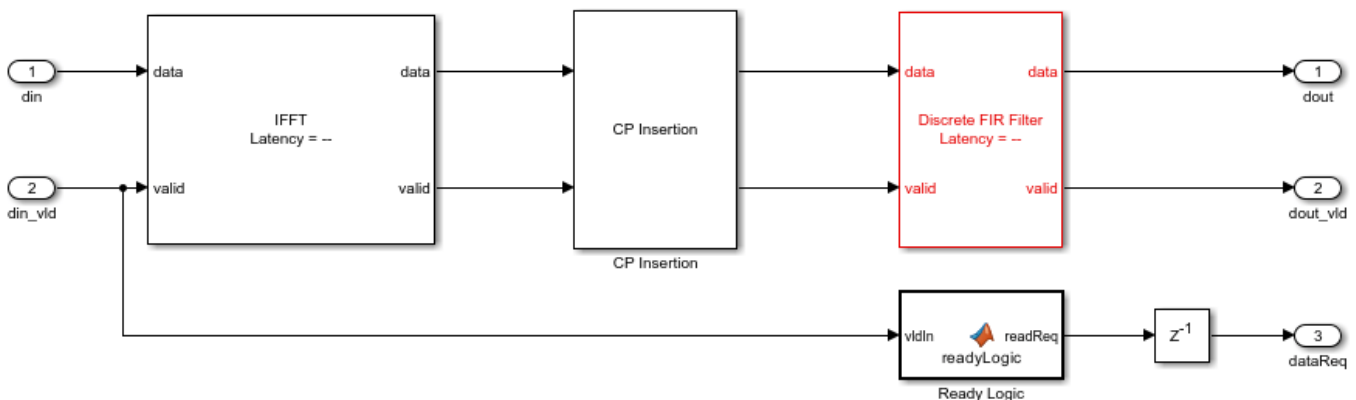

To generate HDL from the model, fixed-point data type must be used instead of double. For 64-point QAM, at least 6 bits + 1 sign bit is needed. However, to achieve reasonable BER, the input word length must be increased, considering the FPGA's limitation. Multipliers in FPGAs have limited input word length. For example, Xilinx's DSP48 has 18*25-bit multiplier. For an optimal design, a wordlength is chosen so that all multipliers in the FFT and the filter are smaller than 18*25-bit

multipliers. In this example, the FFT block uses the "Divide butterfly outputs by two" option. The input word length is 16 bits.

You can run the Simulink model with floating point data by setting WORDLENGTH=-1. However, this mode is not supported for HDL code generation.

```
WORDLENGTH = 16;
```

Set the number of fractional bits to WORDLENGTH - 2 bits to cover -1 <= Symbol <= 1.

```
FRACTIONLENGTH = WORDLENGTH - 2;
```

Generating OFDM Symbols

The input data to the IFFT is assumed to be a proper OFDM symbol and resides in a memory (OFDM Symbol subsystem in the model) that can be read by F-OFDM Subsystem. Therefore, the transmitter's sample rate depends on the data availability in the memory and FPGA clock frequency. If the data is available all the time, then the sample rate is limited to

$$clock\_frequency * \frac{FFTLength}{(FFTLength + Max(CyclicPrefixLength))}.$$

On the other hand, the required sample rate is calculated by $SubcarrierSpacing * FFTLength$ and it is equal to 122.88 Msps for this example. To achieve 122.88 Msps the clock frequency should be at least 135.36 MHz.

```
ifftin = generateOFDMSymbol(txgrid,info,genb);
```

Filter Design

The appropriate filter should have a flat passband over the subcarriers and sharp transition to minimize guard bands. It also needs sufficient stopband attenuation. A prototype filter $w = w_1 * w_2$ is used, where w_1 is a SINC function and

$$w_2 = o.5 * (1 + cos(\tfrac{2*\pi*n}{N-1})).$$

```
fnum = generateFilterCoef(genb,info);
```

Simulation

Set up the model and run. Note that due to the system latency, the model needs to be simulated longer to collect enough data.

```
Nfft = info.Nfft;
CyclicPrefixLengths = info.CyclicPrefixLengths;
SymbolsPerSubframe = info.SymbolsPerSubframe;

STOPTIME = 4 * TotSubframes * info.SamplesPerSubframe;

sim(model);
txSig_fixpt = TX_WAVEFORM(1: size(txSig_ref));
```

Model the channel by adding some noise to the signal. Note that the same noise is used as in the reference MATLAB model.

```
S = RandStream('mt19937ar','Seed',1);
rxSig_fixpt = awgn(double(txSig_fixpt),snrdB,'measured',S);
```

Perform symbol synchronization, recover data, calculate BER, and display constellation.

```
rxSig_fixpt_sync = circshift(rxSig_fixpt,-floor(genb.FilterLength/2));

[constDiagRx,ber,rxgrid_fixpt] = FOFDM_Receiver(rxSig_fixpt_sync,bitsIn, ...
                                 genb, QAMModulation,'F-OFDM Reception (FIXED-POINT)');
disp(['F-OFDM Reception (FIXED-POINT)',' BER = ' num2str(ber(1)) ' at SNR = ' num2str(snrdB) ' dE
constDiagRx(rxgrid_fixpt(:));
```
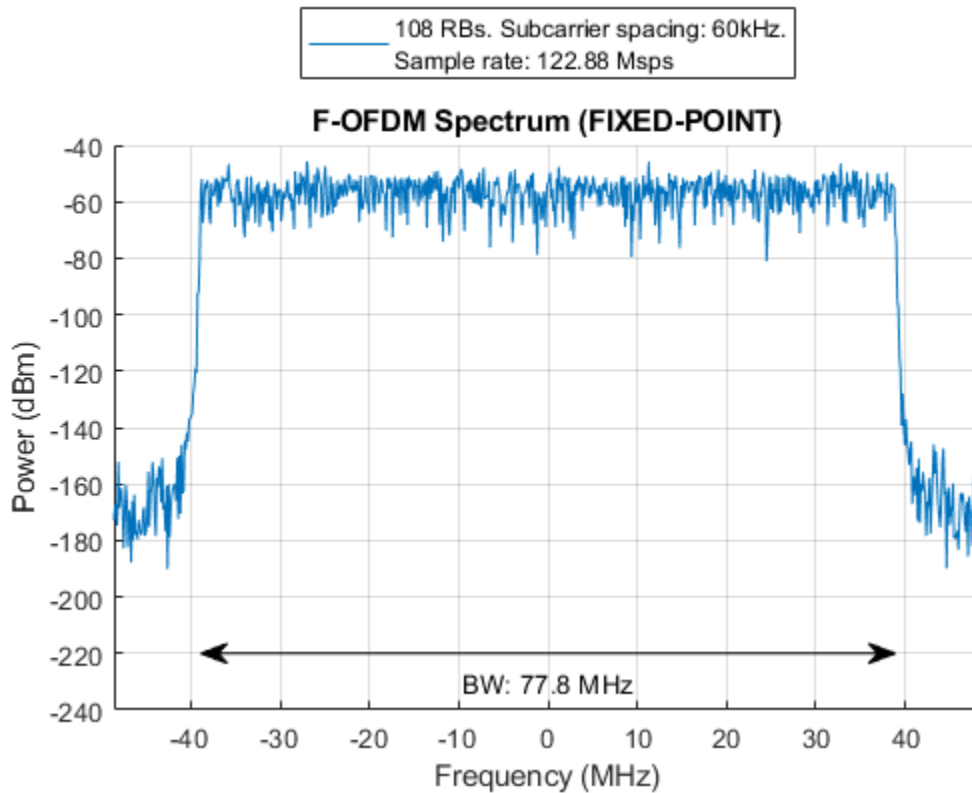
F-OFDM Reception (FIXED-POINT) BER = 0.0094453 at SNR = 18 dB
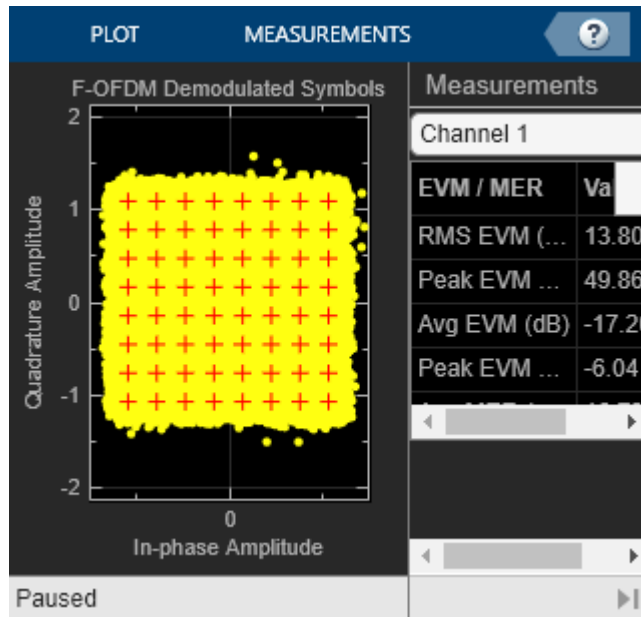
The spectrum shows even for fixed point a clear improvement of out-of-band radiation of the subband signal, and increase in effective bandwidth.

```
FOFDMTransmitterHDLSpectrum(txSig_fixpt,txinfo,genb,'F-OFDM Spectrum (FIXED-POINT)');
```

Simulink HDL-Optimized Model

The fixed point model uses a 513-tap filter in the time domain. This filter requires 2*513 multipliers since the output of IFFT is complex. Even when implemented using a symmetric filter it needs 513 multipliers which is too many multipliers for a normal size FPGA. To reduce the number of multipliers in the filter, the model filters in the frequency domain. A frequency domain FIR filter requires FFT of the input multiplied by FFT of the coefficients and then IFFT the result. The number of complex multipliers in this case is

$$2 * ceil(\frac{log2(FFTLength)}{2}) - 1.$$

The frequency domain filter in this example uses 11 complex multipliers. Note that the actual number of real multipliers depends on FFT and IFFT block setting (Complex multiplication option) and word length. In the model, the time domain FIR filter is replaced by a frequency domain FIR filter implemented with an overlap-save architecture. Due to overlapping characteristic of the overlap-save architecture, the sample-rate is limited to

$$clock\_frequency * \frac{FFTLength}{(FFTLength+Max(CyclicPrefixLength)+2*(FilterLength-1))}.$$

Therefore, to achieve 122.88 Msps sample-rate for this example, the clock frequency must be at least 196.8 MHz.

```
model = 'FOFDMTransmitterHDLExample_HDLOpt';
load_system(model);
open_system([model, '/F-OFDM']);
```

4-21
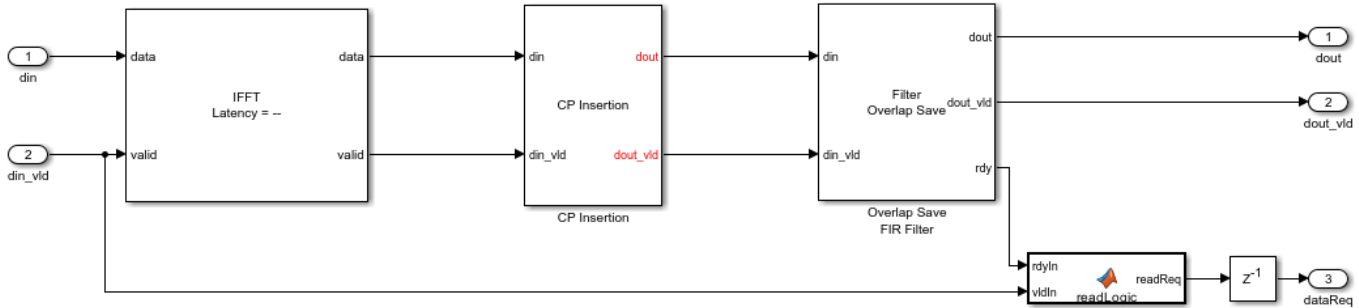
Set the length of the FFT for the filter. The length must be at least 2*FilterLength for frequency domain filtering. However, because it must process the whole OFDM symbol at once use Nfft for FFT length inside the filter. Then, calculate the FFT of the coefficients. Bit-reverse the result since the output of the FFT for the filter is bit-reversed.

```
filterFFTLen = Nfft;
fftFnum = bitrevorder(fft(fnum,filterFFTLen).');
```

For fixed-point input data, the output of the FFT inside the filter has a bit-growth = log2(Nfft) = 11 bits. To map most of the multipliers into DSP block in FPGA, limit the input word length. For example if DSP has a 25*18-bit multiplier, the WORDLENGTH must be 14 bits to achieve 25-bits output of the FFT inside the filter. Also, use 18-bit coefficients.

```
WORDLENGTH = 14;
FRACTIONLENGTH = WORDLENGTH - 2;
if WORDLENGTH > 0 %for fixed point data
    COEF_WL = 18;
    COEF_FR = COEF_WL - 2;
    fftFnum = fi(fftFnum, 1, COEF_WL, COEF_FR,'RoundingMethod','Nearest',...
              'OverflowAction','Wrap');
end
STOPTIME = 4 * TotSubframes * info.SamplesPerSubframe;

sim(model);
txSig_HDLOpt = TX_WAVEFORM_HDLOpt(1: size(txSig_ref));
```

Model the channel by adding some noise to the signal. Note that the same noise is used as in the reference MATLAB model.

```
S = RandStream('mt19937ar','Seed',1);
rxSig_HDLOpt = awgn(double(txSig_HDLOpt), snrdB, 'measured', S);
```

Perform symbol synchronization, recover data, calculate BER, and display constellation.

```
rxSig_HDLOpt_sync = circshift(rxSig_HDLOpt,-floor(genb.FilterLength/2));

[constDiagRx,ber,rxgrid_HDLOpt] = FOFDM_Receiver(rxSig_HDLOpt_sync,bitsIn,...
                                  genb, QAMModulation, 'F-OFDM Reception (HDLOPT)');
disp(['F-OFDM Reception (HDLOPT)',' BER = ' num2str(ber(1)) ' at SNR = ' num2str(snrdB) ' dB']);
constDiagRx(rxgrid_HDLOpt(:));

F-OFDM Reception (HDLOPT) BER = 0.010038 at SNR = 18 dB
```
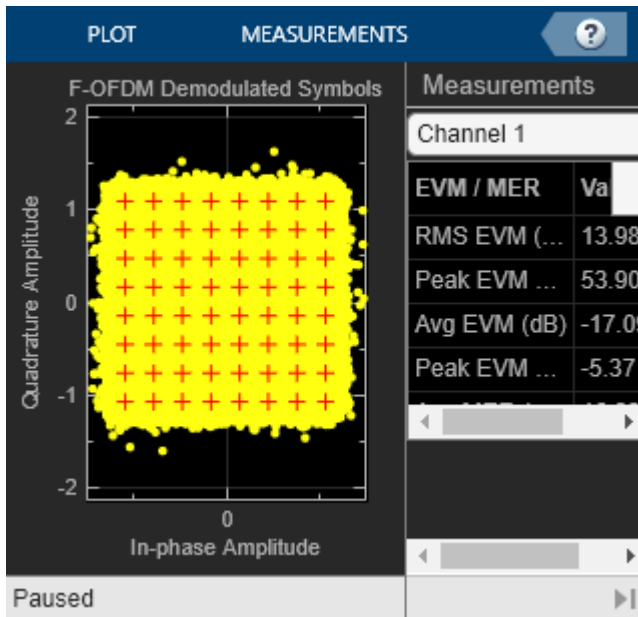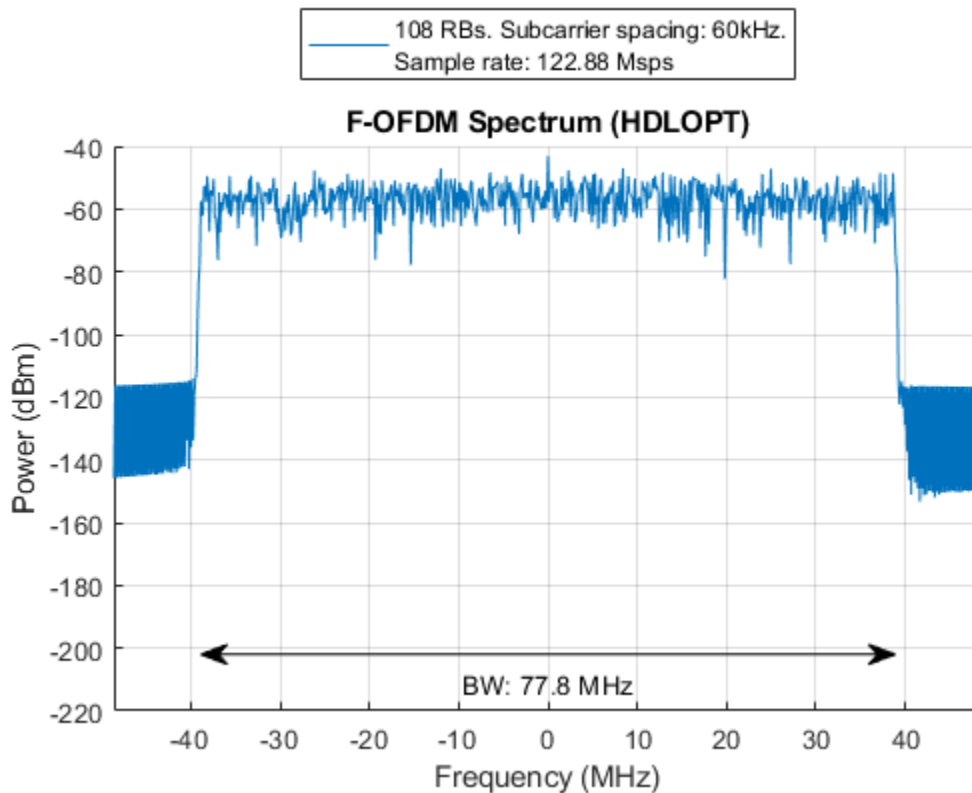
The spectrum shows even for fixed point a clear improvement of out-of-band radiation of the subband signal, and increase in effective bandwidth.

```
FOFDMTransmitterHDLSpectrum(txSig_HDLOpt,txinfo,genb,'F-OFDM Spectrum (HDLOPT)');
```

Generate HDL Code and Test Bench

Use a temporary directory for the generated files:

```
systemname = 'FOFDMTransmitterHDLExample_HDLOpt/F-OFDM';
workingdir = tempname;
```

You can run the following command to check the F-OFDM subsystem for HDL code generation compatibility:

```
checkhdl(systemname,'TargetDirectory',workingdir);
```

Run the following command to generate HDL code:

```
makehdl(systemname,'TargetDirectory',workingdir);
```

Run the following command to generate the test bench:

```
makehdltb(systemname,'TargetDirectory',workingdir);
```

Synthesis Result

The design was synthesized for Xilinx Zynq-7000 (xc7z045-ffg900, speed grade 2) using Vivado. This FPGA has 900 DSP48 slices and therefore, the fixed-point version of the design doesn't fit in this device. The HDL-optimized version of the design fits in this chip and achieves a clock frequency of 205.8 MHz which meets the required clock frequency of 196.8 MHz. The design uses 94 DSP48 (10%) and 24 block RAMs (4%).

Conclusion

In this example a Simulink fixed-point model was developed and optimized for hardware. The model minimized resource usage by optimizing use of DSP on the FPGA. Comparing the results of the floating-point model with the fixed-point model shows that 16-bit data has a similar bit error rate to the floating-point data.

See Also

Related Examples
- "F-OFDM vs. OFDM Modulation"

HDL Implementation of Variable-Size FFT

This example shows how to implement a variable-size FFT by using a single FFT core.

This example includes two models *VariableSizeFFTHDLExample* and *VariableSizeFFTArbitraryValidPatternHDLExample* that show variable-size FFT implementations for different input valid patterns.

Many popular standards like WLAN, WiMax, digital video broadcast (DVB), digital audio broadcast (DAB), and long term evolution (LTE) provide multiple bandwidth options. The required FFT length for OFDM modulation and demodulation for these standards varies with bandwidth option. For example, LTE supports different channel bandwidth options from 1.4 MHz to 20 MHz, which require FFT lengths of 128 to 2048 respectively. The FFT block generates HDL code for a specific FFT length. This example demonstrates how to use the FFT block to implement a variable-size FFT.

This example generates input data in MATLAB® and imports it to Simulink® for the simulation. The imported data is fed to the implementations of variable-size FFT using a single FFT and multiple FFTs. To demonstrate that the single-FFT implementation matches the results of using multiple FFTs of various sizes, both the output streams from the Simulink simulation are exported to MATLAB and compared.

Model Architecture

The top-level subsystem in both the models implement a variable-sized FFT. The top subsystem uses a single FFT block and the bottom subsystem provides reference data by using multiple FFT blocks of various sizes.

The model *VariableSizeFFTHDLExample* can process data with a gap between valid samples, provided the gap depends on FFT length.

```
modelname = 'VariableSizeFFTHDLExample';
open_system(modelname);
```

Variable Size FFT using Single FFT

Multiple FFTs for Reference
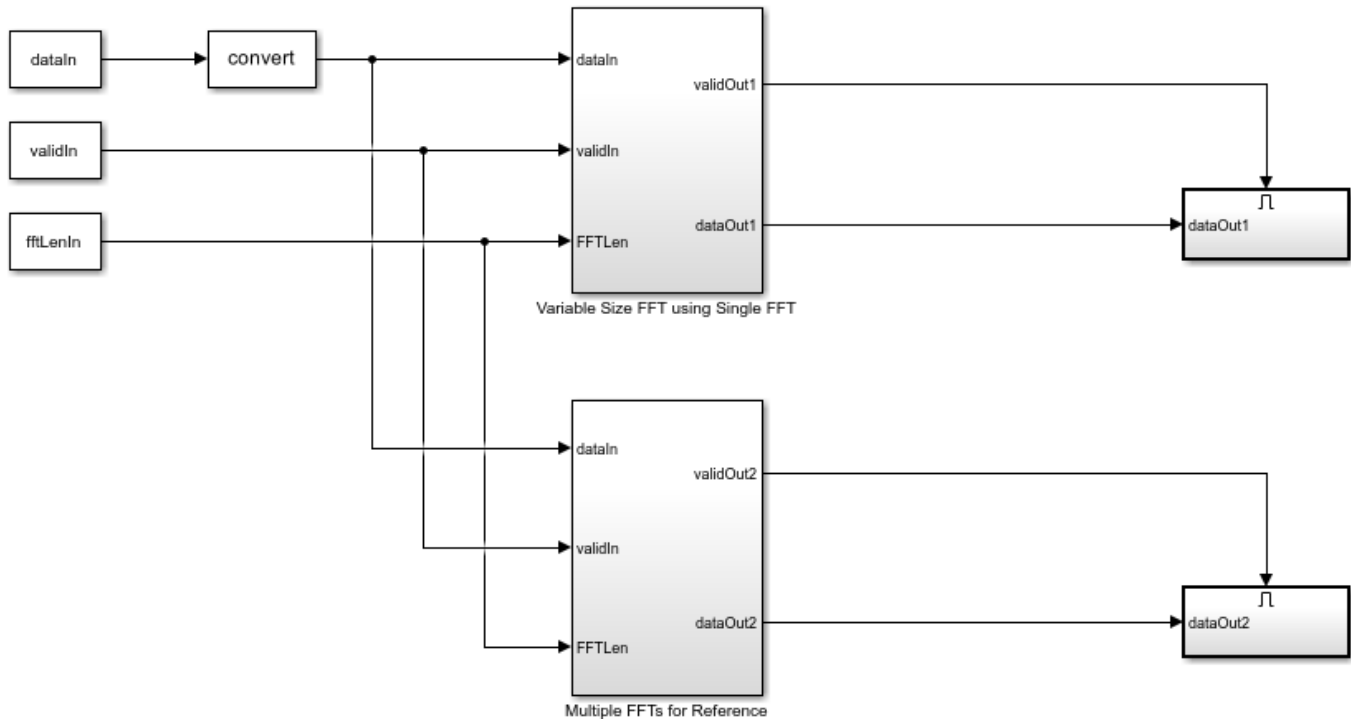
Configuration of FFT Lengths

The FFT lengths are specified through a variable `fftLenVecMulFFTs`. The largest of these lengths is stored in a variable `fftLenSinFFT` and used as the FFT length for the FFT block in the `'Variable Size FFT using Single FFT'` subsystem.

The input `fftLenIn` is generated by using the vector of FFT lengths specified in `fftLenVecMulFFTs`.

```
fftLenVecMulFFTs = [128;256;512;1024;2048];
% Single FFT length used by variable size FFT.
fftLenSinFFT = max(fftLenVecMulFFTs);
% Generate |fftLenIn| by repeating each element of |fftLenVecMulFFTs| by
% |fftLenSinFFT| times and arranging in a single column.
fflen =repmat(fftLenVecMulFFTs.',fftLenSinFFT,1);
fftLenIn = uint16(fflen(:));
```

Input Generation

`dataIn`, `validIn`, and `fftLenIn` inputs are generated in MATLAB and imported to the Simulink model. Random complex input data `randInputData` is generated for each of the FFT lengths specified in `fftLenVecMulFFTs`. Different FFT lengths correspond to different bandwidths and different sampling rates. For instance, in LTE, the FFT lengths of 128, 256, 512, 1024, and 2048 correspond to the sampling rates 1.92 MHz, 3.84 MHz, 7.68 MHz, 15.36 MHz, and 30.72 MHz respectively. The symbol time for any FFT length is $66.67 \mu s$. The example operates at the highest rate among the FFT lengths specified.

The `dataIn` signal is generated by padding zeros in between the `randInputData` samples. The figure below shows the input data and valid patterns for `fftLenVecMulFFTs` of 256 and 512 and

fftLenSinFFT being 2048. For the FFT length of 256, the example inserts 7 invalid samples for every valid sample and for the FFT length of 512, the code inserts 3 invalid samples for every valid sample.

The model *VariableSizeFFTHDLExample* requires the input valid pattern to have a gap between valid samples as shown in the figure below.
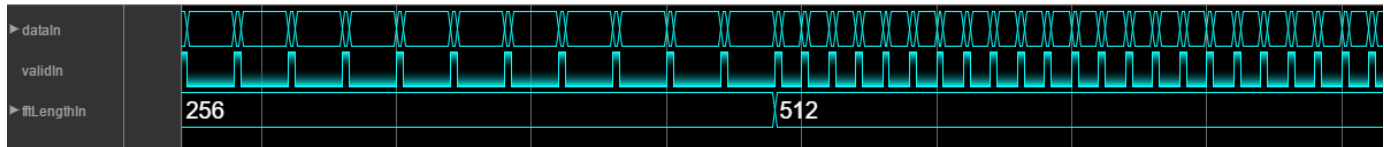


```matlab
rng('default');
dataIn = zeros(length(fftLenVecMulFFTs)*fftLenSinFFT,1);
validIn = false(length(fftLenVecMulFFTs)*fftLenSinFFT,1);
% Loop over the FFT lengths
for ind = 1:length(fftLenVecMulFFTs)
    % Generate data of FFT length samples
    randInputData = complex(randn(1,fftLenVecMulFFTs(ind)),randn(1,fftLenVecMulFFTs(ind)));
    % Zero padding in between input data samples
    upSamplingFac = fftLenSinFFT/fftLenVecMulFFTs(ind);
    dataIn((ind-1)*fftLenSinFFT+1:fftLenSinFFT*ind) = upsample(randInputData,upSamplingFac);
    % Valid corresponding to the generated data
    tempValid = true(1,fftLenVecMulFFTs(ind));
    validIn((ind-1)*fftLenSinFFT+1:fftLenSinFFT*ind) = upsample(tempValid,upSamplingFac);
end
inputDataType = 'fixdt(1,16,14)';  % Input data type can be modified here
set_param('VariableSizeFFTHDLExample/Data Type Conversion','OutDataTypeStr', inputDataType);
% Get FFT latency
fftObj = dsp.HDLFFT('FFTLength',fftLenSinFFT,...
    'Architecture','Streaming Radix 2^2',...
    'ComplexMultiplication','Use 3 multipliers and 5 adders',...
    'BitReversedOutput',false,...
    'BitReversedInput',false,...
    'Normalize',false);
latency=getLatency(fftObj); % Default latency is 4137 for 2048 point FFT.
additionPipelineDelay = 6; % Number of additional pipeline delays
% Simulink simulation end time Total Latency = Latency of FFT + Latency of
% data controller (5 clock cycles).
% Total simulation running time = Total
% number of input samples + Total Latency + Pipeline delay.
simTime = fftLenSinFFT*(length(fftLenVecMulFFTs) + 1) + latency + additionPipelineDelay ;
```

Variable-Size FFT using Single FFT

The 'Variable-Size FFT using Single FFT' design includes a Data Controller, an FFT block, and a Bin selection subsystem.

```matlab
open_system([modelname '/Variable Size FFT using Single FFT']);
```
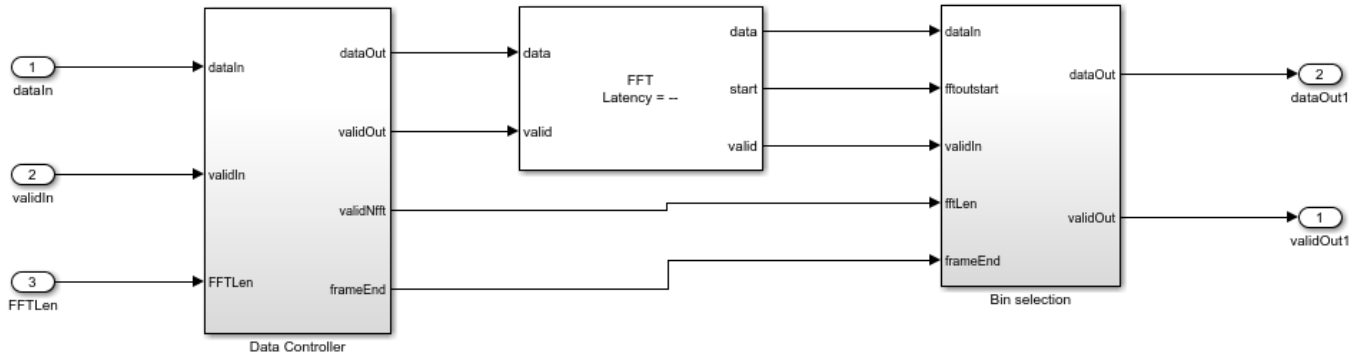
The `Data Controller` subsystem controls the input data so that the input to the FFT block has data samples with zeros padded in between them. The FFT block is configured for an FFT length of 2048, the largest FFT length required by the LTE standard. To simplify selection of the output bins, the FFT block is configured to output the samples in bit-natural order. The FFT length is specified through input port and is sampled at the start of the frame. The requested FFT length must be delayed to match the FFT latency. The FFT length is registered using the start output signal of the FFT and the generated end of the frame signal. This method avoids implementing a large delay-matching memory. Since the input data has zeros in between samples, the output of the large FFT contains repeated copies of the FFT length samples. To get the required FFT output, the first FFT length samples are collected from the FFT output. This operation is performed by modifying the output valid signal of the FFT using the `Bin selection` subsystem.

Multiple FFTs for Reference

This subsystem is used as a reference to compare against the output of Variable Size FFT using Single FFT. The subsystem includes five different FFT blocks (FFT 128, FFT 256, FFT 512, FFT 1024, and FFT 2048) and one MATLAB Function block. The input data will be fed to all five FFTs. Depending on the requested FFT length, one of the five FFT blocks is activated and FFT operation is performed. The MATLAB function block `pickFFTData` selects the output from the appropriate FFT block. The output is saved to MATLAB for comparison with the output of the Variable-Size FFT using Single FFT.

```
open_system([modelname '/Multiple FFTs for Reference']);
```
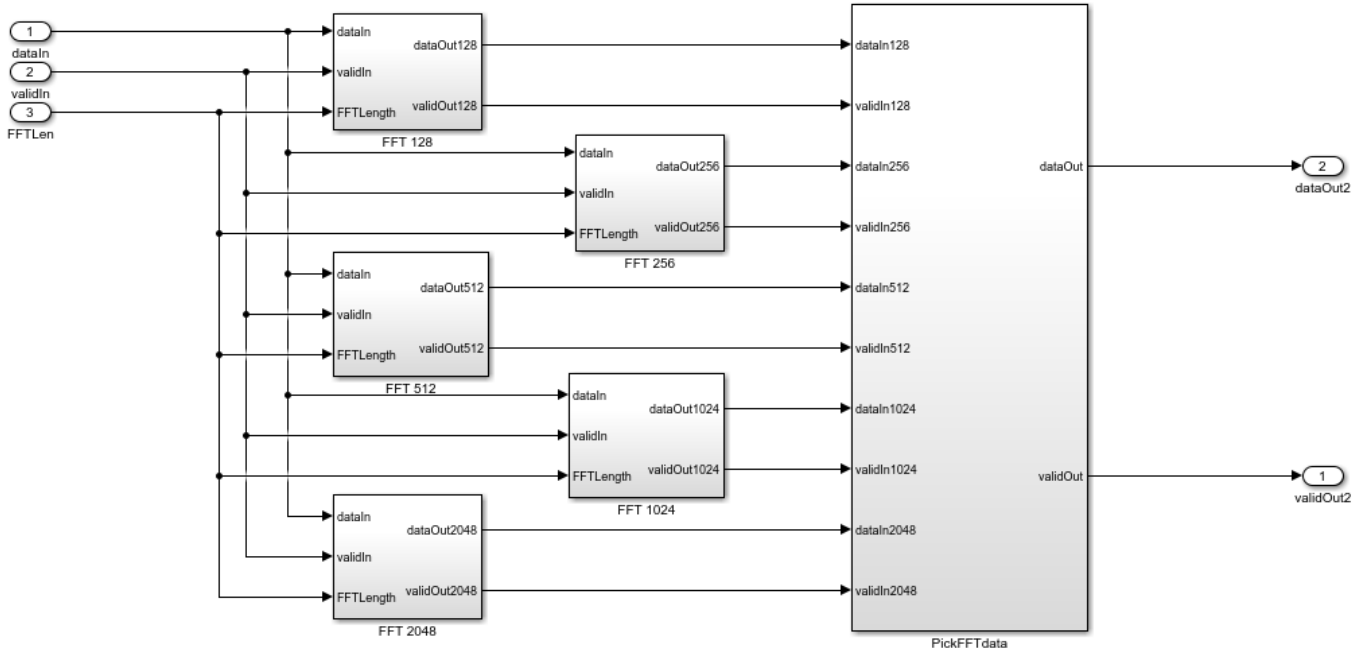
Run Simulink Model

The MATLAB script configures desired vector of FFT lengths, the size of the single FFT, and generates input data with a valid signal. It then runs the model, and compares the output of the two subsystems in MATLAB.

Run the model using the `sim` command on the MATLAB command line.

```
sim(modelname);
```

Verification

The output from both subsystems is sent to the MATLAB workspace and the difference is plotted. In this case, the output of the two subsystems are identical and the error between the two sets of values is 0.

```
dataOut1 = out1(:);
dataOut2 = out2(:);
figVSF = figure('Visible', 'off');
plot(abs(dataOut1-dataOut2));
title('Difference between the two outputs for fixed valid pattern')
xlabel('Sample Index');
ylabel('Error');
figVSF.Visible = 'on';
bdclose(modelname);
```
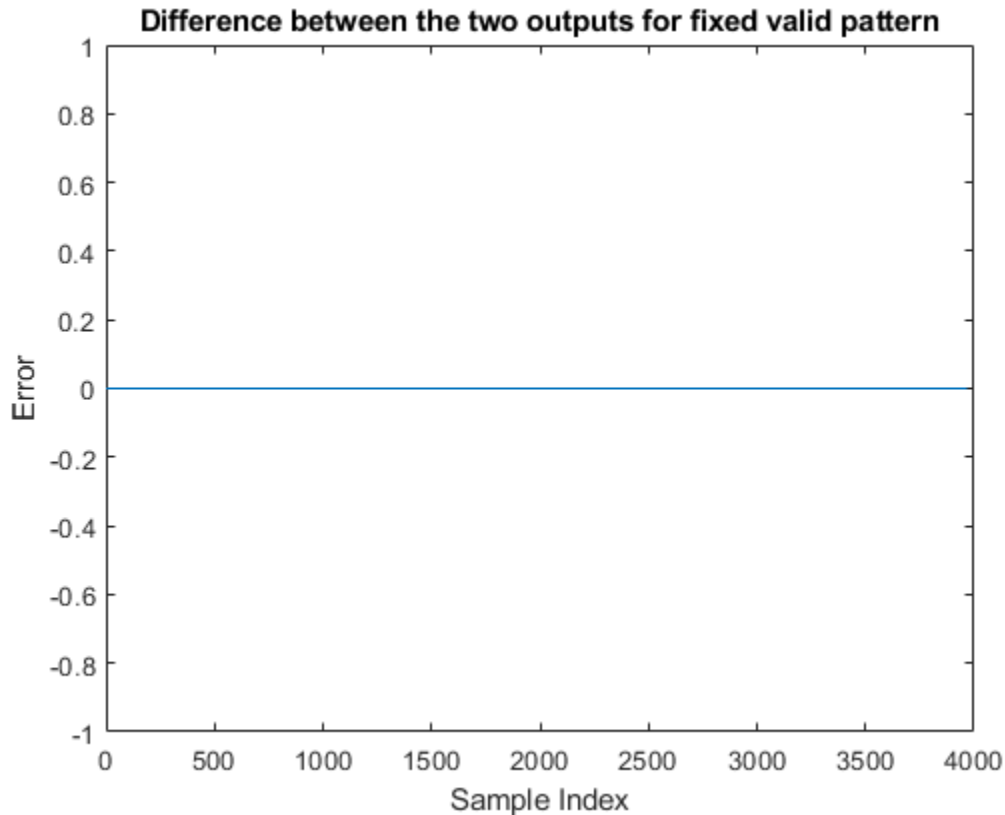
Support for Arbitrary Input Valid Patterns

The above model *VariableSizeFFTHDLExample* has a requirement of having a minimum gap between input data samples. The gap depends on the specified FFT length and the largest FFT length handled by the design. There may be cases where the input data may not conform to this pattern. For example, the data may be continuous and padded with zeros at the end of the input data samples. The following figure shows a contiguous input valid pattern with invalid samples padded at the end of input data samples for FFT lengths of 256 and 512. The single FFT length is set to 2048. In this case, the 256 valid samples are followed by 1792 invalid samples and the 512 valid samples are followed by 1536 invalid samples.
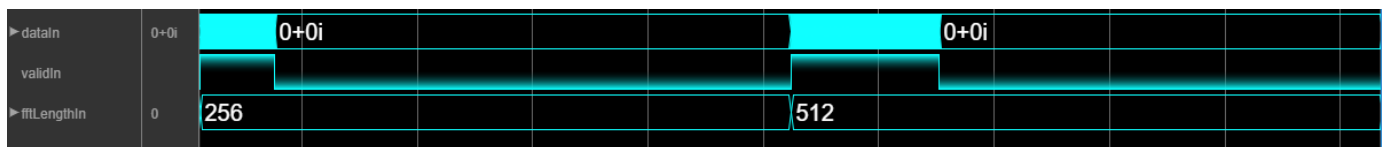
In such scenario, the design has to store the input samples into a RAM, and pad invalid samples between valid samples before sending it on to the FFT. The model *VariableSizeFFTArbitraryValidPatternHDLExample* can handle any arbitrary pattern of valid input so long as the gap between frames is at least the single FFT length (2048 samples for LTE). This model is the same as the model *VariableSizeFFTHDLExample*, except for the data controller subsystem. The data controller subsystem in the model uses a RAM of size `2*fftLenSinFFT` (as shown in the figure below) to store input samples, reads out the valid samples while padding zeros between them and then passing them to the FFT. While the input data is being written into one half of the memory, the

data is read from the other half of the memory. As a result, the total latency is increased by
fftLenSinFFT.

```
modelname = 'VariableSizeFFTArbitraryValidPatternHDLExample';
load_system(modelname);
open_system([modelname '/Variable Size FFT using Single FFT/Data Controller']);
```
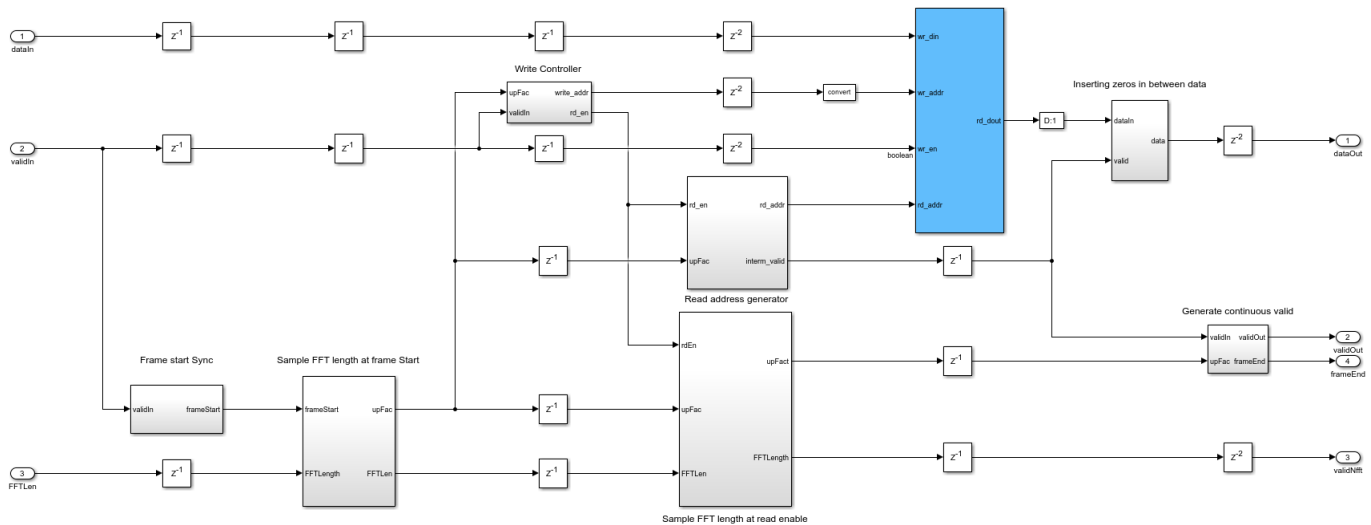

Arbitrary Input Data and Valid Generation

For generating arbitrary data and valid inputs, users can select any of these three options: zero
padding of fixed size in between data samples, zero padding at the end of data samples, and zero
padding of random size in between data samples. The input data and valid generation for these three
different zero padding patterns are shown below. The
VariableSizeFFTArbitraryValidPatternHDLExample model uses the generated data and valid for
simulation and verification.

```
% Initialization of input data and valid
dataIn = zeros(length(fftLenVecMulFFTs)*fftLenSinFFT,1);
validIn = false(length(fftLenVecMulFFTs)*fftLenSinFFT,1);
zeroPaddingPattern = 'InBetween'; %'AtEnd','Random'
switch zeroPaddingPattern
    case 'InBetween'
        % Zero padding in between input data samples
        for ind = 1:length(fftLenVecMulFFTs)
            % Generate data of FFT length samples
            randInputData = complex(randn(1,fftLenVecMulFFTs(ind)),randn(1,fftLenVecMulFFTs(ind)
            % Zero padding in between input data samples
            upSamplingFac = fftLenSinFFT/fftLenVecMulFFTs(ind);
            dataIn((ind-1)*fftLenSinFFT+1:fftLenSinFFT*ind) = upsample(randInputData,upSamplingFa
            % Valid corresponding to the generated data
            validIn((ind-1)*fftLenSinFFT+1:upSamplingFac:fftLenSinFFT*ind) = true;
        end
    case 'AtEnd'
        % Zero padding at the end of input data samples
        for ind = 1:length(fftLenVecMulFFTs)
            % Generate data of FFT length samples
            randInputData = complex(randn(1,fftLenVecMulFFTs(ind)),randn(1,fftLenVecMulFFTs(ind)
            % Zero padded data
```

```
                dataIn(((ind-1)*fftLenSinFFT+1):((ind-1)*fftLenSinFFT+fftLenVecMulFFTs(ind))) = rand
                % Valid corresponding to data generated
                validIn(((ind-1)*fftLenSinFFT+1):((ind-1)*fftLenSinFFT+fftLenVecMulFFTs(ind))) = true
            end
        otherwise % Random
            for ind =1:length(fftLenVecMulFFTs)
                % Zero padding at random
                randIndices = randperm(fftLenSinFFT);
                % Generate data of FFT length samples
                randInputData = complex(randn(1,fftLenVecMulFFTs(ind)),randn(1,fftLenVecMulFFTs(ind)
                indices = randIndices(1:fftLenVecMulFFTs(ind));
                % If the random indices does not have the first sample
                if(sum(indices==1)==0)
                    indices(1) = 1;
                end
                % Zero padded data
                dataIn(indices+(ind-1)*fftLenSinFFT) = randInputData;
                % Valid corresponding to data generated
                validIn(indices+(ind-1)*fftLenSinFFT) = true;
            end
    end
end
```

Run the Simulink model

Before running the model, make sure that `dataIn`, `validIn`, `fftLenIn`, and the necessary variables are initialized.
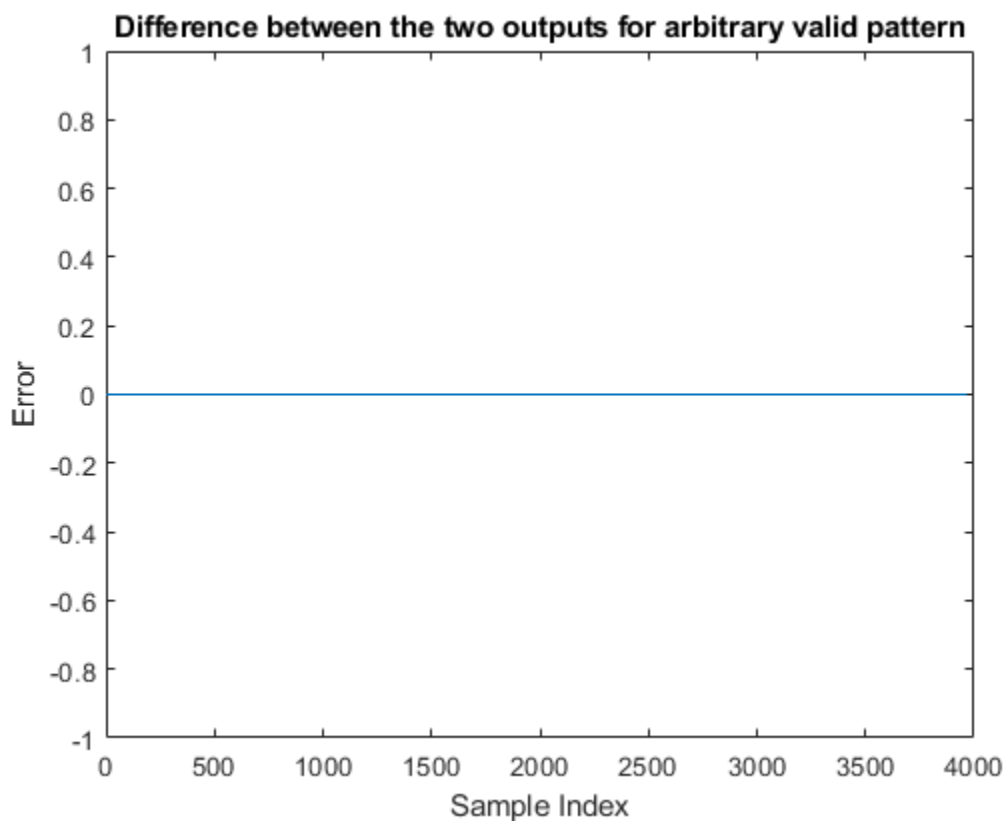
```
sim(modelname);
```

Verification

```
dataOut1 = out1(:);
dataOut2 = out2(:);
figVSFAIV = figure('Visible', 'off');
plot(abs(dataOut1-dataOut2));
title('Difference between the two outputs for arbitrary valid pattern')
xlabel('Sample Index');
ylabel('Error');
figVSFAIV.Visible = 'on';
bdclose(modelname);
```

Difference between the two outputs for arbitrary valid pattern

HDL Code Generation and Verification

To generate the HDL code referenced in this example, an HDL Coder™ license is needed.

You can use the commands `makehdl` and `makehdltb` to generate the HDL code and the testbench for the subsystems.

HDL code generated for the Variable Size FFT subsystems were synthesized for the Xilinx® Zynq®-7000 ZC706 board. The synthesis results are shown in the following table.

Hardware Type	Single FFT	Single FFT supporting arbitrary input valid pattern	Multiple FFTs
Slice LUT	5580	5732	19736
Slice Registers	8035	8082	29303
RAMB36E1	2	6	4
RAMB18E1	18	18	48
DSP48E1	16	16	58
Max Freq (in MHz) Post P&R	265	237.6	243.1

4-33

The table above shows that implementing a variable-size FFT using a single FFT uses fewer hardware resources than using a multiple FFT solution. To support an arbitrary input valid pattern, the hardware implementation uses more RAM.

See Also

FFT

Accelerate BER Measurement for Wireless HDL LTE Turbo Decoder

This example shows the workflow to measure the BER of the Wireless HDL Toolbox™ LTE Turbo Decoder block using parsim to parallelize the simulations across EbNo points. This approach can be used to accelerate other Monte Carlo simulations.

Introduction

HDL implementations of reference applications are often complex and take a lot of time to simulate. As a result, figuring out the bit error rate (BER) performance by running multiple simulations at different SNR points can be very time consuming. One way to optimize this is to parallelize simulations using the `parsim` command. The `parsim` command runs multiple simulations in parallel when called with a Parallel Computing Toolbox™ license available. This example measures the BER of the LTE Turbo Decoder. To achieve sufficient statistical accuracy, around 100 errors must be obtained at the decoder for each EbNo value. This translates to 1e8 bits at a BER of 10e-6. This type of Monte Carlo simulation is a suitable candidate to parallelize using parsim, where the BER for every EbNo point is performed on workers in parallel.

For every parallel simulation, this example sets up the input data as follows:

1 Generate input data frames

2 Turbo encode

3 QPSK modulate

4 Add AWGN based on the EbNo value

5 Demodulate the noisy symbols

6 Generate soft decisions

The soft decisions become the input to the LTE Turbo Decoder in Simulink®. The turbo decoded bits are compared to the transmitted bits to calculate the BER. Each parallel simulation sends the results back to the main host.

Configure Parameters and Simulation Objects

The total number of information bits for each `EbNo` point, `bitsPerEbNo`, is divided over multiple simulations, defined by `parsimPerEbNo`. In this way, every simulation runs `bitsPerParsim` bits for a single `EbNo` point. The total number of simulations is `length(EbNo)*parsimPerEbNo`. This example is configured to run only a small number of bits for demonstration purposes. In a real scenario, you must run a sufficient number of samples through the decoder for an accurate measure of the BER at the higher `EbNo` points. When choosing these parameters, consider the memory resources available on the host. A large input data set per simulation or large number of workers could result in slow down or memory exhaustion. The structure `simParam` contains the parameters required for each simulation. This structure is sent to the simulations at a later stage.

```
EbNo = 0:0.1:1.1;
bitsPerEbNo = 1e5; %1e8;
parsimPerEbNo = 2; %10;
bitsPerParsim = ceil(bitsPerEbNo/parsimPerEbNo);

simParam.blkSize = 6144;
simParam.turboIterations = 6;
```

```
simParam.numFrames = ceil(bitsPerParsim/simParam.blkSize);          % frames per simulation
simParam.modScheme = 'QPSK';
simParam.bps = 2;                                                   % bits per symbol
tailBits = 4;                                                       % encoder property
simParam.encoderRate = simParam.blkSize/(3*(simParam.blkSize+tailBits)); % rate 1/3 Turbo code
simParam.samplesizeIn = floor(1/simParam.encoderRate);             % 3 samples in at a time
simParam.inframeSize = simParam.samplesizeIn*(simParam.blkSize+tailBits);

model = 'LTEHDLTurboDecoderBERExample';
open_system(model);
```
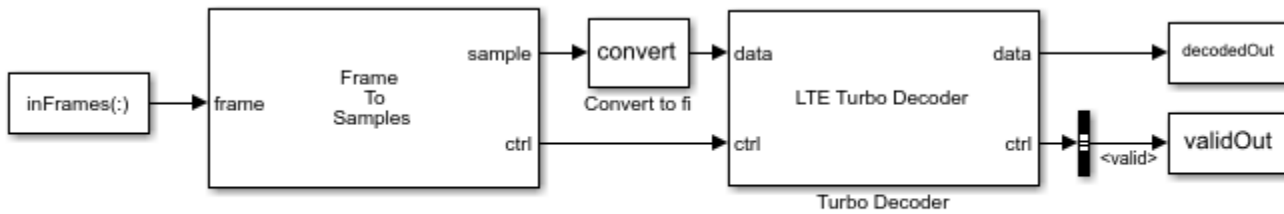

Copyright 2019 The MathWorks, Inc.

Start a local parallel pool with minimum of 1 and maximum of `maxNumWorkers`. If a Parallel Computing Toolbox™ license is not available, the simulations will be serialized. The actual size of the pool depends on the number of available cores. Each parallel worker gets assigned one core on which an independent MATLAB® session is launched.

```
maxNumWorkers = 3;
pool = parpool('local', [1 maxNumWorkers]);

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 3).
```

Preallocate a `parsim` object to hold the data required for each simulation. The object can also include handles to functions, which the model calls before or after a simulation. The MATLAB® session on which parsim is executed acts as the main host. The main host is responsible for launching the simulations on the workers, sending the required data to every worker, and receiving the results.

```
parsimIn(1:length(EbNo)*parsimPerEbNo) = Simulink.SimulationInput(model);
```

Replicate `EbNo` points to set up `parsimPerEbNo` simulations.

```
repEbNo = repmat(EbNo,parsimPerEbNo,1);
repEbNo = repEbNo(:);
```

Minimizing data transmission to the workers improves the performance and stability of the main host. Therefore, this example generates the input data in-model, rather than passing the large input data set to each worker. Input data is generated using the pre-simulation function, `presimGenInput` and the BER calculation is also performed in the post-simulation function, `postsimOutput`. These function handles are assigned to each `SimulationInput` object. The post-simulation function is assigned inside the pre-simulation function as shown in the section Pre-Simulation and Post-Simulation Functions.

```
for noiseRatio = 1:length(repEbNo)
    % Calculate the noise variance.
    EsNo = repEbNo(noiseRatio) + 10*log10(simParam.bps);
```

```
    snrdB = EsNo + 10*log10(simParam.encoderRate);
    noiseVar = 1./(10.^(snrdB/10));

    % Use random but reproducible data.
    seed = noiseRatio;

    % For Rapid Accelerator mode, set the simulation
    % stop time before compilation.
    parsimIn(noiseRatio) = parsimIn(noiseRatio).setModelParameter('StopTime',num2str(simParam.num

    % Set pre-simulation function.
    parsimIn(noiseRatio) = parsimIn(noiseRatio).setPreSimFcn(@(simIn) presimGenInput(simIn,noise\
end
```

Run and show progress of the simulations in the command window. At the end of the simulations, the results are sent back to the main host in an array of structures, `parsimOut`, with one entry created per simulation. Once simulations are complete, shut down the parallel pool.

```
parsimOut = parsim(parsimIn,'ShowProgress','on','StopOnError','on');
delete(pool);
```

```
[16-Jul-2021 12:34:57] Checking for availability of parallel pool...
[16-Jul-2021 12:34:58] Starting Simulink on parallel workers...
[16-Jul-2021 12:35:53] Configuring simulation cache folder on parallel workers...
[16-Jul-2021 12:35:53] Loading model on parallel workers...
[16-Jul-2021 12:36:06] Running simulations...
[16-Jul-2021 12:38:49] Completed 1 of 24 simulation runs
[16-Jul-2021 12:38:49] Completed 2 of 24 simulation runs
[16-Jul-2021 12:38:49] Completed 3 of 24 simulation runs
[16-Jul-2021 12:38:56] Completed 4 of 24 simulation runs
[16-Jul-2021 12:38:56] Completed 5 of 24 simulation runs
[16-Jul-2021 12:38:56] Completed 6 of 24 simulation runs
[16-Jul-2021 12:39:03] Completed 7 of 24 simulation runs
[16-Jul-2021 12:39:03] Completed 8 of 24 simulation runs
[16-Jul-2021 12:39:03] Completed 9 of 24 simulation runs
[16-Jul-2021 12:39:09] Completed 10 of 24 simulation runs
[16-Jul-2021 12:39:09] Completed 11 of 24 simulation runs
[16-Jul-2021 12:39:09] Completed 12 of 24 simulation runs
[16-Jul-2021 12:39:15] Completed 13 of 24 simulation runs
[16-Jul-2021 12:39:16] Completed 14 of 24 simulation runs
[16-Jul-2021 12:39:16] Completed 15 of 24 simulation runs
[16-Jul-2021 12:39:21] Completed 16 of 24 simulation runs
[16-Jul-2021 12:39:21] Completed 17 of 24 simulation runs
[16-Jul-2021 12:39:22] Completed 18 of 24 simulation runs
[16-Jul-2021 12:39:27] Completed 19 of 24 simulation runs
[16-Jul-2021 12:39:27] Completed 20 of 24 simulation runs
[16-Jul-2021 12:39:28] Completed 21 of 24 simulation runs
[16-Jul-2021 12:39:33] Completed 22 of 24 simulation runs
[16-Jul-2021 12:39:33] Completed 23 of 24 simulation runs
[16-Jul-2021 12:39:33] Completed 24 of 24 simulation runs
[16-Jul-2021 12:39:33] Cleaning up parallel workers...
```
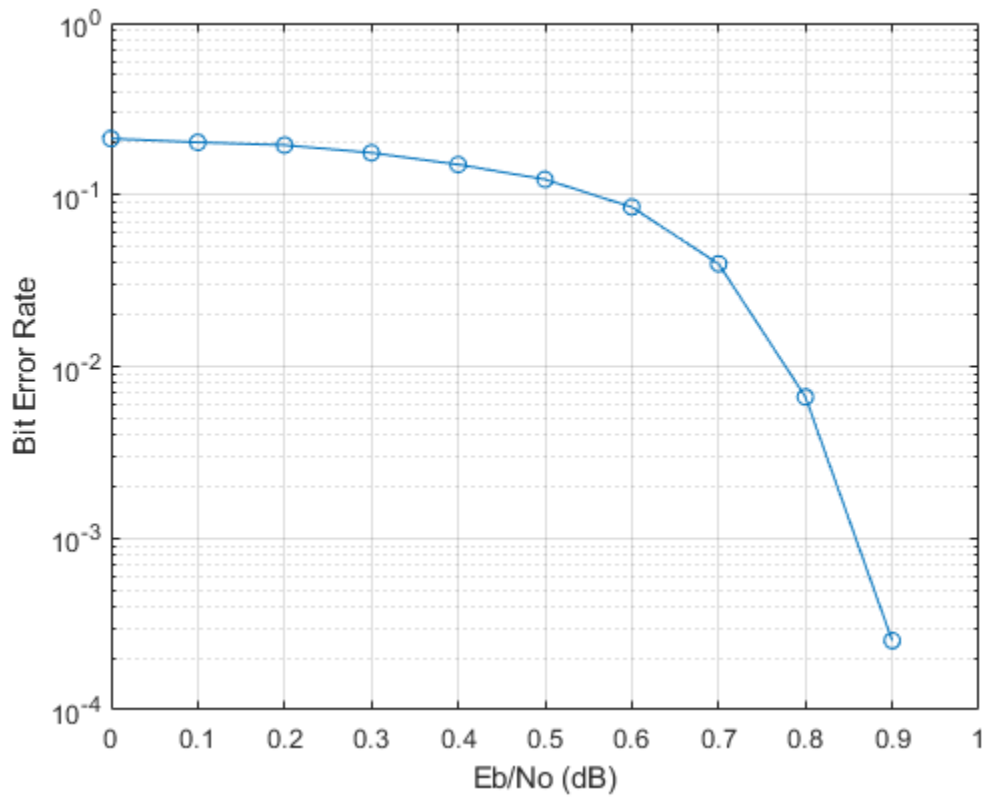
Plot BER

Extract the BER values from the array of structures. Combine the BER results for each `EbNo` point and find the average BER per `EbNo` point.
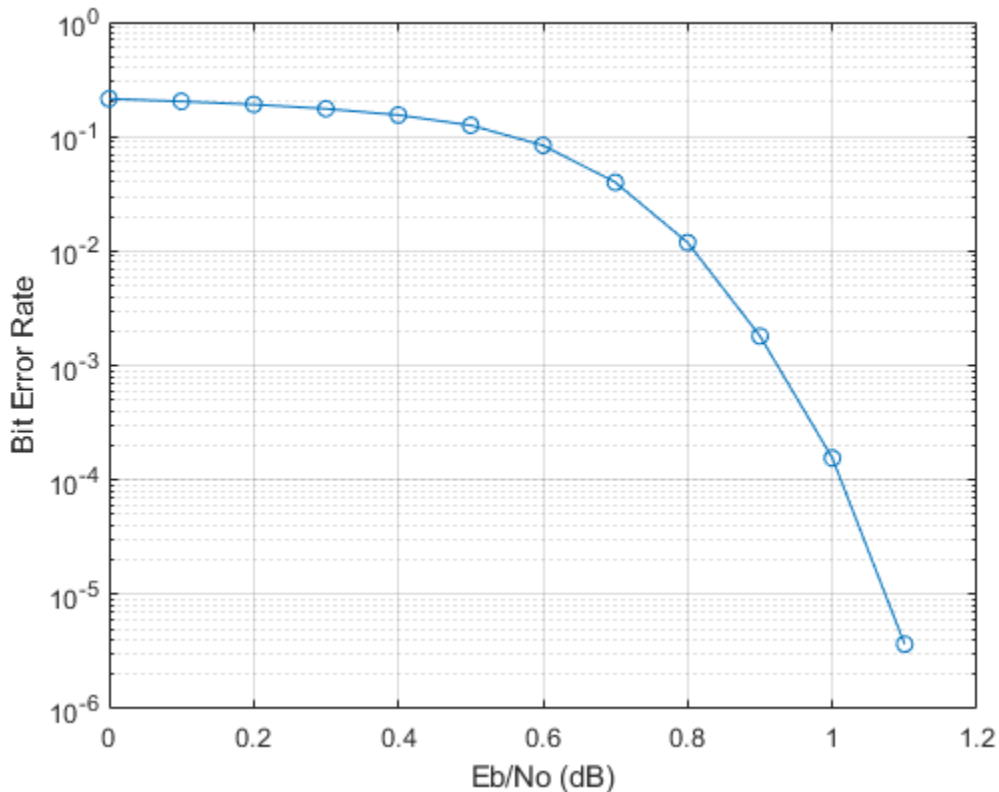
```
BER = [parsimOut(:).BER];
BER = transpose(reshape(BER,parsimPerEbNo,length(BER)/parsimPerEbNo));
```

```
avgBER = mean(BER,2);
semilogy(EbNo,avgBER,'-o');
grid;
xlabel('Eb/No (dB)');
ylabel('Bit Error Rate');
```


The plot below shows the results of the BER measurement with `bitsPerEbNo` = 1e8.

Pre-Simulation and Post-Simulation Functions

These functions independently generate input data and process output data for each simulation, which eliminates the need for the main host to store the data in memory for all simulations. The `presimGenInput` function generates input bits, then encodes, modulates and converts them to soft decisions. To make the input frames and parameters available to the model, they are assigned as variables in the global workspace using the `setVariable` function.

```
function simIn = presimGenInput(simIn,noiseVar,seed,simParam)

    rng(seed);

    % Preallocate arrays for speed.
    txBits  = zeros(simParam.blkSize,simParam.numFrames,'int8');
    inFrames = zeros(simParam.inframeSize,simParam.numFrames,'single');

    % Generate input frames, turbo encode, modulate and add noise based on
    % noise variance.
    for currentFrame = 1:simParam.numFrames
        txBits(:,currentFrame) = randi([0 1],simParam.blkSize,1);
        codedData = lteTurboEncode(txBits(:,currentFrame));
        txSymbols = lteSymbolModulate(codedData,simParam.modScheme);
        noise = (sqrt(noiseVar/2))*complex(randn(size(txSymbols)),randn(size(txSymbols)));
        rxSymbols = txSymbols + noise;
        inFrames(:,currentFrame) = lteSymbolDemodulate(rxSymbols,simParam.modScheme,'Soft');
    end
```

```
% Set up parameters for Frame to Samples block to serialize data.
% Leave sufficient gap between frames.
simParam.idleCyclesBetweenSamples = 0;
halfIterationLatency = (ceil(simParam.blkSize/32)+3)*32; % window size = 32
algFrameDelay = 2*simParam.turboIterations*halfIterationLatency+(simParam.inframeSize/simPara
simParam.idleCyclesBetweenFrames = algFrameDelay;

% Assign variables to global workspace.
simIn = simIn.setVariable('inFrames',inFrames);
simIn = simIn.setVariable('simParam',simParam);

% Set post-simulation function and send required data.
simIn =  simIn.setPostSimFcn(@(simOut) postsimOutput(simOut,txBits,simParam));

end
```

The post-simulation function receives the outputs of the simulation and computes the BER. The results are stored in a structure `results` which parsim returns as `parsimOut`.

```
function results = postsimOutput(out, txBits, simParam)
    decodedOutValid = out.decodedOut(out.validOut);

    results.numErrors = sum(xor(txBits(:),decodedOutValid));
    results.BER = results.numErrors/(simParam.numFrames*simParam.blkSize);
end
```

Conclusion

This example showed how to efficiently measure the BER curve for the Wireless HDL LTE Turbo Decoder block using parsim. If a parallel pool is not used, the linear time to complete the simulations would be approximately 16 hours. As a result of parallelization, the time to run all simulations came down to 5.4 hours, using 3 workers. This was achieved by running the simulations in Rapid Accelerator mode. This workflow can be applied to complex reference applications that require Monte Carlo or other simulations.

Encode message to RS codeword

This example shows how to use the RS Encoder block to encode a message to a Reed-Solomon (RS) codeword. In this example, a set of random inputs frames are generated and provided to the `comm.RSEncoder` System object. Using the `whdlFramesToSamples` function, these frames are converted into samples and provided as input to the RS Encoder block. The output of the RS Encoder block is then compared with the output of the `comm.RSEncoder` System object to check whether the encoded output codeword for the given input message is same. By default, the puncturing option is disabled in this example. To enable puncturing, set the puncturing value to `true`. This example model supports HDL code generation for the RS Encoder subsystem.

Set Up Input Data Parameters

Set up these workspace variable for the models to use. These variables configure the RS Encoder block inside the model.

```
nMessages = 3;
n = 255;                              % Specify codeword length
k = 239;                              % Specify message length
m = n-k;                              % Parity length
inDataType = fixdt(0,ceil(log2(n)),0);
puncturing = false;                   % true for puncturing
puncturePattern = randsrc(m,1,[0 1]); % Considered, when punturing is true
shortMsg = false;                     % true for shortened message
k1 = k-1;                             % Considered when shortMsg is true
```

Generate Random Input Samples

Generate random samples using `n`, `k`, and `m` variables and provide those generated samples as input to the `comm.RSEncoder` System object.

```
hRSEnc = comm.RSEncoder;
hRSEnc.CodewordLength = n;
hRSEnc.MessageLength = k;

if isequal(shortMsg,true)
    hRSEnc.ShortMessageLength = k1;
else
    k1 = k;
end

if isequal(puncturing,true)
    hRSEnc.PuncturePatternSource = "Property";
    hRSEnc.PuncturePattern = puncturePattern;
    puncLen = n-k-sum(hRSEnc.PuncturePattern);
else
    puncLen = 0;
end

data = cell(1,nMessages);
refData = (zeros(k1+m-puncLen,nMessages));

for ii = 1:nMessages
    data{ii} = randi([0 n],k1,1);
    refData(:,ii) = hRSEnc(data{ii});
```

```
    end

    refOutput = refData(:);
```

**Generate Input Control Samples for the Simulink® Model**

```
    gapBetweenFrames = n-k;
    gapBetweenSamples = 0;

    [simDataIn, ctrlIn] = whdlFramesToSamples(data,gapBetweenSamples,gapBetweenFrames);
    simStart = ctrlIn(:,1);
    simEnd = ctrlIn(:,2);
    simValidIn = ctrlIn(:,3);
    stopTime = length(simValidIn);
```

Run Simulink Model

Run the Simulink model. The block imports the workspace variables and generates the output.

```
    modelname = 'HDLRSEncoder';
    open_system(modelname);
    if isequal(puncturing,true)
        set_param([modelname '/RS Encoder/RS Encoder'],'PuncturePatternSource','on');
        set_param([modelname '/RS Encoder/RS Encoder'],'PuncturePattern',['[' num2str(puncturePattern
    end
    out = sim(modelname);
```
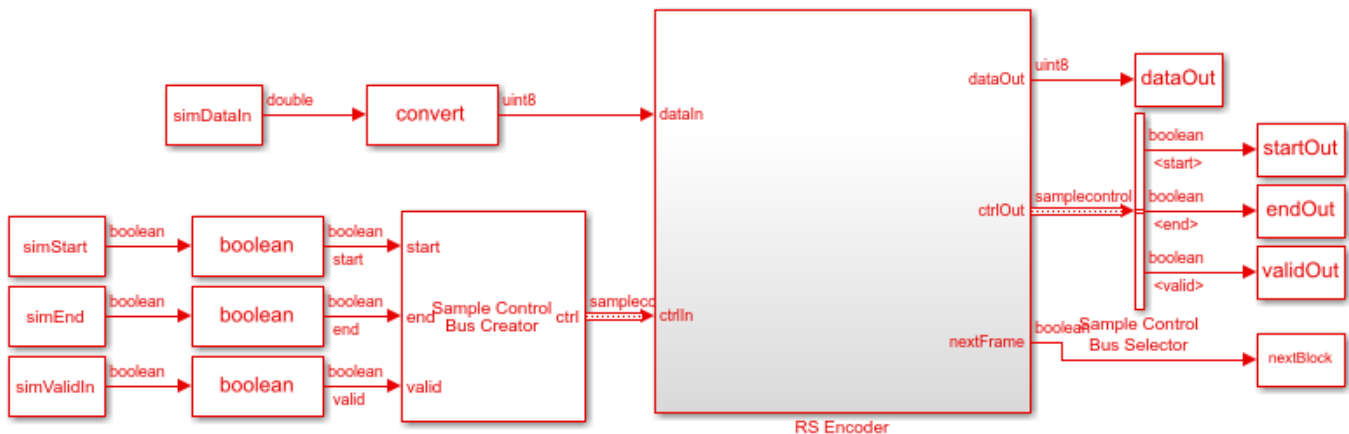

Copyright 2020 The MathWorks, Inc.

Export the Simulink Block Output to the MATLAB® Workspace

The encoded samples from the RS Encoder block are exported to the MATLAB workspace.

```
    simOutput = dataOut(validOut);
```

Compare the Simulink Block Output with the MATLAB Function Output

Capture the output of the RS Encoder block. Compare that output with the output of the `comm.RSEncoder` System object.

```
fprintf('\nHDL RS Encoder\n');
difference = double(simOutput) - double(refOutput);
fprintf('\nTotal Number of samples differed between Simulink block output and MATLAB function ou
```

```
HDL RS Encoder

Total Number of samples differed between Simulink block output and MATLAB function output is: 0
```

See Also

Blocks
RS Encoder

HDL Implementation of AWGN Generator

This example shows the implementation of an additive white Gaussian noise (AWGN) generator that is optimized for HDL code generation and hardware implementation. The hardware implementation of AWGN accelerates the performance evaluation of wireless communication systems using an AWGN channel. In this example, the Simulink® model accepts signal-to-noise ratio (SNR) values as inputs and generates Gaussian random noise along with valid signal. The example supports SNR input ranges from –20 to 31 dB in steps of 0.1 dB.

Modern wireless communication systems includes many different simulation parameters, such as channel bandwidth, modulation type, and code rate. The performance evaluation of these systems with these simulation parameters is a bottleneck. Hardware capabilities of FPGAs can speed up simulations.

Model Architecture

```
% Run this command to open the HDLAWGNGenerator model.

modelname = 'HDLAWGNGenerator';
open_system(modelname);
```
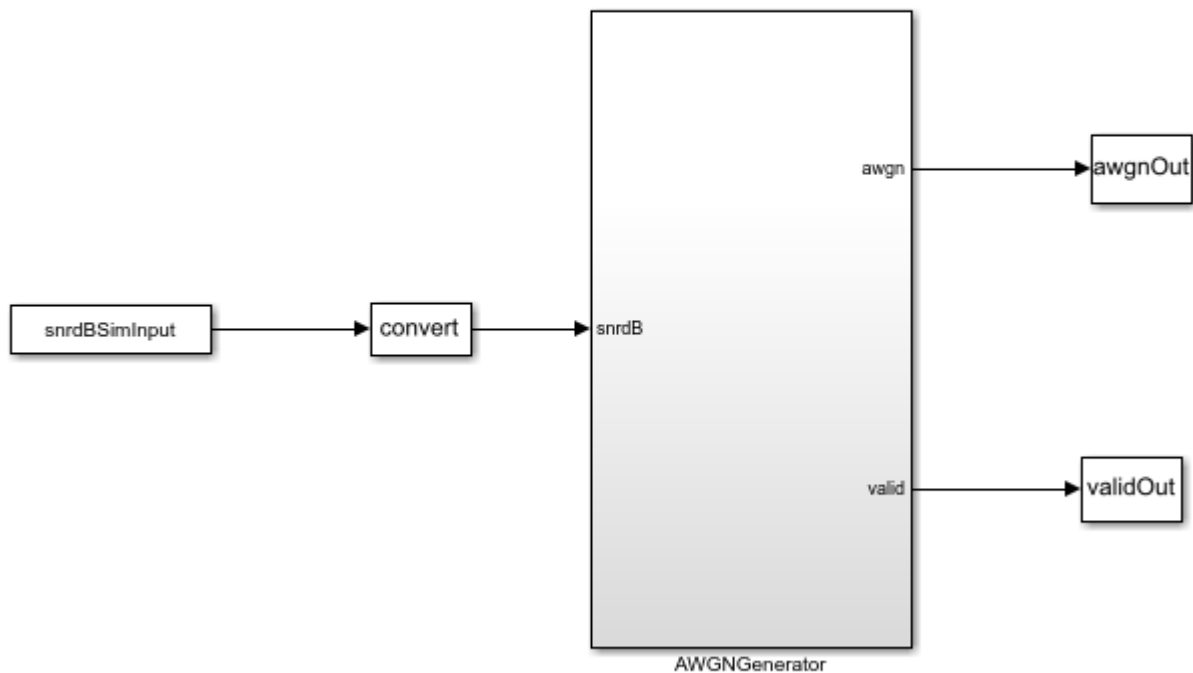

Copyright 2020 The MathWorks, Inc.

This example demonstrates the implementation of an AWGN generator based on the Box-Muller method. The Box-Muller method is widely adopted for Gaussian noise generation because of its hardware-friendly architecture and constant output rate. The top-level structure of the model includes these three subsystems.

- SNR dB to Linear Scale Converter
- Gaussian Noise Generator with Unit Variance
- Gaussian Noise Generator with Required Variance

```
% Run this command to open the subsystems inside AWGNGenerator model.

open_system([modelname '/AWGNGenerator']);
```
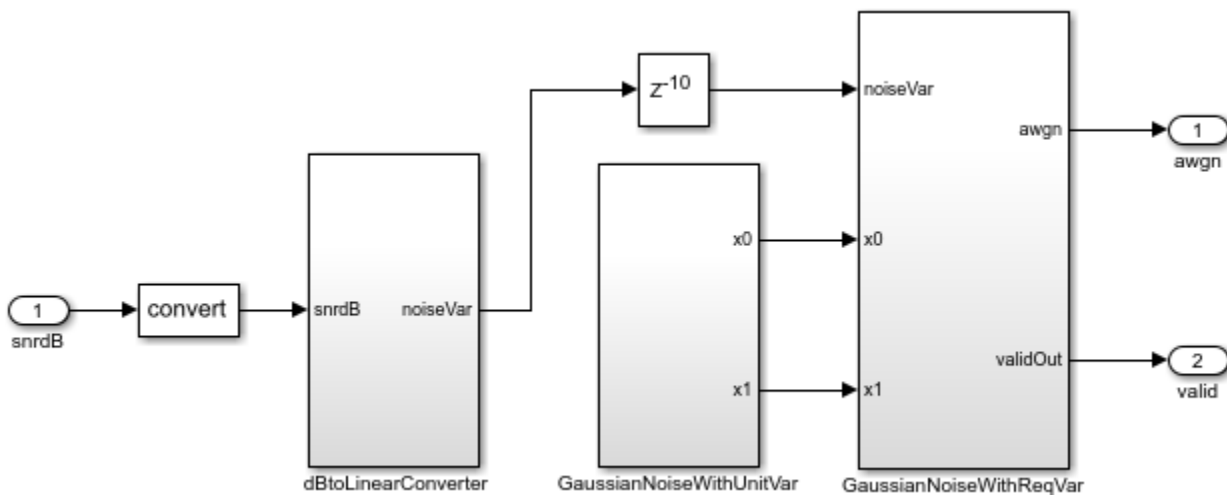

SNR dB to Linear Scale Converter

The dBtoLinearConverter subsystem takes an SNR value in dB as input and converts it into noise variance in a linear scale. This noise power is used to multiply the output of the Gaussian noise with unit variance. This lookup table approach is used for converting an SNR value in dB to a noise power value in a linear scale. During the conversion, the signal power is assumed to be 1. This subsystem has a latency of 1 clock cycle.

Gaussian Noise Generator with Unit Variance

The GaussianNoiseWithUnitVar subsystem generates Gaussian noise with unit variance by using the Box-Muller method. The Box-Muller method uses two uniformly distributed random variables to generate two normally distributed random variables through a series of logarithmic, square root, sine, and cosine operations as shown in this figure. Those two uniformly distributed random varibles are generated using the Tausworthe algorithm.
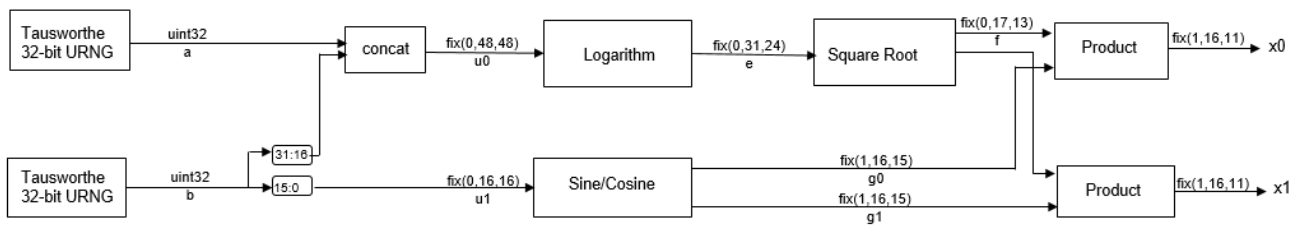
Implementation of HDL Tausworthe Uniform Random Number

The Tausworthe Uniform Random Number Generator module is used to generate two 32-bit uniform random integers. Each 32-bit uniform random number with improved statistical properties is obtained by combining three linear feedback shift register (LFSR) based uniform random number generators (URNGs). This implementation requires these two seeds: TausURNG1 and TausURNG2. The `whdlexamples.hdlawgnGen_init.m` script file initializes these seeds.

The ConcatandExtract subsystem accepts 32-bit uniform random integers, *a* and *b*, to generate two uniform random numbers, *u0* and *u1*, in the range [0, 1) with bit-widths 48 and 16, respectively. *u0* is generated by concatenating the 32-bit value of *a* and higher 16 bits of *b*. Uniform random number *u1* is generated by extracting the lower 16 bits of *b*.

```
open_system([modelname '/AWGNGenerator/GaussianNoiseWithUnitVar/TausUniformRandGen']);
close_system([modelname '/AWGNGenerator/GaussianNoiseWithUnitVar/TausUniformRandGen']);
open_system([modelname '/AWGNGenerator/GaussianNoiseWithUnitVar/TausUniformRandGen/TausURNG1']);
close_system([modelname '/AWGNGenerator/GaussianNoiseWithUnitVar/TausUniformRandGen/TausURNG1'])
```
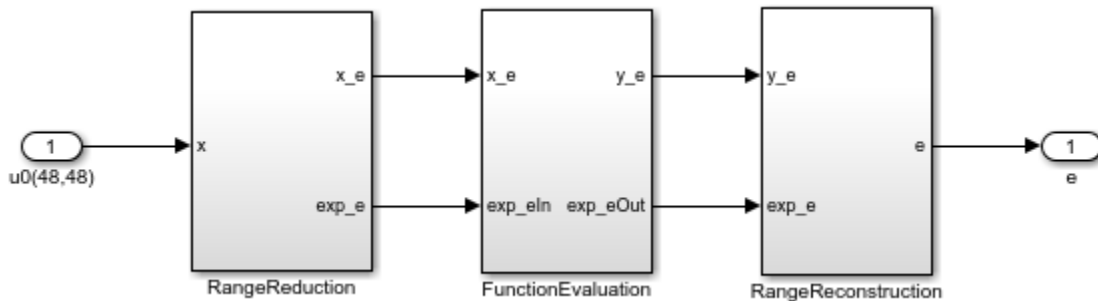
Implementation of HDL Logarithm

HDL logarithm subsystem evaluates the approximate logarithm based on the piecewise linear polynomial method. This module has latency of 3 clock cycles. Implementation of the HDL logarithm involves these three steps.

1 Range Reduction – In this step, the original range of the input, which is [0, 1-2^(–48)) is reduced to a more convenient smaller range of [1, 2). The log function is approximated on the reduced range in the next step.

2 Function Evaluation – The log function is approximated over 256 equally spaced segments in the range [1, 2) by using a second-degree polynomial. Coefficients of the second-degree polynomial are obtained using the `polyfit` function. These coefficients are stored in a lookup table, which is indexed using the first 8 bits of input to the function evaluation block.

3 Range Reconstruction – The result of the function evaluation is expanded back to the original range. A bit left shift operation is used for range reconstruction and to implement the $-2*log$ function.

Run this command to open HDL logarithm subsystem.

```
open_system([modelname '/AWGNGenerator/GaussianNoiseWithUnitVar/logImplementation/log']);
```
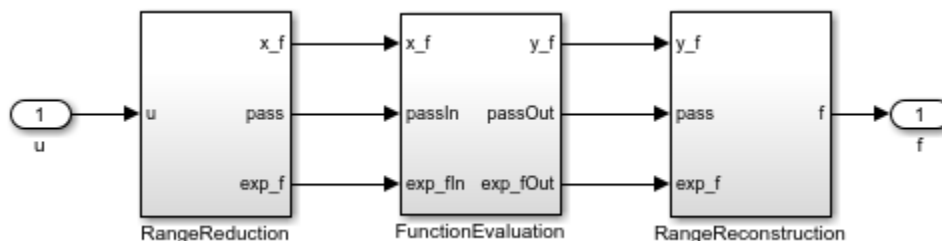
Implementation of HDL Square Root

The HDL Square root subsystem evaluates approximate square root based on the piecewise linear polynomial method. This module has a latency of 2. The implementation of the HDL square root involves these three steps.

1. Range Reduction – The input data type to the module is `fi(0, 31, 24)`. This range is reduced to a smaller range of [1, 4). The square root function is approximated on the reduced range in the next step.

2. Function Evaluation – The square root function is approximated over 64 equally spaced segments in the range [1, 2) and [2, 4) by using a first-degree polynomial. Coefficients of the first-degree polynomial are stored in a lookup table, which is indexed using the first 6 bits of input to the function evaluation block.

3. Range Reconstruction – The result of the function evaluation is expanded back to the original range using a left shift operation.

```
close_system([modelname '/AWGNGenerator/GaussianNoiseWithUnitVar/logImplementation/log']);
open_system([modelname '/AWGNGenerator/GaussianNoiseWithUnitVar/SqrtImplementation/SqrtEval']);
```


Implementation of HDL Sine and Cosine

The HDL optimized implementation of a sine or cosine function uses a lookup table approach. **Sin** and **Cos** are implemented using the existing Sine HDL Optimized and Cosine HDL Optimized (HDL Coder) blocks in the HDL Coder / Lookup Tables library.

```
close_system([modelname '/AWGNGenerator/GaussianNoiseWithUnitVar/SqrtImplementation/SqrtEval']);
```

Gaussian Noise Generator with Required Variance

The GaussianNoiseWithReqVar subsystem converts Gaussian noise with unit variance to Gaussian noise with required variance. This subsystem takes inputs from dBToLinearConvertor and GaussianNoiseWithUnitVar subsystems. The linear noise variance obtained from

dBToLinearConvertor is multiplied with normally distributed random variables obtained from GaussianNoiseWithUnitVar.

Results and Plots

The `whdlexamples.hdlawgnGen_init.m` script file is used to specify the SNR range, generate the required number of noise samples, initialize the seeds for TausURNG1 and TausURNG2 subsystem and to generate coefficients for the function evaluation of the HDL log and square root.

The `whdlexamples.hdlawgnGen_init.m` script file is the initialization function of HDLAWGNGenerator model. This function generates the input data and initializes the seeds for tausURNG and coefficients for the function evaluation. Simulate `HDLAWGNGenerator.slx` to generate 10^6 valid AWGN samples for each SNR of 5 dB and 15 dB. The implementation is pipelined to maximize the synthesis frequency, generating AWGN with an initial latency of 11. Plot the probability density function (PDF) of the AWGN output.

```matlab
latency = 11;
NumOfSamples = 10^6;

% Simulate the model
open_system('HDLAWGNGenerator');
set_param(gcs,'SimulationMode','Accel');
fprintf('\n Simulating HDL AWGN Generator...\n');
outSimulink = sim('HDLAWGNGenerator','ReturnWorkspaceOutputs','on');
fprintf('\n Simulation complete.\n');
awgnSimulink = outSimulink.awgnOut;

% Plot PDF
figure;
title('PDF for Real Part of AWGN');
hold on
histogram(real(awgnSimulink(latency+1:NumOfSamples+latency)),500, ...
        'Normalization','pdf','BinLimits',[-2 2],'FaceColor','blue', ...
        'EdgeColor','none');
histogram(real(awgnSimulink(NumOfSamples+latency+1:end)),500,...
        'Normalization','pdf','BinLimits',[-2 2],'FaceColor','yellow', ...
        'EdgeColor','none');
legend('5 dB SNR','15 dB SNR');

figure;
title('PDF for Imaginary Part of AWGN');
hold on
histogram(imag(awgnSimulink(latency+1:NumOfSamples+latency)),500, ...
        'Normalization','pdf','BinLimits',[-2 2],'FaceColor','blue', ...
        'EdgeColor','none');
histogram(imag(awgnSimulink(NumOfSamples+latency+1:end)),500, ...
        'Normalization','pdf','BinLimits',[-2 2],'FaceColor','yellow', ...
        'EdgeColor','none');
legend('5 dB SNR','15 dB SNR');
```
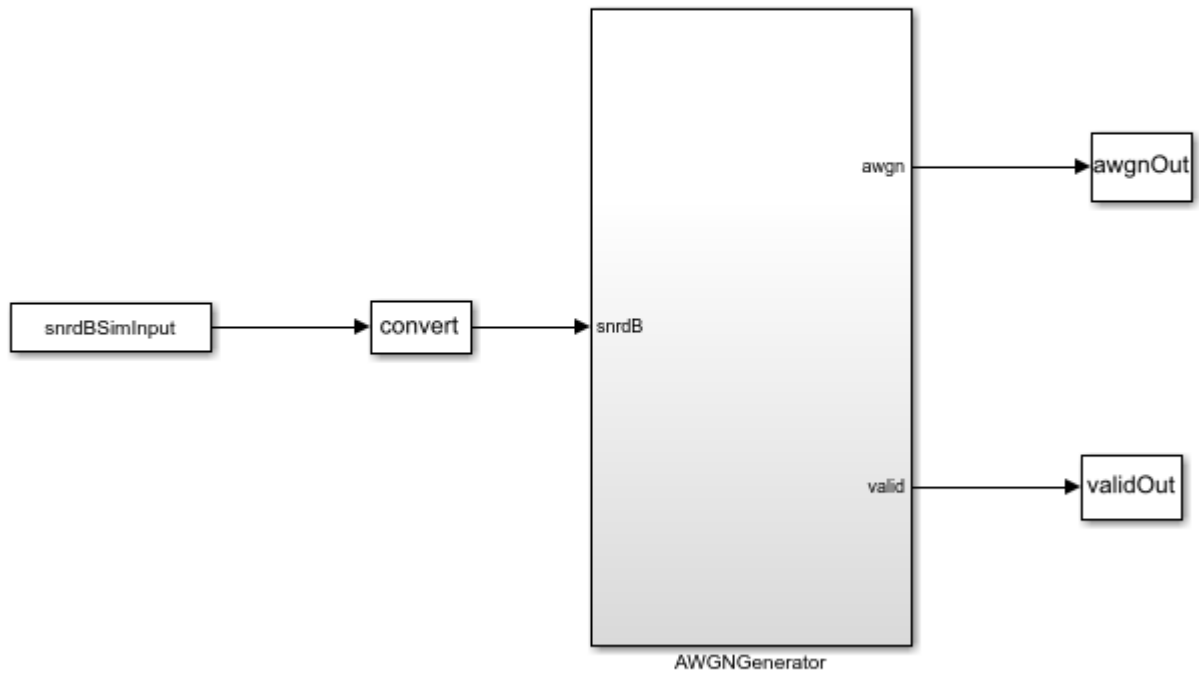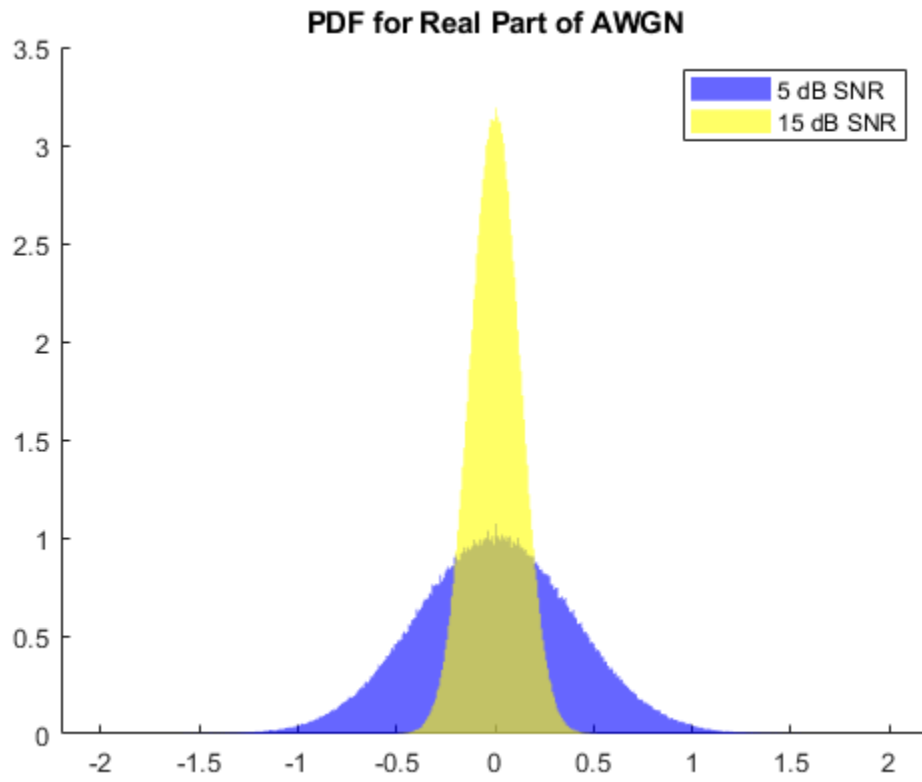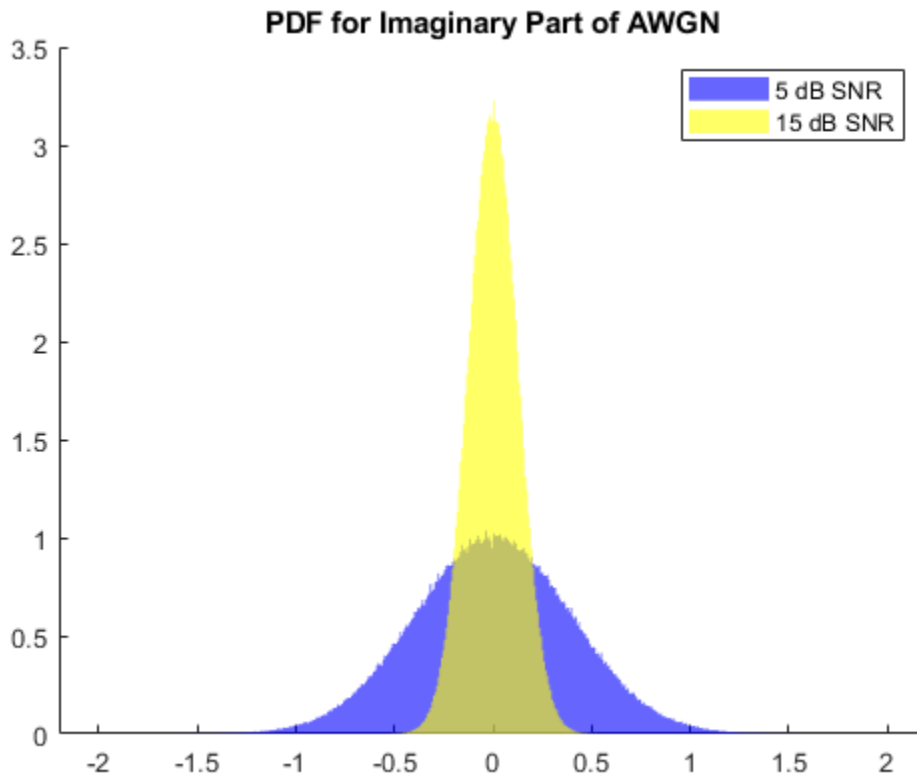
```
 Simulating HDL AWGN Generator...

 Simulation complete.
```

HDL AWGN Generator

Copyright 2020 The MathWorks, Inc.

Verification

Compare the output of the AWGN Simulink model with the output of the HDL equivalent AWGN MATLAB® function.

```
NumOfSamples = 1000;
% MATLAB output
fprintf('\n Simulating MATLAB HDL AWGN Generator for comparison...\n');
awgnMatlab=whdlexamples.hdlawgn(snrdBSimInput(1:NumOfSamples),seedsURNG1,seedsURNG2);
fprintf('\n Simulation complete. \n')

% Compare MATLAB and Simulink outputs
figure;
ax=axes('FontSize', 20);
plot(1:1000,real([awgnSimulink(latency+1:NumOfSamples+latency) awgnMatlab]));
xlabel(ax,'Number of Samples');
ylabel(ax,'Real Part of AWGN');
title(ax,'Comparison of MATLAB and Simulink Output (Real Part)');
legend('Simulink output','MATLAB output');

figure;
ax=axes('FontSize', 20);
plot(1:1000,imag([awgnSimulink(latency+1:NumOfSamples+latency) awgnMatlab]));
xlabel(ax,'Number of Samples');
ylabel(ax,'Imaginary Part of AWGN');
title(ax,'Comparison of MATLAB and Simulink Output (Imaginary Part)');
legend('Simulink output','MATLAB output');
```
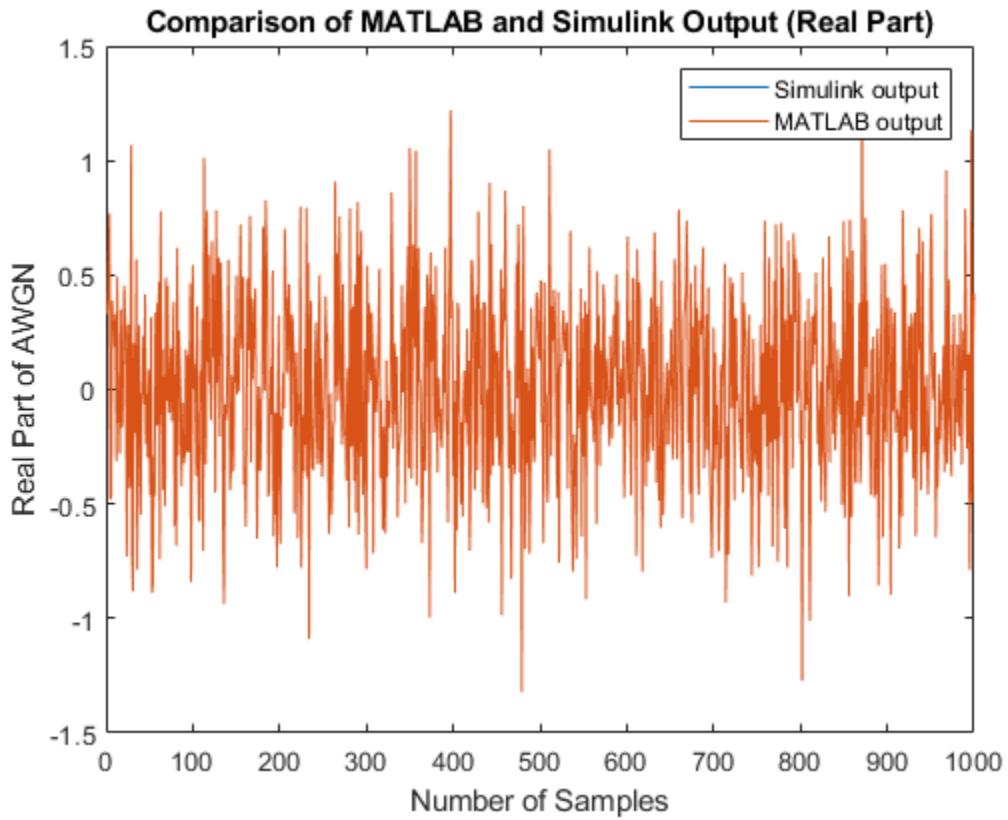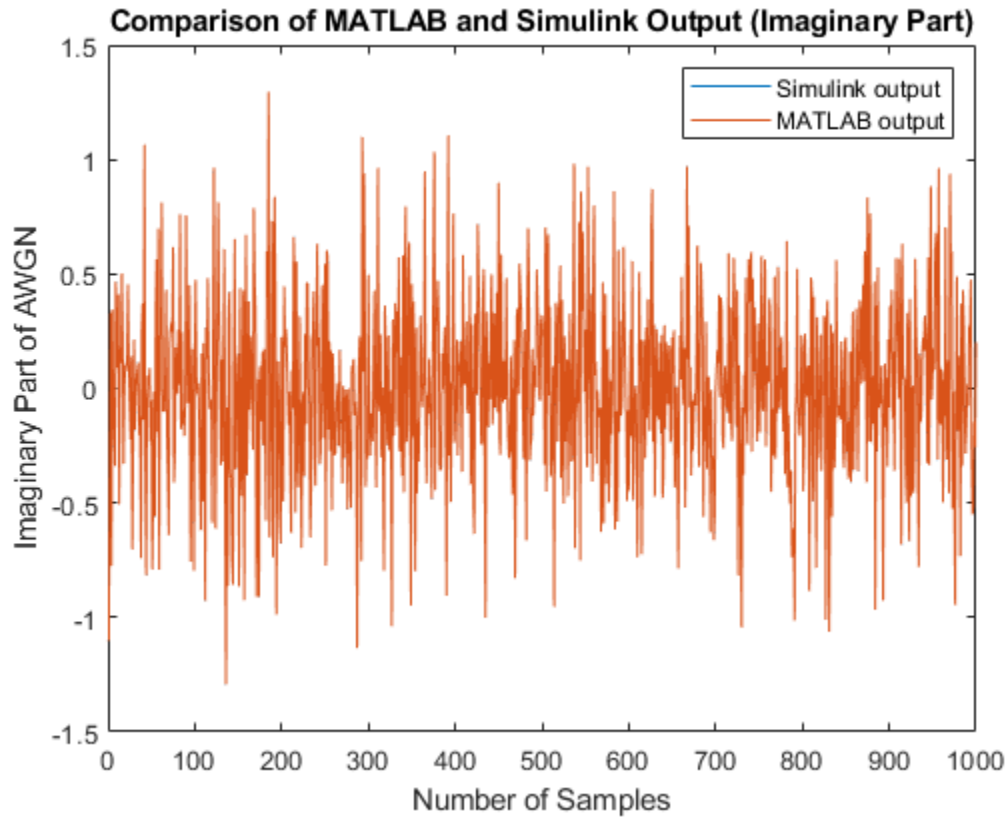
Simulating MATLAB HDL AWGN Generator for comparison...

Simulation complete.

Comparison of MATLAB and Simulink Output (Imaginary Part)

HDL Code Generation

To check and generate the HDL code referenced in this example, you must have an HDL Coder™ license.

To generate the HDL code, enter this command at the MATLAB command prompt.

```
makehdl('HDLAWGNGenerator/AWGNGenerator')
```

To generate a test bench, enter this command at the MATLAB command prompt.

```
makehdltb('HDLAWGNGenerator/AWGNGenerator')
```

In this example, HDL code generated for the AWGNGenerator module is implemented for the Xilinx® Zynq®-7000 ZC706 board. The implementation results are shown in this table.

Hardware Type	Usage
Slice LUT	6171
Slice Registers	1668
RAMB18E1	9
DSP48E1	16
Max Freq (MHz)	250

References

1. J.D. Lee, J.D. Villasenor, W. Luk, and P.H.W. Leong. "A Hardware Gaussian Noise Generator Using the Box-Muller Method and Its Error Analysis," 659–71. IEEE, 2006. https://doi.org/10.1109/TC.2006.81.

# HDL Implementation of Digital Predistorter

This example shows the implementation of a digital predistorter (DPD) model that is optimized for HDL code generation and hardware implementation. The predistortion mechanism is executed in two stages. In the first stage, a set of DPD coefficients are estimated based on the input and output data of the power amplifier (PA). In the second stage, the input data of the PA is predistorted based on the estimated DPD coefficients and provided as new input to the PA. This example demonstrates a system-level simulation in which the Digital Predistorter subsystem generates HDL code, while the DPD coefficient estimation generates C/C++ code. This example model supports only Normal and Accelerator simulation modes.

Digital Predistortion

Digital predistortion is a baseband signal processing technique that is used for correcting impairments in radio frequency (RF) power amplifiers. These impairments cause out-of-band emissions or spectral regrowth and in-band distortion, which results in an increased bit error rate (BER) and a decreased throughput of the system. Power amplifiers cause unwanted effects in the system due to their nonlinear behavior. Communication systems using orthogonal frequency division multiplexing (OFDM), such as a wireless local area network (WLAN), worldwide interoperability for microwave access (WiMax), long term evolution (LTE), and 5G new radio (NR), are vulnerable to these unwanted effects. A precorrection is applied on the signal so that the cascade of the DPD and PA is close to an ideal, linear, and memoryless system. This linearization can improve PA power efficiency and can be more spectrum efficient. This figure shows the top-level structure of the example.

Run this command to open the example.

```
modelname = 'DPDHDLExample';
open_system(modelname);
```
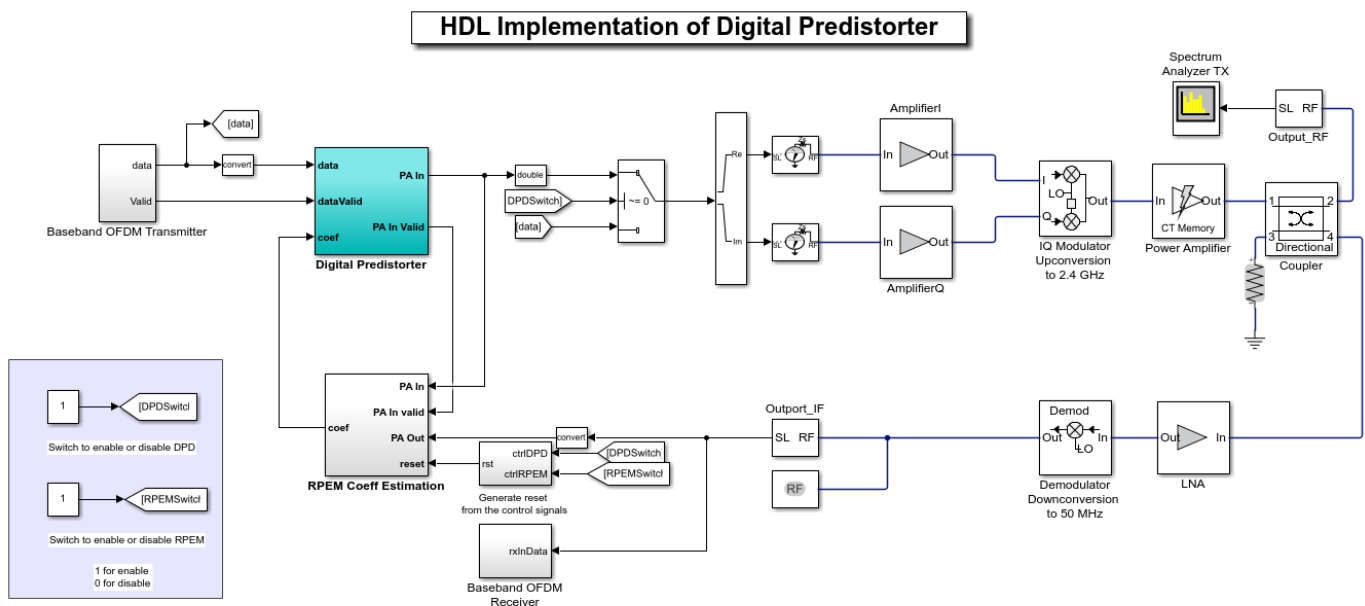
Baseband OFDM Transmitter

The Baseband OFDM Transmitter subsystem generates a baseband signal and provides that signal as input data to the Digital Predistorter subsystem. The `OFDMTx` function in this subsystem generates an OFDM transmitter waveform with synchronization, reference, header, pilots, and data signals and returns `txWaveform`, `txGrid`, and `diagnostics` using the transmitter parameter set **txParam**. For more information about the `OFDMTx` function, see the "HDL OFDM MATLAB References" on page 5-169 example. You can also replace the Baseband OFDM Transmitter subsystem with any custom transmitter to provide data to the Digital Predistorter subsystem. This figure shows the baseband input signal generation for this example. Run this command to open the Baseband OFDM Transmitter subsystem.

```
load_system(modelname);
open_system([modelname '/Baseband OFDM Transmitter']);
```
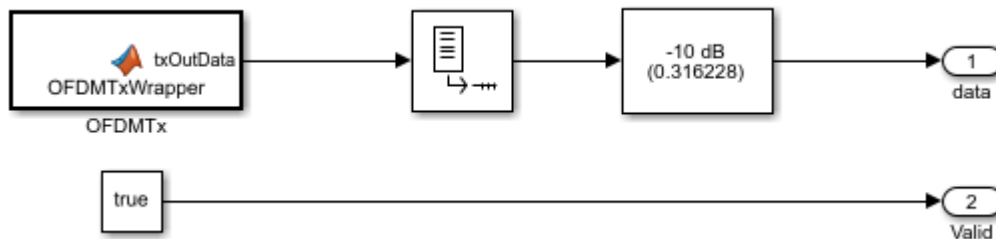

Coefficients Estimation

The RPEM Coeff Estimation subsystem estimates a set of coefficients by collecting data from the input and output of the PA. These coefficients are used to distort the signal before the power amplifier. PA characteristics vary over time and operating conditions, so an adaptive recursive prediction error method (RPEM) algorithm is used to estimate the DPD coefficients. The number of coefficients to be estimated depends on the memory depth and polynomial degree of the PA. In this example, because the total number of coefficients that need to be estimated is 25, the memory depth and polynomial degree of the PA are set to 5. For more information about the RPEM, see [1]. To generate C/C++ code for RPEM Coeff Estimation subsystem, use the `slbuild` command. Run this command to open the RPEM Coeff Estimation subsystem.

```
load_system(modelname);
open_system([modelname '/RPEM Coeff Estimation']);
```
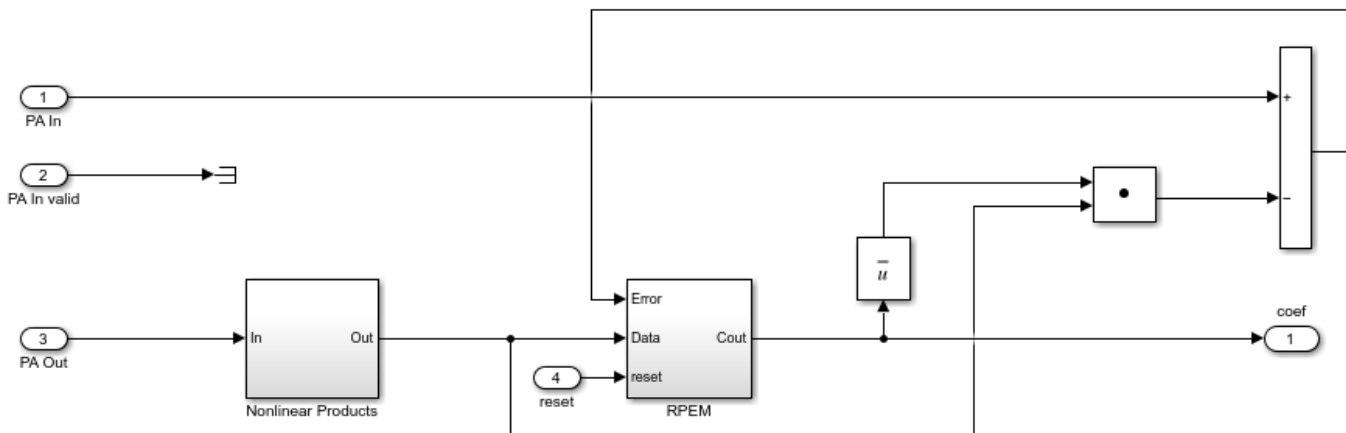
Digital Predistorter

The Digital Predistorter subsystem distorts the input data using the coefficients estimated by the RPEM Coeff Estimation subsystem. The DPD design in this example is based on a memory polynomial, which corrects the nonlinearities and memory effects in the PA. The estimated coefficients and the generated input data are provided as input to the DPD for applying predistortion. The input data is first placed in a shift register based on the memory depth. Second, this vector is concatenated with the nonlinear products of the data depending on the polynomial degree. This concatenation forms a vector of 25 that means memory depth times degree elements. The dot product of the obtained vector and estimated coefficients provides the predistorted input that is fed as input to PA after upsampling. Run this command to open the Digital Predistorter subsystem.

```
load_system(modelname);
open_system([modelname '/Digital Predistorter']);
```
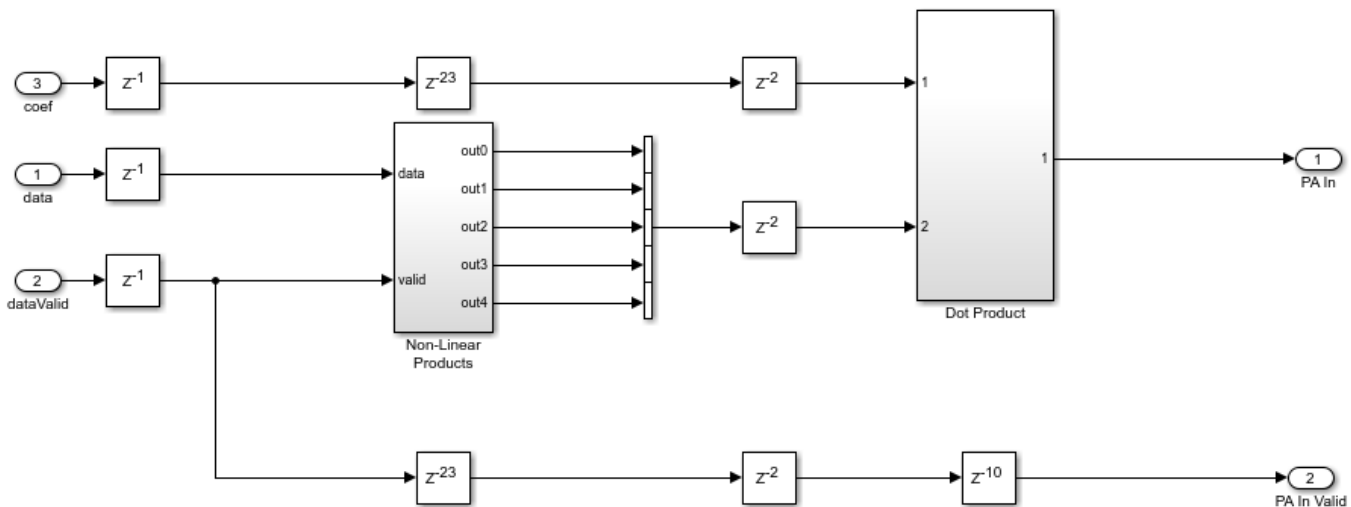

RF Blocks Configuration

This example has a control switch to enable or disable predistortion and coefficient estimation. If you enable the switch, the example provides the output data from the Digital Predistorter subsystem as input to RF blocks. Otherwise, the example provides the output data from the Baseband OFDM Transmitter subsystem as input to RF blocks as in-phase (I), quadrature-phase (Q) samples. These I/Q samples are upsampled to 2.4 GHz and provided as input to the PA. The coefficient matrix required by the PA is preloaded based on the standard-compliant LTE signal with a sample rate of 15.36 MHz. These coefficients are stored in a MAT file, and the values are loaded while initializing the example. In the other path, the data is passed through a low noise amplifier (LNA) and is down-converted before providing it to the RPEM Coeff Estimation subsystem.

Baseband OFDM Receiver

The Baseband OFDM Receiver subsystem collects the down-converted data and provides it as an input to the OFDMRx function. This function performs carrier frequency offset estimation and correction, frame synchronization, OFDM demodulation, channel estimation, channel equalization, phase offset correction, and decodes the transmitted bits. For more information about the OFDMRx function, see the "HDL OFDM MATLAB References" on page 5-169 example.

Verification and Results

Run the model. By default, the Digital Predistorter and RPEM Coeff Estimation are enabled. If you disable the DPD, the error vector magnitude (EVM) increases, and the spectral regrowth in adjacent channels increases. The constellation and spectrum analyzer diagrams show the results of running the model with the DPD enabled.

```
sim(modelname);
```

```
 Estimating carrier frequency offset ...

 First four frames are used for carrier frequency offset estimation.

 Estimated carrier frequency offset is 3.252304e+00 Hz.

 Detected and processing frame 5
-----------------------------------------

 Header CRC passed

 Modulation: 16QAM, codeRate=1/2 and FFT Length=128

 Data CRC passed

 Data decoding completed
-----------------------------------------

 Detected and processing frame 6
-----------------------------------------

 Header CRC passed

 Modulation: 16QAM, codeRate=1/2 and FFT Length=128

 Data CRC passed

 Data decoding completed
-----------------------------------------
```
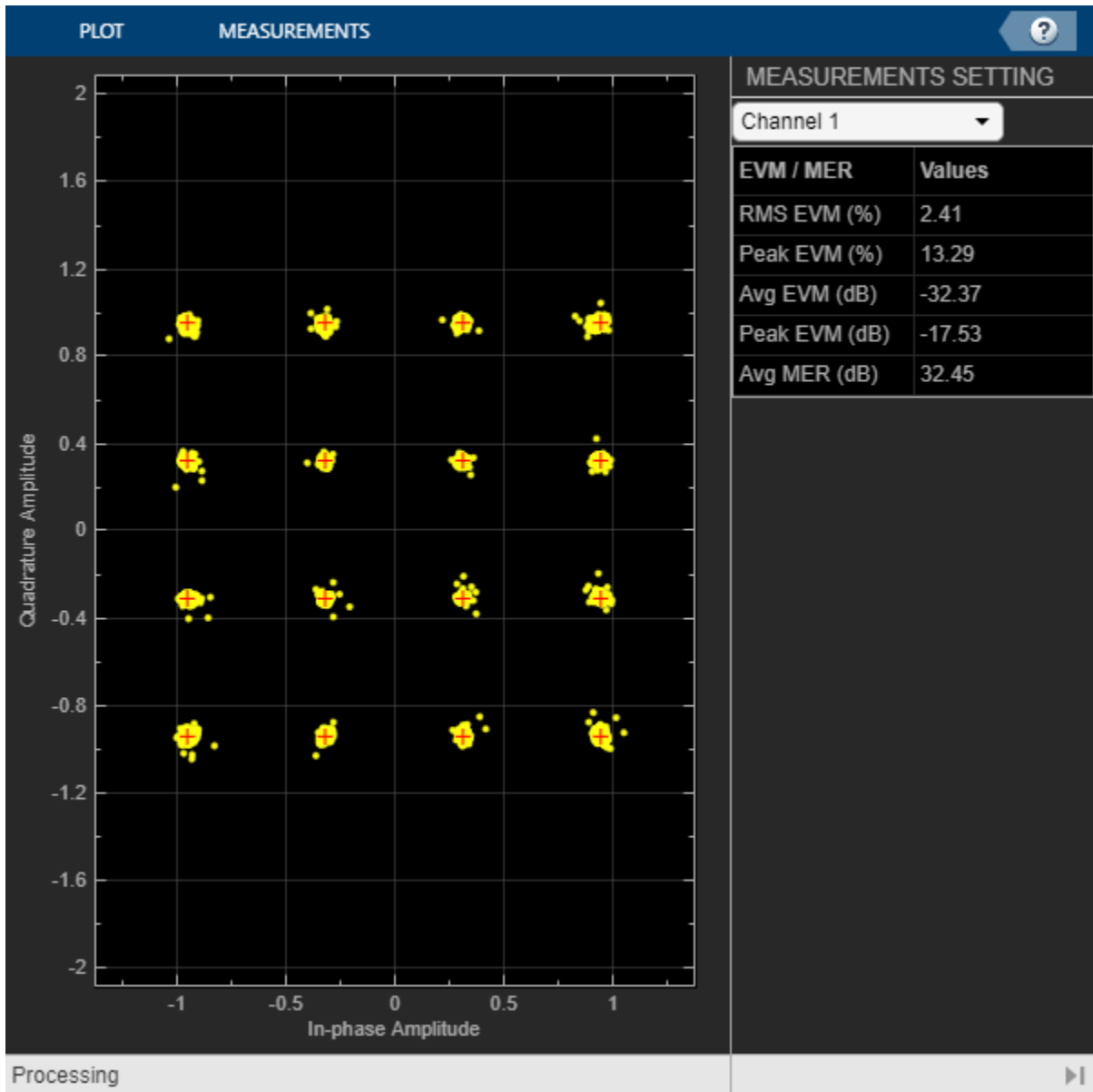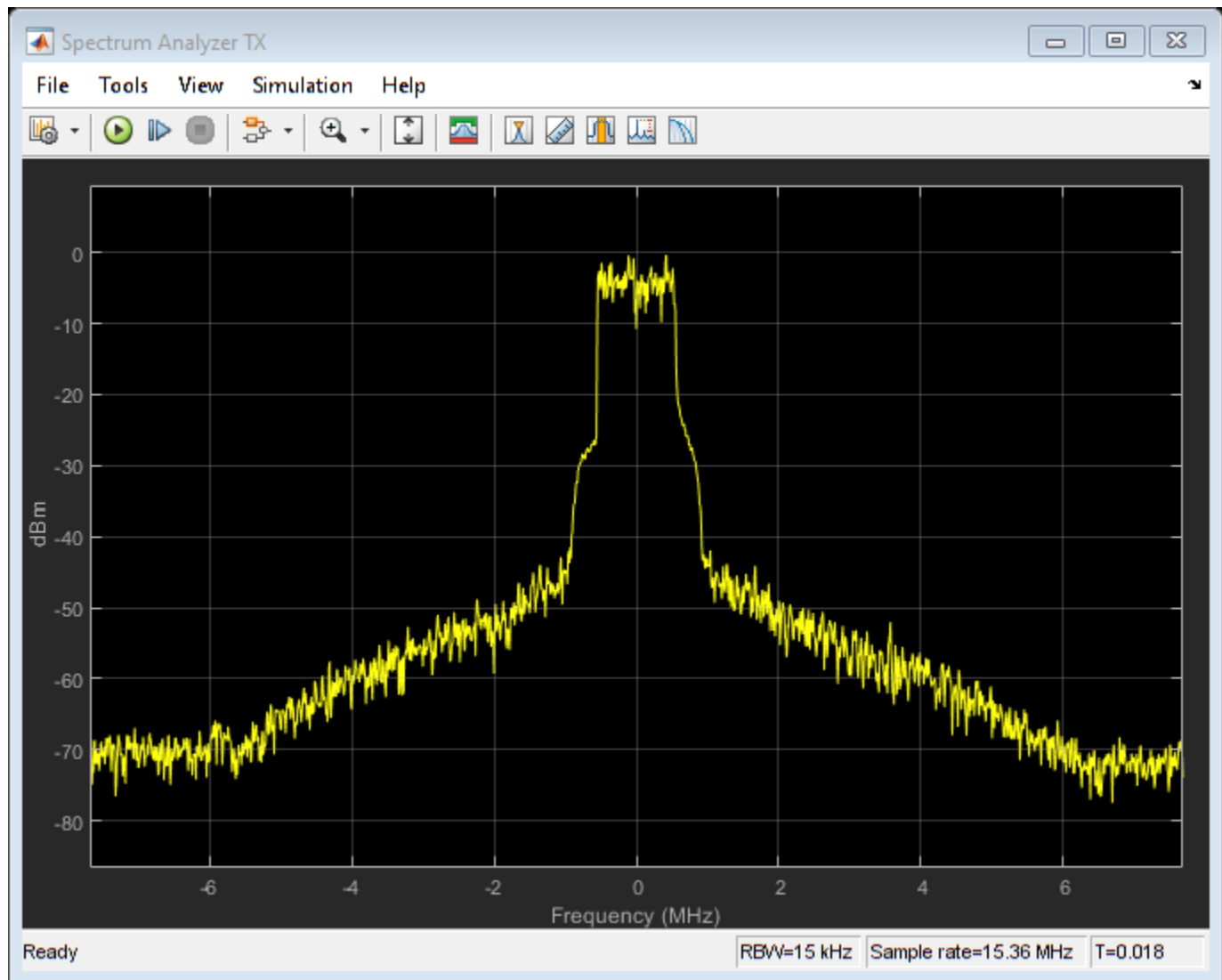
HDL Code Generation and Implementation Results

To check and generate HDL for this example, you must have HDL Coder™. Use the `makehdl` and `makehdltb` commands to generate the HDL code and test bench for the **Digital Predistorter** subsystem.

The Digital Predistorter subsystem is synthesized on a Xilinx® Zynq®-7000 ZC706 evaluation board. The frequency obtained after place and route is about 220 MHz. Create a table that displays the post place and route resource utilization results for a 16-bit complex input.

```
F = table(...
    categorical({'Slice LUT'; 'Slice Registers';'DSP'}), ...
    categorical({'6028'; '8115'; '160'}), ...
    categorical({'218600'; '437200'; '900'}), ...
    categorical({'2.75'; '1.85'; '17.78'}), ...
    'VariableNames', ...
    {'Resources','Utilized','Available','Utilization (%)'});
disp(F);
```

```
     Resources        Utilized     Available    Utilization (%)
  _____    _____     _____    _____

  Slice LUT             6028        218600          2.75
  Slice Registers       8115        437200          1.85
  DSP                    160           900         17.78
```

References

1. Gan, Li, and Emad Abd-Elrady. "Digital Predistortion of Memory Polynomial Systems Using Direct and Indirect Learning Architectures." In *Proceedings of the Eleventh IASTED International Conference on Signal and Image Processing (SIP)* (F. Cruz-Roldan and N. B. Smith, eds.), No. 654-802. Calgary, AB: ACTA Press, 2009.

See Also

Related Examples

- "HDL OFDM MATLAB References" on page 5-169
- "Digital Predistortion to Compensate for Power Amplifier Nonlinearities"

Encode Streaming Data Using General CRC Generator HDL Optimized Block for 5G NR Standard

This example shows how to use the General CRC Generator HDL Optimized block for encoding streaming data according to the 5G NR standard.

In this example, the output of this block is compared with the function `nrCRCEncode` (5G Toolbox). A cyclic redundancy check (CRC) is an error-detection code designed to detect errors in streaming data. A CRC generator calculates a short fixed-length binary sequence checksum and appends it with the data. A CRC detector performs a CRC on the data and compares the resulting checksum with the appended checksum. If the two checksums do not match, an error is detected. The CRC generator and detector are used in the 5G NR system to detect any errors in the transport blocks of control and uplink and downlink data channels. The 5G NR standard specifies six different cyclic generator polynomials: CRC6, CRC11, CRC16, CRC24A, CRC24B, and CRC24C. For more information about these polynomials, see TS 38.212 Section 5.1 [1].

Generate Input Data for NR CRC Generator

Select a CRC polynomial specified in the 5G NR standard. Generate random input data of length `frameLen` and control signals that indicate the frame boundaries. The example model imports the MATLAB® workspace variables `dataIn`, `startIn`, `endIn`, `validIn`, `sampleTime`, and `simTime`.

```
CRCType = 'CRC24A'; % Specify the CRCType as 'CRC6','CRC11','CRC16','CRC24A','CRC24B' or 'CRC24C
frameLen = 100;
msg = randi([0 1],frameLen,1);

[dataIn,ctrlIn] = whdlFramesToSamples(msg);

dataIn = timeseries(logical(dataIn'));
startIn = timeseries(logical(ctrlIn(:,1)));
endIn = timeseries(logical(ctrlIn(:,2)));
validIn = timeseries(logical(ctrlIn(:,3)));

sampleTime = 1;
simTime = length(ctrlIn(:,3)) + 100;
```

Run NR CRC Generator Model

The `nrCRCGeneratorExampleInit.m` script configures the General CRC Generator HDL Optimized block by setting the parameters of the block based on the specified CRC generator polynomial, `CRCType`. This script also provides input to the reference function `nrCRCEncode` (5G Toolbox). The NR CRC Generator subsystem contains the General CRC Generator HDL Optimized block. Running the model imports the input signal variables from the workspace and returns the CRC-encoded output and control signals that indicate the frame boundaries. The model exports variables `encOut` and `ctrlOut` to the MATLAB® workspace.

```
[poly,crcPolynomial,initState,finalXORValue] = nrCRCGeneratorExampleInit(CRCType);
open_system('NRCRCGeneratorHDL');
encOut = sim('NRCRCGeneratorHDL');
```
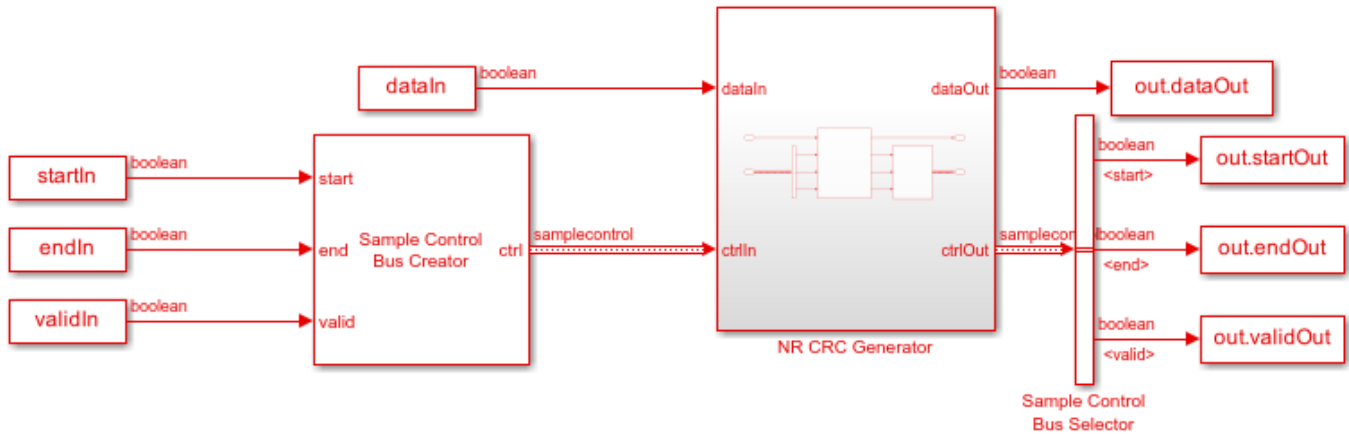
Copyright 2020 The MathWorks, Inc.

Verify NR CRC Generator Results

Convert the streaming data output of the NR CRC Generator subsystem to frames. Compare those frames with the output of the `nrCRCEncode` function.

```
startIdx = find(encOut.startOut);
endIdx = find(encOut.endOut);
dataOut = encOut.dataOut;

dataRef = nrCRCEncode(msg,poly);
bitErr = sum(abs(dataRef - dataOut(startIdx:endIdx)));
fprintf('CRC-encoded frame: Behavioral and HDL simulation differ by %d bits\n',bitErr);

close_system('NRCRCGeneratorHDL');
```

```
CRC-encoded frame: Behavioral and HDL simulation differ by 0 bits
```

References

1 3GPP TS 38.212. *NR ; Multiplexing and Channel Coding. 3rd Generation Partnership Project; Technical Specification Group Radio Access Network.*

See Also

Blocks
General CRC Generator HDL Optimized

Functions
nrCRCEncode

DVB-S2 HDL LDPC Encoder

This example shows how to implement DVB-S2 LDPC encoding using Simulink® blocks that are optimized for HDL code generation.

The DVB-S2 LDPC Encoder block in this example works in conjunction with the DVB-S2 LDPC Decoder block. The output results of this example are compared with those of the `ldpcEncode` helper function in Satellite Communications Toolbox.

Introduction

Digital Video Broadcast Satellite Second Generation (DVB-S2) is a European Telecommunications Standards Institute (ETSI) standard of the second generation for digital data transmission through satellites [1]. The DVB-S2 standard is designed for broadcast services, interactive services, digital satellite news gathering, and professional services. In 2005, DVB-S2 became the first standard to adopt low-density parity-check (LDPC) codes. DVB-S2 offers a powerful forward error correction (FEC) based on LDPC codes concatenated with Bose Chaudhuri Hocquenghem (BCH) codes. This mechanism allows quasi error-free operation at about 0.7 dB to 1 dB from the Shannon limit [1], which yields better decoding performance.

FEC performs outer coding with BCH codes and inner coding with LDPC codes. It accepts BBFRAMEs as inputs and outputs FECFRAMEs. A BBFRAME consists of a BBHEADER followed by a DATA FIELD. FEC coding processes each BBFRAME of k_{bch} bits to generate a FECFRAME of n_{ldpc} bits as shown in section 5.3 [1]. The following figure shows the frame format of FECFRAME data.
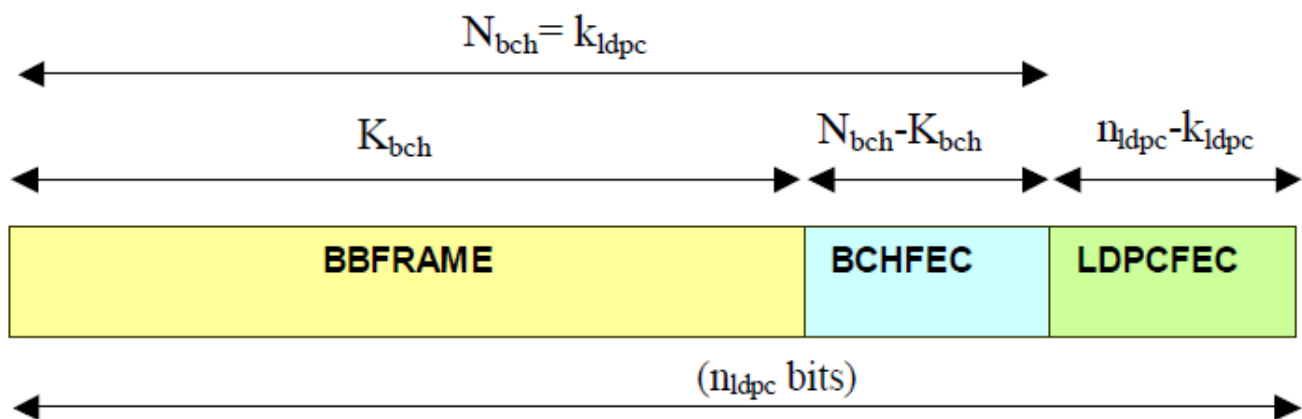
The LDPC codes in the DVB-S2 standard have two block lengths. Normal frames have a block length equal to 64,800 and short frames have block length equal to 16,200. The standard specifies 11 code rates for normal frame and 10 code rates for short frame. LDPC code parameters for coded (n_{ldpc}) and uncoded (k_{ldpc}) block lengths for different frames are defined in table 5a in section 5.3 of ETSI EN 302 307 [1].

Model Architecture

This figure shows the high-level architecture block diagram of the implementation of the `DVB-S2 LDPC Encoder` block.
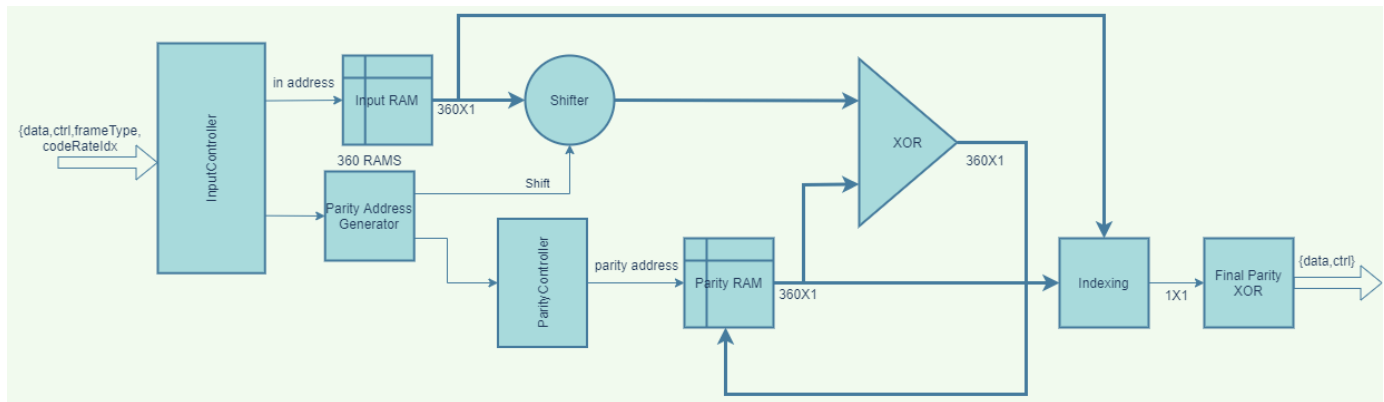
This figure shows the top-level structure of the `dvbs2hdlLDPCEncoder` model. You can generate the HDL code for the `DVB-S2 LDPC Encoder` subsystem in the model.

```
modelName = 'dvbs2hdlLDPCEncoder';
open_system(modelName);
set_param(modelName,'Open','on');
```
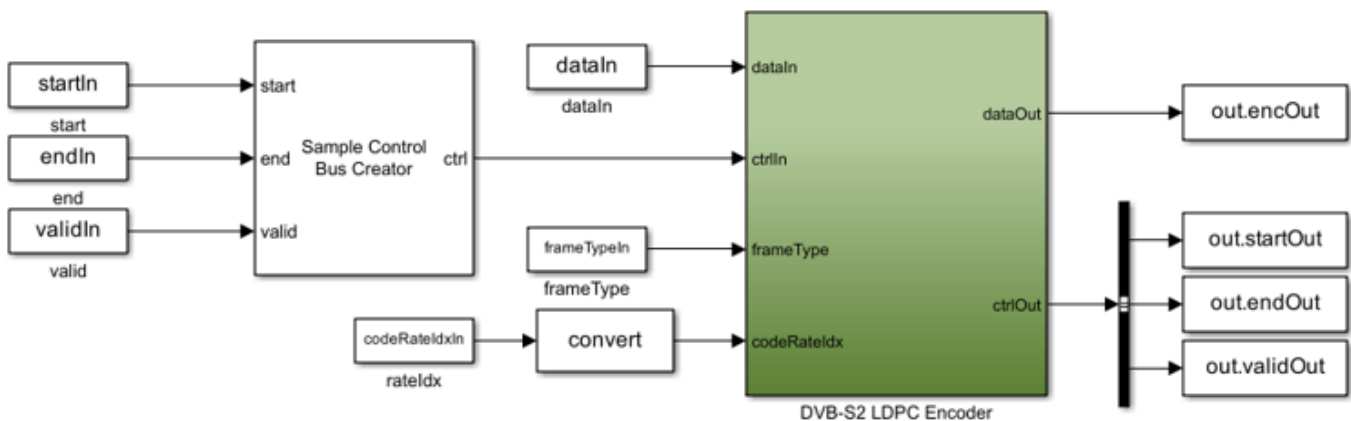

Copyright 2021-2022 The MathWorks, Inc.

DVB-S2 LDPC Encoder

The `DVB-S2 LDPC Encoder` subsystem accepts input data, the control signal, the frame type, and the code rate index. The *frameType* and *codeRateIdx* signals are sampled at the start of a frame. The `inputController` function controls the reading and writing of input data in the `Input RAM` subsystem and enables the encoding after writing the whole frame into the RAM. Using the parity bit addresses specified in standard Annex B, C [1], the shift values for the `Circular Shifter` subsystem and address for the `Parity RAM` subsystem are calculated and stored in the shiftLUT and ramLUT blocks respectively. `Parity Address Generator` subsystem generates the corresponding shift value and the address of the `Parity RAM` subsystem based on the input configuration of the FEC frame and code rate. `ParityController` function controls the parity calculation and reading of parity data. `Circular Shifter` subsystem shifts the data circularly, and an exclusive-OR operation is applied to the shifted data with the output of `Parity RAM` subsystem and stores the data in the same address. `Indexing` function multiplexes the input and parity bits and outputs the bits serially.

The final parity bits are calculated by applying an exclusive-OR operation to the current parity bits with the previous parity bits.

```
set_param(modelName,'Open','off');
open_system([modelName '/DVB-S2 LDPC Encoder']);
```
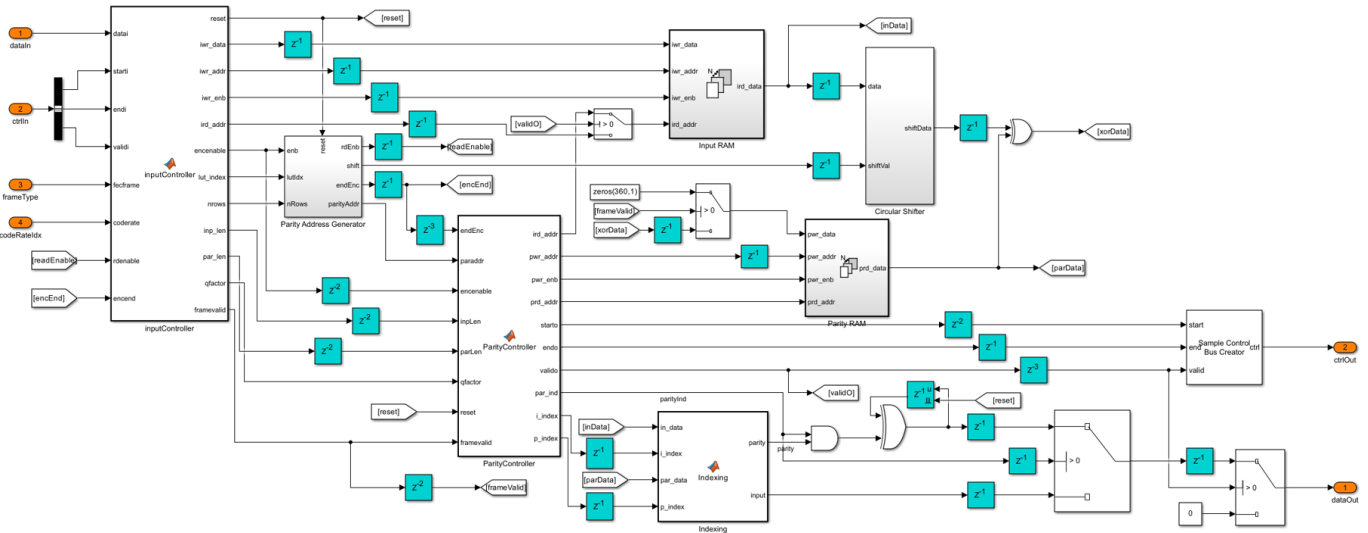

Parity Address Generator

The `inputController` function generates the *encenable* signal counts the number of columns and rows of parity bit addresses. The ramLUT block stores the addresses of the `Parity` RAM subsystem. The shiftLUT block stores the shift values of input data calculated using the following equations from the parity bit addresses specified in standard Annex B, C [1].

$$RAM\,Address = remainder(Parity bit addresses, qFactor)$$

$$shift = (Parity length - (Parity bit addresses - RAM\,Address))/qFactor$$

```
set_param([modelName '/DVB-S2 LDPC Encoder'],'Open','off');
open_system([modelName '/DVB-S2 LDPC Encoder/Parity Address Generator']);
```
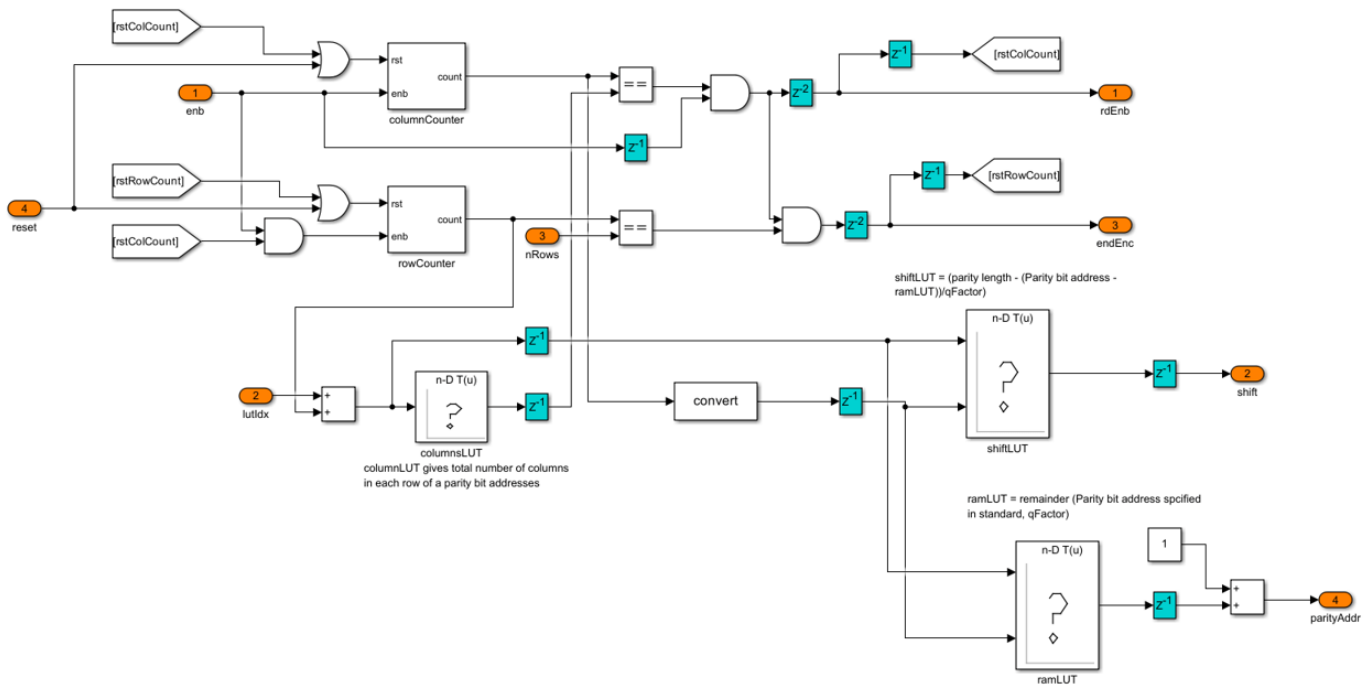
Circular Shifter

`Circular Shifter` subsystem shifts the data based on the shift value. The circular shift network is implemented with a fixed 360 parallelism factor, which supports shift values in the range from 0 to 359. Using the selectors and multiplexers, data is shifted by powers of 2. The `Circular Shifter` subsystem uses each bit of shift for appropriate routing and selection of data.

```
set_param([modelName '/DVB-S2 LDPC Encoder/Parity Address Generator'],'Open','off');
open_system([modelName,'/DVB-S2 LDPC Encoder/Circular Shifter']);
```
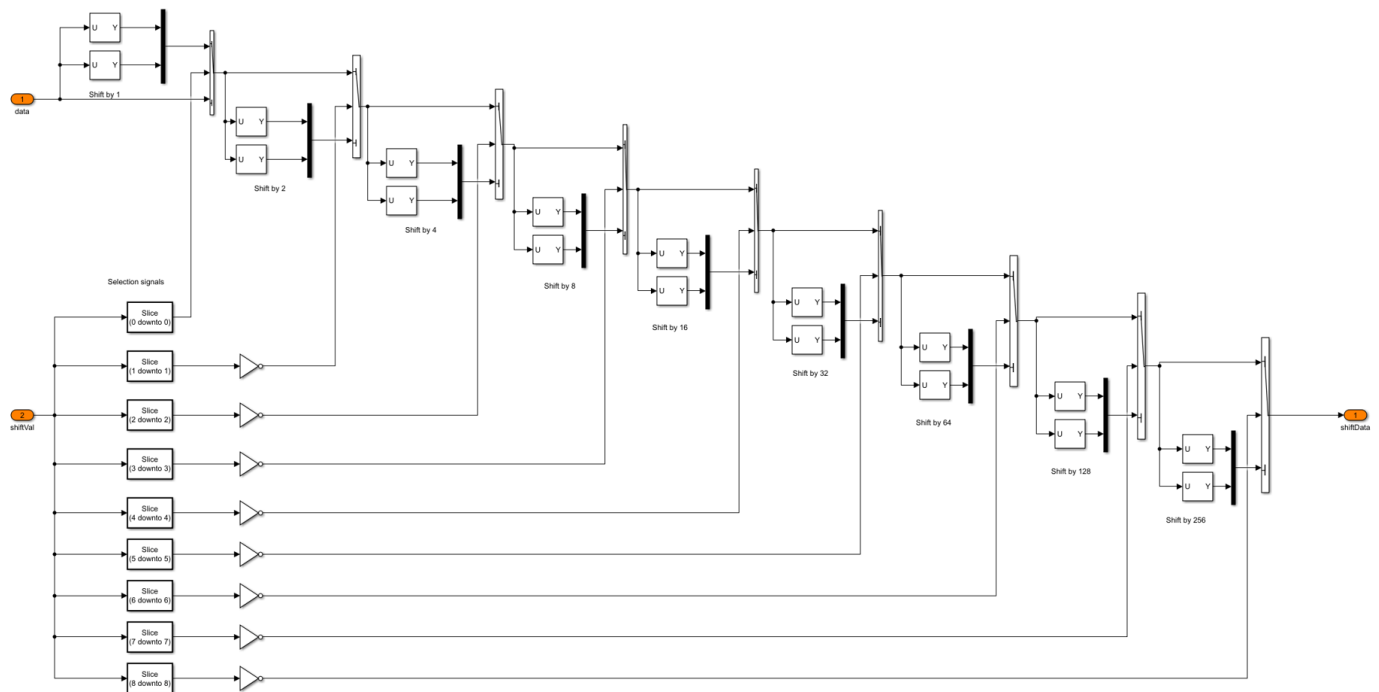
Set Up Input Variables

Choose a series of input values for the FEC frame type and a code rate according to the DVB-S2 standard. You can change the variable values in this section based on your requirements. Specify the `codeRateIdx` values from 0 to 10 that correspond to the `codeRateSet` values `'1/4'`, `'1/3'`, `'2/5'`, `'1/2'`, `'3/5'`, `'2/3'`, `'3/4'`, `'4/5'`, `'5/6'`, `'8/9'`, `'9/10'`.

```
fecFrameSet = {'Normal','Short'};
codeRateSet = {'1/4','1/3','2/5','1/2','3/5','2/3','3/4','4/5','5/6','8/9','9/10'};

frameType = [1 0];                   % FEC frame type
codeRateIdx = [1 0];                 % Code rate index
numFrames = 2;
```

Download DVB-S2 LDPC Parity Matrices Data Set

This example loads a MAT file with DVB-S2 LDPC parity matrices for the reference MATLAB® function. If the MAT file is not available on the MATLAB path, use these commands to download and unzip the MAT file.

```
if ~exist('dvbs2xLDPCParityMatrices.mat','file')
    if ~exist('s2xLDPCParityMatrices.zip','file')
        url = 'https://ssd.mathworks.com/supportfiles/spc/satcom/DVB/s2xLDPCParityMatrices.zip';
        websave('s2xLDPCParityMatrices.zip',url);
        unzip('s2xLDPCParityMatrices.zip');
    end
    addpath('s2xLDPCParityMatrices');
end
```

Generate Input Data

Generate inputs for the `ldpcEncode` helper function with the specified frame type and code rate variables. Create vectors of the frame type and code rate index using the `frameType` and

codeRateIdx variables, respectively. Convert the frames of input data to samples with a control bus signal that indicates the frame boundaries. Provide these vectors and control bus as input to the DVB-S2 LDPC Encoder subsystem.

The encFrameGap variable in the script accommodates the latency of the DVB-S2 LDPC Encoder subsystem for the specified block length and code rate.

```matlab
% Initialize inputs
fecFrameType = fecFrameSet(frameType+1);
codeRate = codeRateSet(codeRateIdx+1);
msg = {numFrames};                  % Input to ldpcEncode  function
refOut = cell(1,numFrames);         % Output from ldpcEncode function

encSampleIn = [];
encStartIn = [];
encEndIn = [];
encValidIn = [];
fFrameIn = [];
codeRateIn = [];

for ii = 1:numFrames
    fFrame =  fecFrameType{ii};

    % Input and code word length calculation
    if strcmpi(fFrame,'Normal')
        cwLen = 64800;
        R = str2num(codeRate{ii}); %#ok<*ST2NM>
    else
        cwLen = 16200;
        ReffList = [1/5 1/3 2/5 4/9 3/5 2/3 11/15 7/9 37/45 8/9];
        RactList = [1/4 1/3 2/5 1/2 3/5 2/3 3/4 4/5 5/6 8/9];
        Reff = ReffList(RactList == str2num(codeRate{ii}));
        R = Reff(1);
    end
    inpLen = cwLen*R;

    % Input bits generation
    msg{ii} = (randi([0 1],inpLen,1));

    % LDPC encoding
    refOut{ii} = satcom.internal.dvbs.ldpcEncode(int8(msg{ii}),codeRate{ii},cwLen);

    % Value of 2000 is selected to accommodate the maximum latency of the
    % block considering different frame type and code rate configurations
    encFrameGap = cwLen + 2000;

    encSampleIn = [encSampleIn msg{ii}' zeros(1,encFrameGap)]; %#ok<*AGROW>
    encStartIn = logical([encStartIn 1 zeros(1,inpLen-1) zeros(1,encFrameGap)]);
    encEndIn = logical([encEndIn zeros(1,inpLen-1) 1 zeros(1,encFrameGap)]);
    encValidIn = logical([encValidIn ones(1,inpLen) zeros(1,encFrameGap)]);
    fFrameIn = logical([fFrameIn repmat(frameType(ii),1,inpLen) zeros(1,encFrameGap)]);
    codeRateIn = [codeRateIn repmat(codeRateIdx(ii),1,inpLen) zeros(1,encFrameGap)];
end

dataIn = timeseries(logical(encSampleIn'));
startIn = timeseries(encStartIn);
endIn = timeseries(encEndIn);
validIn = timeseries(encValidIn);
```

```
frameTypeIn = timeseries(fFrameIn);
codeRateIdxIn = timeseries(codeRateIn);

[columnSum,ramLUT,shiftLUT] = columnShiftRAMLUT(1);

simTime = length(encValidIn);
```

Run Simulink Model

The `DVB-S2 LDPC Encoder` subsystem contains the implementation of the DVB-S2 LDPC Encoder block. Running the model imports the input signal variables `dataIn`, `startIn`, `endIn`, `validIn`, `frameTypeIn`, `codeRateIdxIn`, and `simTime` to the block from the script and exports a stream of encoded output samples `encOut` and a control bus containing `startOut`, `endOut`, and `validOut` signals from the block to the MATLAB® workspace.

```
enc = sim(modelName);
```

Compare Simulink Block Output with MATLAB Function Output

Convert the streaming data output of the `DVB-S2 LDPC Encoder` subsystem to frames. Compare the frames with the output of the `ldpcEncode` helper function.

```
startIdx = find(squeeze(enc.startOut));
endIdx = find(squeeze(enc.endOut));
encData = squeeze(enc.encOut);

encHDL = {numFrames};
for ii = 1:numFrames
    idx = startIdx(ii):endIdx(ii);
    encHDL{ii} = encData(idx);
    HDLOutput = double(encHDL{ii}(1:length(refOut{ii})));
    error = sum(abs(double(refOut{ii})-HDLOutput(:)));
    fprintf('Encoded %s FEC frame and code rate %s: Output data differs by %d bits\n',fecFrameTyp
end

h = warning('off','MATLAB:rmpath:DirNotFound');
rmpath('s2xLDPCParityMatrices');
warning(h);clear h;

Encoded Short FEC frame and code rate 1/3: Output data differs by 0 bits
Encoded Normal FEC frame and code rate 1/4: Output data differs by 0 bits
```
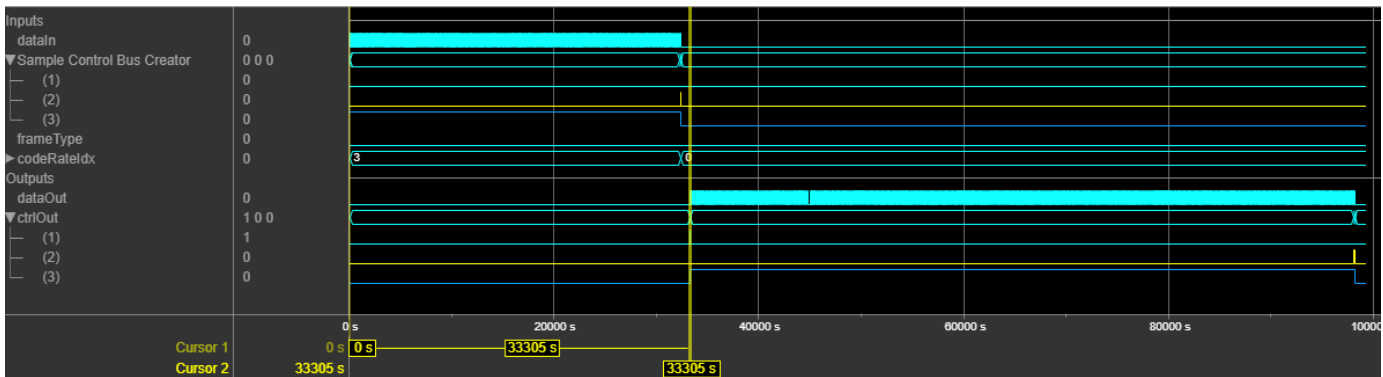
Latency

The latency of the `DVB-S2 LDPC Encoder` model varies with FEC frame types and code rate configurations.

This figure shows the latency of the block when the input **frameType** is `Normal` and **codeRateIdx** is 3.

The following display shows the latency of the `DVB-S2 LDPC Encoder` block for different FEC frame types and code rate configurations.

```
F = table(...
    categorical({'1/4';'1/3';'2/5';'1/2';'3/5';'2/3';'3/4';'4/5';'5/6';'8/9';'9/10'}), ...
    categorical({'16746';'22326';'26790';'33305';'40182';'44166';'49686';'52998';'55206';'58606'
    categorical({'3372';'5586';'6702';'7376';'10050';'11046';'12102';'12816';'13568';'14656'; '-
    'VariableNames',{'Code Rate','Normal FEC Frame','Short FEC Frame'});

disp(F);
```

Code Rate	Normal FEC Frame	Short FEC Frame
1/4	16746	3372
1/3	22326	5586
2/5	26790	6702
1/2	33305	7376
3/5	40182	10050
2/3	44166	11046
3/4	49686	12102
4/5	52998	12816
5/6	55206	13568
8/9	58606	14656
9/10	59334	-

HDL Code Generation

To check and generate HDL for this example, you must have an HDL Coder™ product. Use the `makehdl` and `makehdltb` commands to generate the HDL code and test bench for the `DVB-S2 LDPC Encoder` subsystem.

The `DVB-S2 LDPC Encoder` subsystem is synthesized on a Xilinx® Xilinx Zynq® UltraScale+ MPSoC ZCU102 evaluation board. The resource utilization results are shown in the table below.

```
 F = table(...
    categorical({'Slice LUT';'Slice Registers';'RAMB36';'DSP'; ...
    'Max. Frequency (MHz)'}) ,...
    categorical({'9710';'3860';'17';'0';'384.09'}), ...
    'VariableNames',{'Resources','Values'});

disp(F);
```

```
      Resources          Values
   _____    _____

Slice LUT              9710
Slice Registers        3860
RAMB36                 17
DSP                    0
Max. Frequency (MHz)   384.09
```

References

1 ETSI Standard EN 302 307-1 V1.4.1(2014-11): Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2).

See Also

Blocks
DVB-S2 LDPC Decoder

WLAN HDL LDPC Encoder

This example shows how to implement a WLAN LDPC encoder using Simulink® blocks that are optimized for HDL code generation. This example supports the IEEE® 802.11n™, 802.11ac™, 802.11ax™, and 802.11ad™ standards. The WLAN LDPC Encoder block in this example works in conjunction with the WLAN LDPC Decoder block. To verify the behavior of the WLAN LDPC Encoder block, compare the output of the block with the output of the `ldpcEncode` function.

The IEEE 802.11n (also known as Wi-Fi™ 4) [1], IEEE 802.11ac (also known as Wi-Fi 5) [1] and IEEE 802.11ax (also known as Wi-Fi 6) [2] standards use convolutional codes for forward error correction (FEC). Due to the advantages over convolutional codes, the standards use low-density parity check (LDPC) codes. These standards define quasi cyclic (QC) LDPCs with codeword block lengths of 648, 1296, and 1944 and subblock sizes, also known as expansion factors, of 27, 54, and 81, respectively. For each codeword block length, the standard defines code rates of 1/2, 2/3, 3/4, and 5/6.

The IEEE 802.11ad [3] standard defines a QC LDPC with a codeword block length of 672 and a subblock size of 42. For each codeword block length, the standard define code rates of 1/2, 2/3, 3/4, and 5/6.

The standards define the parity check matrices (PCMs) for LDPCs with various combinations of code word block lengths and code rates.

Model Architecture

This figure shows the architecture block diagram of the WLAN LDPC Encoder block implementation.

Alpha Calculation

Parity Calculation
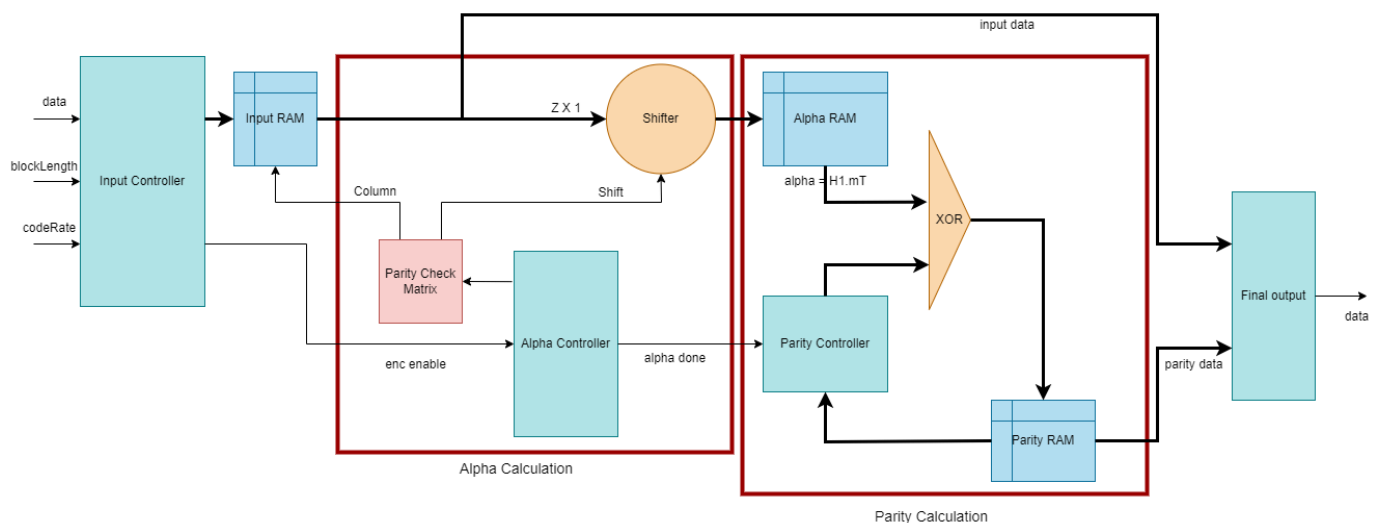
Parity Check Matrix

PCM has the size $N - by - K$, where N is output length and K is input length. You can partition the PCM into two concatenated submatrices H_1 and H_2 such that $H = [H_1 H_2]$, where H_1 an $(N - K) - by - (K)$ submatrix and H_2 an $(N - K) - by - (N - K)$ submatrix. Let $c = [m\,p]$ be a codeword block, with m and p representing the information and parity bit sequences, respectively. The encoding consists of these steps.

1. Calculate Alpha Bit Sequence

From the encoding property, $H.c^T = 0$,

$$[H_1\ H_2][m\ p]^T = 0,$$

$$\alpha + H_2.p^T = 0,$$

$\alpha = H_1.m^T$. This calculation of the alpha bit sequence is the same for the IEEE 802.11n/ac/ax and IEEE 802.11ad standards.

2 .Calculate Parity Bit Sequence

The parity bit sequence p can be expressed as $[p(0)\ p(1)\ \ldots\ p(N-K-1)]$. The $p(0)$ term is subsequence of parity bits with the size of a subblock.

For the IEEE 802.11n/ac/ax LDPC base parity check matrices, H_1 is a sparse matrix and H_2^{-1} has a regular pattern, the array multiplications of $p^T = H_2^{-1}.\alpha$ have linear complexity, which makes it straightforward to compute the parity bit sequence.

For the IEEE 802.11ad LDPC base parity check matrices, H_1 is a sparse matrix and H_2^{-1} has an irregular pattern, which makes calculating the parity bit sequence calculation more complicated. Calculating the parity bit sequence consists of solving this equation:
$$\alpha + H_2[p(0)\ p(1)\ \ldots\ p(N-K-1)]^T = 0.$$

So, two separate models were implemented for IEEE 802.11n/ac/ax and IEEE 802.11ad specified LDPC encoders.

IEEE 802.11n/ac/ax Standard LDPC Encoder

The `WLAN LDPC Encoder` subsystem accepts input data, a control signal, a block length index, and a code rate index. At the start of each frame, the subsystem samples the `blockLenIdx` and `codeRateIdx`. The `inputController` function controls the reading and writing of input data in the `Input RAM` subsystem and enables the encoding after writing the entire frame to RAM. The `Calculate Alpha` subsystem calculates the alpha bit sequence and stores its value in the `Calculate Alpha/Alpha RAM` subsystem. The `Calculate Parity` subsystem calculates the parity and stores them in the `Calculate Parity/Parity RAM` subsystem. The `outputController` function multiplexes and serializes the bits from the `Input RAM` and `Calculate Parity` subsystems.
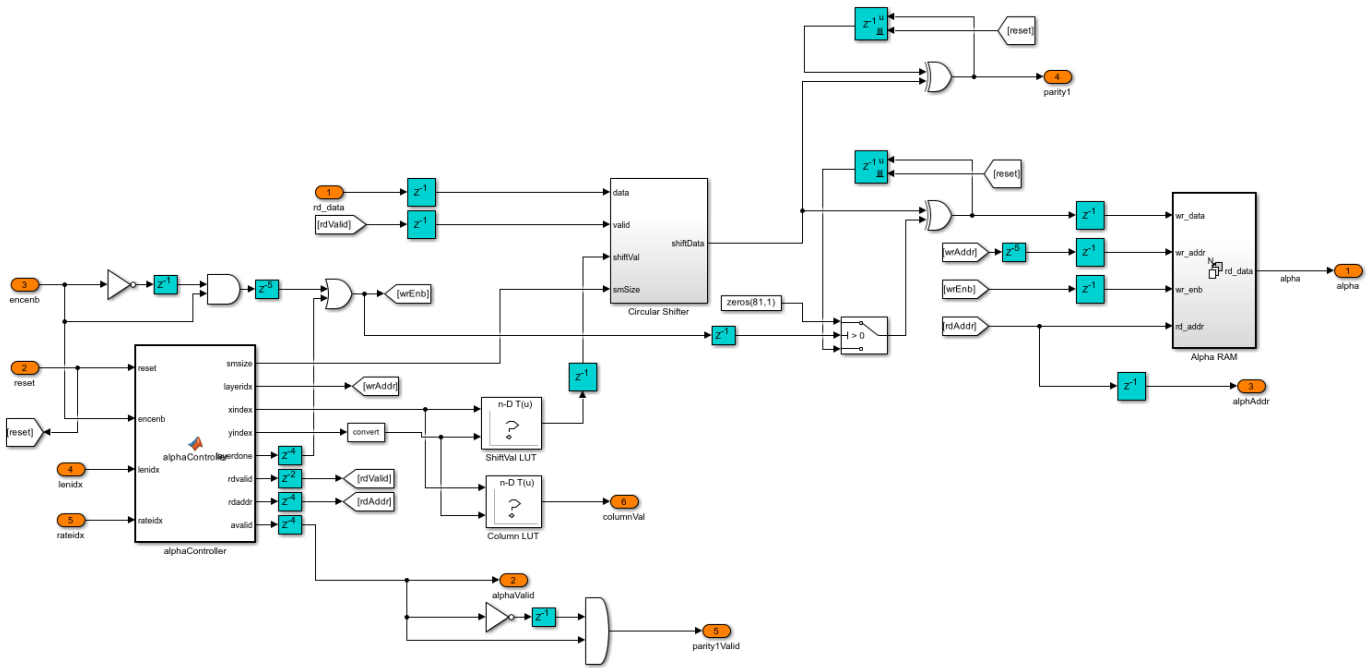
```
modelName = 'wlan11achdlLDPCEncoder';
load_system(modelName);
open_system([modelName '/WLAN LDPC Encoder']);
set_param([modelName '/WLAN LDPC Encoder'],'Open','on');
```
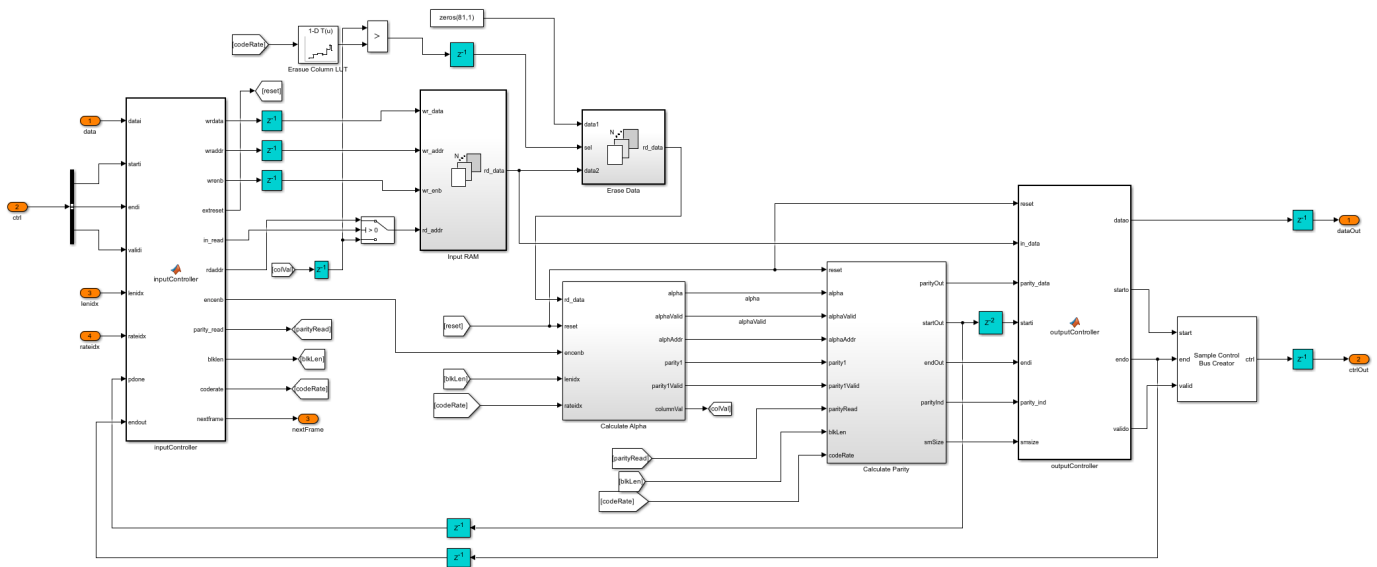
Calculate Alpha Bit Sequence

The `Calculate Alpha` subsystem calculates α bit sequence. Using the parity check matrix specified in the standard [1], the ShiftVal LUT and Column LUT blocks calculate and store the shift values for the `Circular Shifter` subsystem and the address for the `Input RAM` subsystem. The `alphaController` function controls the calculation of the alpha bit values by providing the necessary control signals to the `Circular Shifter` subsystem, ShiftVal LUT block, and Column LUT block. The `Circular Shifter` subsystem shifts the data circularly, XORs the shifted data with previous shifted data, and stores the data in the `Alpha RAM` subsystem. By XORing all the alpha values, the model calculates the first column of the parity bit sequence.

```
set_param([modelName '/WLAN LDPC Encoder'],'Open','off');
open_system([modelName '/WLAN LDPC Encoder/Calculate Alpha']);
```

Calculate Parity Bit Sequence

The `Shiftby1` subsystem shifts the first column of parity bit sequence by one circle. The model then calculates the remaining parity bit subsequences using the shifted first column of parity bit sequence data and the alpha bit values from the `Calculate Alpha` subsystem. The `parityController` function controls the parity calculation as well as the reading and writing of parity data to the `Parity RAM` subsystem.
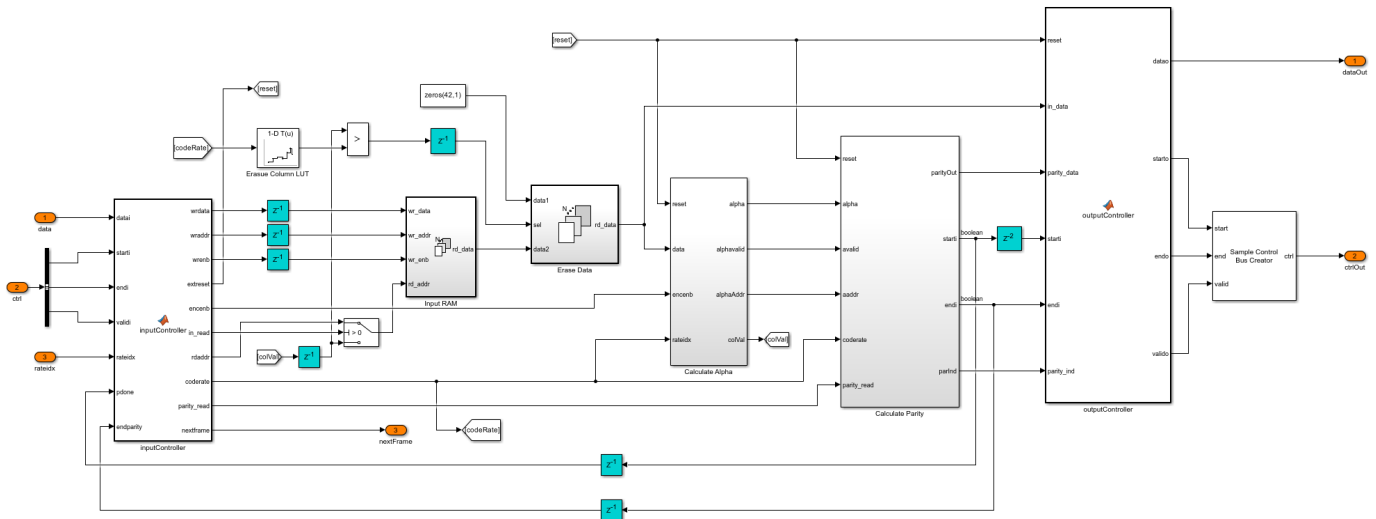
```
set_param([modelName '/WLAN LDPC Encoder/Calculate Alpha'],'Open','off');
open_system([modelName '/WLAN LDPC Encoder/Calculate Parity']);
```
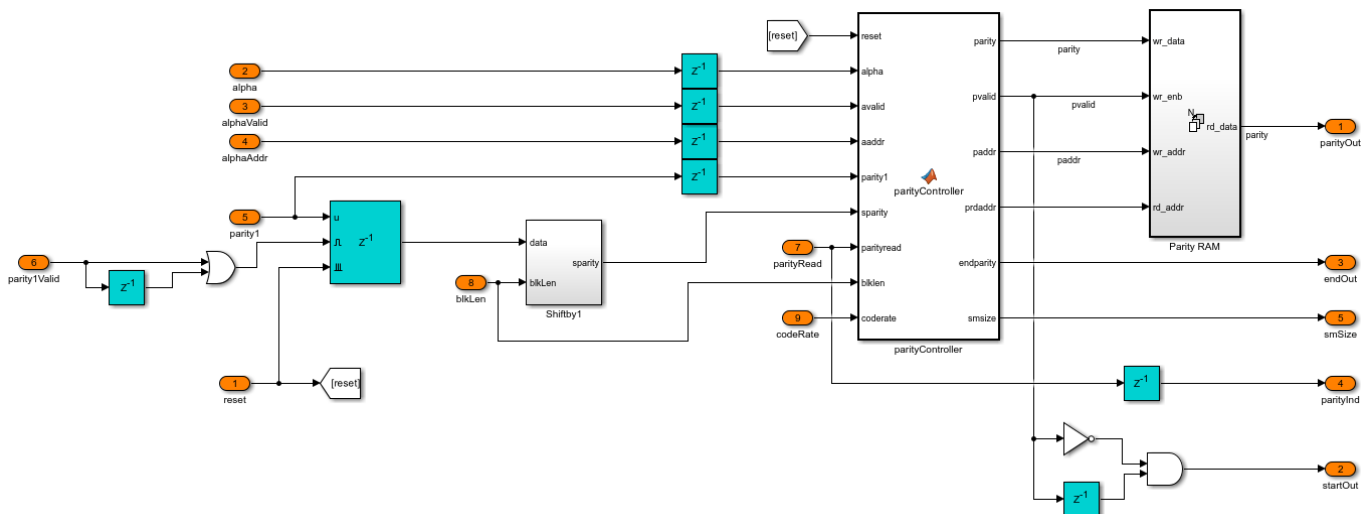

IEEE 802.11ad Standard LDPC Encoder

The `WLAN LDPC Encoder` subsystem accepts input data, a control signal, and a code rate index. The subsystem samples the `codeRateIdx` input at the start of a frame. The architecture is same as the the IEEE 802.11n/ac/ax LDPC Encoder.

```
modelName = 'wlan11adhdlLDPCEncoder';
load_system(modelName);
open_system([modelName '/WLAN LDPC Encoder']);
set_param([modelName '/WLAN LDPC Encoder'],'Open','on');
```


Calculate Alpha Bit Sequence

The `Calculate Alpha` subsystem architecture is same as that of the IEEE 802.11n/ac/ax LDPC Encoder's.

```
set_param([modelName '/WLAN LDPC Encoder'],'Open','off');
open_system([modelName '/WLAN LDPC Encoder/Calculate Alpha']);
```
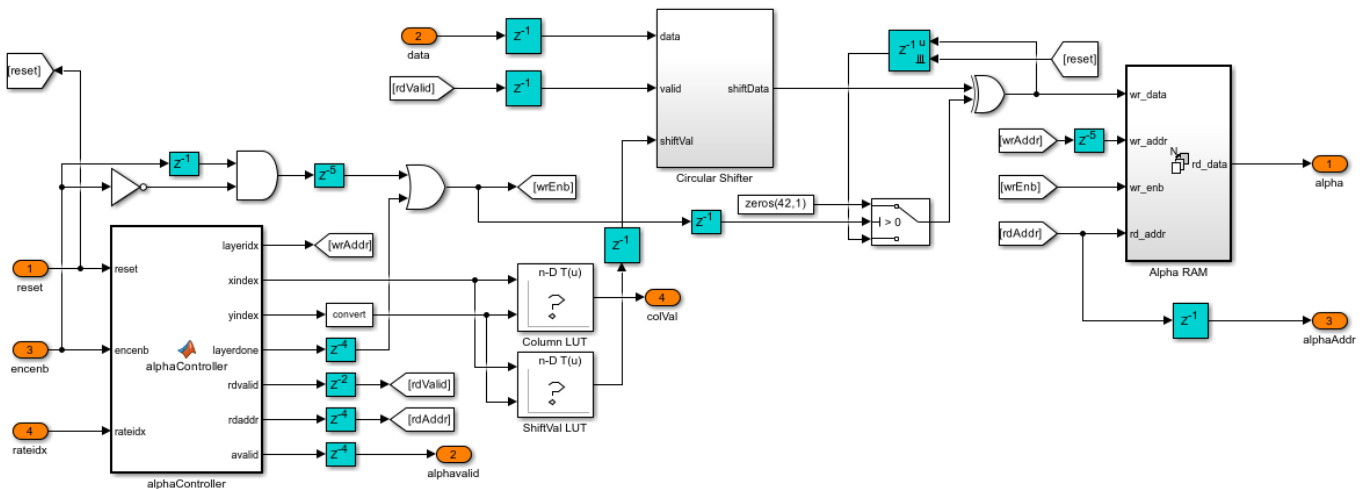

Calculate Parity Bit Sequence

The model calculates the first column of the parity bit sequence by XORing the shifted alpha bit values from the `Circular Shifter` subsystem. Using the first column of parity bit sequence data and the alpha values from the `Calculate Alpha` subsystem, the `parityController` function calculates the remaining parity bit subsequences sequentially and stores them in the `Parity RAM` subsystem. The `parityController` function controls the parity calculation and writing of parity data to the `Parity RAM` subsystem. The `parityReading` function reads the parity data from `Parity RAM` subsystem and outputs parity bits serially.

```
set_param([modelName '/WLAN LDPC Encoder/Calculate Alpha'],'Open','off');
open_system([modelName '/WLAN LDPC Encoder/Calculate Parity']);
```
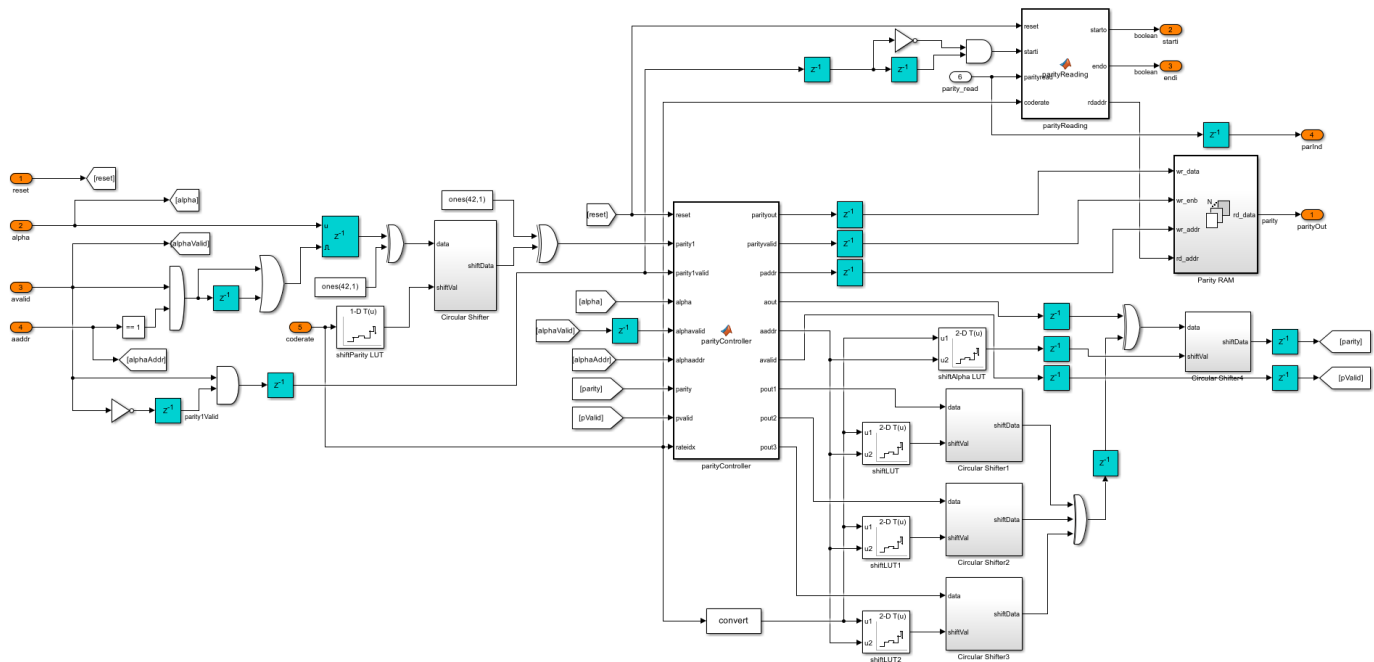

Set Up Input Variables

Choose a WLAN specific standard and input values for the block length and code rate according to your chosen standard. You can change the variable values in this section based on your requirements.

```
standard = 'IEEE 802.11 n/ac/ax';  % IEEE 802.11 n/ac/ax or IEEE 802.11 ad
codeRateIdx = [0; 1; 2; 3];        % Code rate index
blkLenIdx = [2; 1; 2; 0];          % Block length index when standard is
                                   % set to 'IEEE 802.11 n/ac/ax'
numFrames = 4;
```

Generate Input Data

Generate inputs for the `ldpcEncode` function with the specified block length and code rate variables. Create vectors of block length index and code rate index using the `blockLenIdx` and `codeRateIdx` variables, respectively. Convert the frames of input data to samples with a control bus signal that indicates the frame boundaries. Provide these vectors and the control bus as inputs to the `WLAN LDPC Encoder` subsystem.

The `encFrameGap` variable accommodates the latency of the `WLAN LDPC Encoder` subsystem for the specified block length and code rate.

```matlab
msg = {numFrames};
refOut = cell(1,numFrames);

encSampleIn = [];
encStartIn = [];
encEndIn = [];
encValidIn = [];
encBlkLenIdxIn = [];
encCodeRateIdxIn = [];

close_system(modelName);

% Calculate input and output codeword length
for idx = 1:numFrames
    if strcmpi(standard,'IEEE 802.11 n/ac/ax')
        blockLenSet = [648,1296,1944,1944];
        rateSet = {'1/2','2/3','3/4','5/6'};

        blkLen = blockLenSet(blkLenIdx(idx)+1);
        codeRate = rateSet{codeRateIdx(idx)+1};
        modelName = 'wlan11achdlLDPCEncoder';
    else
        rateSet = {'1/2','5/8','3/4','13/16'};
        blkLen = 672;
        codeRate = rateSet{codeRateIdx(idx)+1};
        modelName = 'wlan11adhdlLDPCEncoder';
    end

    % Encoder configuration
    encConfig = wlanEncoderConfiguration(blkLen,codeRate);

    % Input bits generation
    msg{idx} = randi([0 1],encConfig.NumInformationBits,1,'int8');

    % LDPC encoding
    refOut{idx} = ldpcEncode(msg{idx}, encConfig);

    len = length(msg{idx});
    encFrameGap = 2000 + len;
    encIn =  msg{idx}';

    encSampleIn = [encSampleIn encIn zeros(size(encIn,1),encFrameGap)]; %#ok<*AGROW>
    encStartIn = logical([encStartIn 1 zeros(1,len-1) zeros(1,encFrameGap)]);
    encEndIn = logical([encEndIn zeros(1,len-1) 1 zeros(1,encFrameGap)]);
    encValidIn = logical([encValidIn ones(1,len) zeros(1,encFrameGap)]);
    encBlkLenIdxIn = ([encBlkLenIdxIn repmat(blkLenIdx(idx),1,len) zeros(1,encFrameGap)]);
    encCodeRateIdxIn = ([encCodeRateIdxIn repmat(codeRateIdx(idx),1,len) zeros(1,encFrameGap)]);
end

dataIn = timeseries(encSampleIn'>0);
startIn = timeseries(encStartIn);
endIn = timeseries(encEndIn);
validIn = timeseries(encValidIn);

if strcmpi(standard,'IEEE 802.11 n/ac/ax')
    blockLenIdx = timeseries(fi(encBlkLenIdxIn,0,2,0));
end
codeRateIdx = timeseries(fi(encCodeRateIdxIn,0,2,0));
```

```
simTime = length(encValidIn);

[columnLUT,shiftLUT] = columnShiftLUTWLAN(standard);
```

Run WLAN LDPC Encoder Model

The `WLAN LDPC Encoder` subsystem contains the implementation of the WLAN LDPC Encoder block. Run the model to import the input signal variables `dataIn`, `startIn`, `endIn`, `validIn`, `blockLenIdx`, `codeRateIdx`, and `simTime` to the block from the script. The model exports a stream of encoded output samples `encOut` and a control bus containing the `startOut`, `endOut`, and `validOut` signals to the MATLAB® workspace.

```
load_system(modelName);
open_system(modelName);
sim(modelName);
```
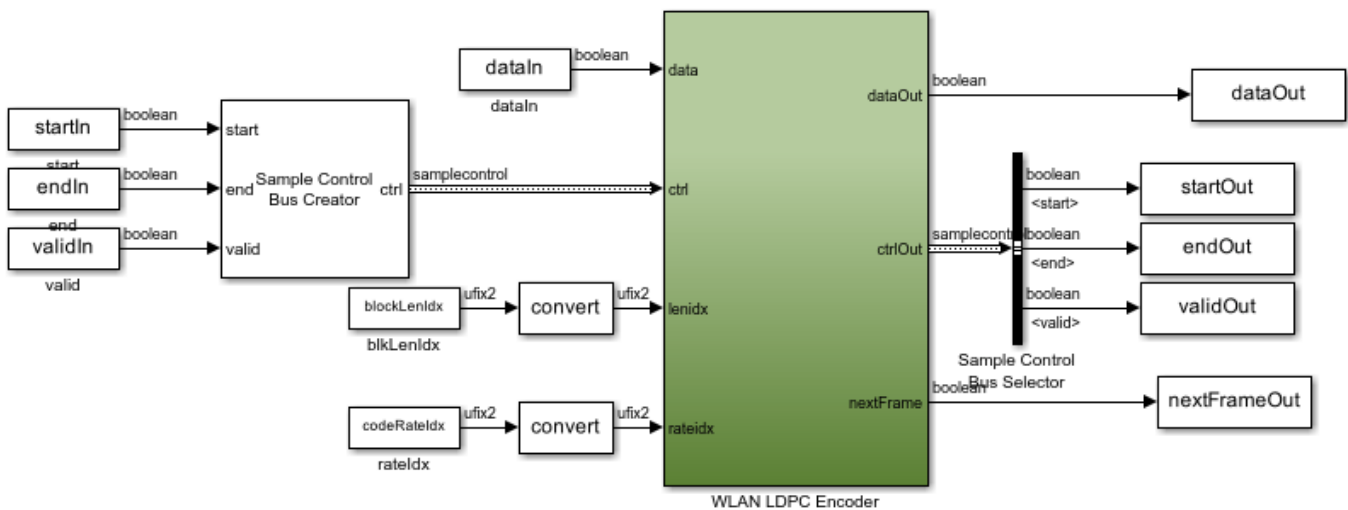

Copyright 2022 The MathWorks, Inc.

Compare Block Output with Function Output

Convert the streaming data output of the `WLAN LDPC Encoder` subsystem to frames. Compare the frames with the output of the `ldpcEncode` function.

```
startIdx = find(squeeze(startOut));
endIdx = find(squeeze(endOut));
enc = squeeze(dataOut);

encHDL = {numFrames};
for i = 1:numFrames
    idx = startIdx(i):endIdx(i);
    encHDL{i} = enc(idx);
    HDLOutput = encHDL{i};
    error = sum(abs(double(refOut{i})-HDLOutput(:)));
    fprintf('Encoded frame %d: Output data differs by %d bits\n',i,error);
end
```

```
close_system(modelName,0);
```

```
Encoded frame 1: Output data differs by 0 bits
Encoded frame 2: Output data differs by 0 bits
Encoded frame 3: Output data differs by 0 bits
Encoded frame 4: Output data differs by 0 bits
```

Latency

The latency of the `WLAN LDPC Encoder` block varies with the selected standard, block length, and code rate configurations. This figure shows the latency of the block when you set the **blockLenIdx** input port to 2 and the **codeRateIdx** input port to 0.
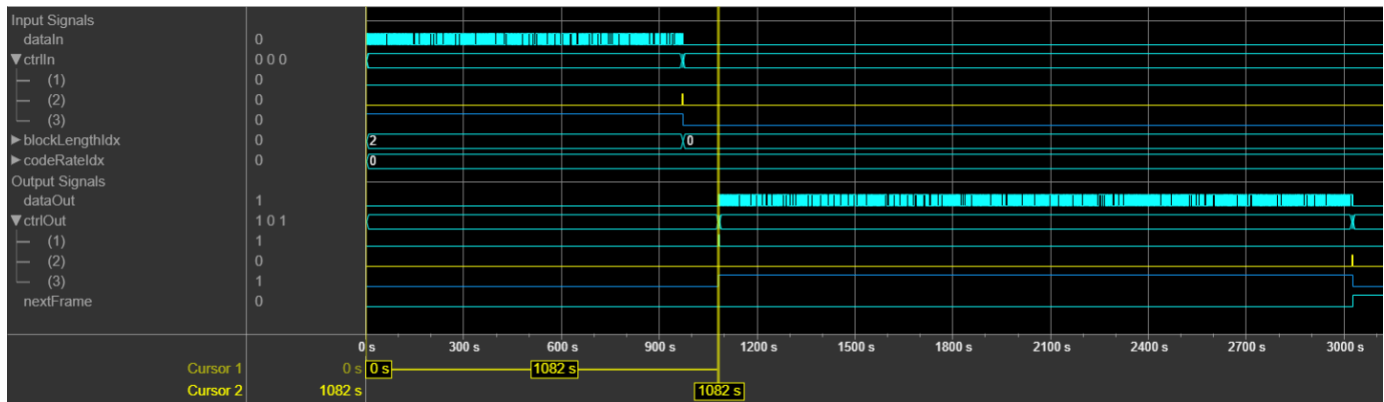
This table shows the latency of the WLAN LDPC Encoder block for different standards and configurations.

Standard	Block Length (N)	Code Rate (R)	Latency
IEEE 802.11 n/ac/ax	648	1/2	436
		2/3	540
		3/4	592
		5/6	644
	1296	1/2	758
		2/3	972
		3/4	1078
		5/6	1181
	1944	1/2	1082
		2/3	1404
		3/4	1561
		5/6	1715
IEEE 802.11 ad	672	1/2	400
		5/8	482
		3/4	572
		13/16	603

Generate HDL Code

To check and generate HDL code for this example, you must have an HDL Coder™ license. Use the `makehdl` and `makehdltb` commands to generate the HDL code and test bench for the `WLAN LDPC Encoder` subsystem.

The `WLAN LDPC Encoder` subsystem is synthesized on a Xilinx® Xilinx Zynq® UltraScale+ MPSoC ZCU102 evaluation board. This table shows the post place and route resource utilization results.

```
 F = table(...
    categorical({'IEEE 802.11 n/ac/ax';'IEEE 802.11 ad'}),...
    categorical({'2495';'2482'}),...
    categorical({'2153';'1793'}),...
    categorical({'0.5';'0.5'}),...
    categorical({'514.88';'516'}),...
    'VariableNames',{'Standard','Slice LUTs','Slice Registers','BRAM', ...
    'Max. Frequency (MHz)'});
```

```
disp(F);
```

Standard	Slice LUTs	Slice Registers	BRAM	Max. Frequency (MHz)
IEEE 802.11 n/ac/ax	2495	2153	0.5	514.88
IEEE 802.11 ad	2482	1793	0.5	516

References

1   IEEE Std 802.11™-2020. IEEE Standard for Information Technology - Telecommunications and Information Exchange between Systems - Local and Metropolitan Area Networks - Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.

2   IEEE Std 802.11ax™-2021. IEEE Standard for Information Technology - Telecommunications and Information Exchange between Systems - Local and Metropolitan Area Networks - Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications - Amendment 1: Enhancements for High-Efficiency WLAN.

3   IEEE Std 802.11ad™-2012. IEEE Standard for Information technology - Telecommunications and information exchange between systems--Local and Metropolitan Area Networks - Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. Amendment 3: Enhancements for Very High Throughput in the 60 GHz Band.

See Also

Blocks
WLAN LDPC Decoder

DVB-S2 HDL BCH Encoder

This example shows how to implement a DVB-S2 BCH encoder using Simulink® blocks that are optimized for HDL code generation.

To verify the behavior of the DVB-S2 HDL BCH Encoder example, you compare the output of the example with the `bchEncode` helper function.

Introduction

Digital Video Broadcast Satellite Second Generation (DVB-S2) is a European Telecommunications Standards Institute (ETSI) standard for digital data transmission through satellites. DVB-S2 offers powerful forward error correction (FEC) based on low-density parity-check (LDPC) codes concatenated with Bose-Chaudhuri-Hocquenghem (BCH) codes.

The FEC mechanism performs outer coding with BCH codes and inner coding with LDPC codes. This mechanism accepts baseband frames (BBFRAMEs) of K_{bch} bits as input and gives out forward error correction frames (FECFRAMEs) of n_{ldpc} bits as output as specified in the section 5.3 of ETSI EN 302 307-1 [1]. This figure shows the frame format of FECFRAME data.
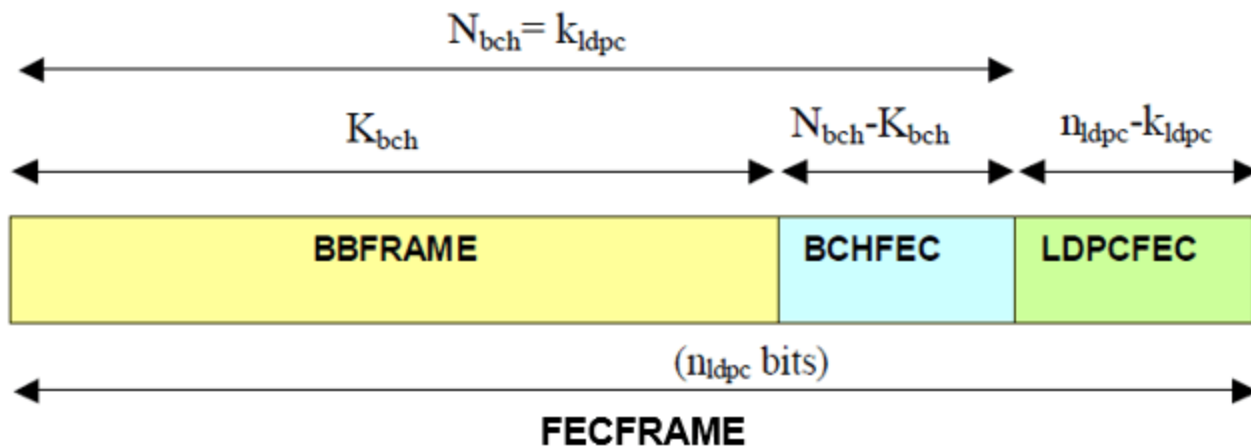
The DVB-S2 standard specifies 11 code rates for normal frames and 10 code rates for short frames. In tables 5a and 5b in the section 5.3 of [1], the standard defines BCH code rates for different uncoded block lengths and for different frame types.
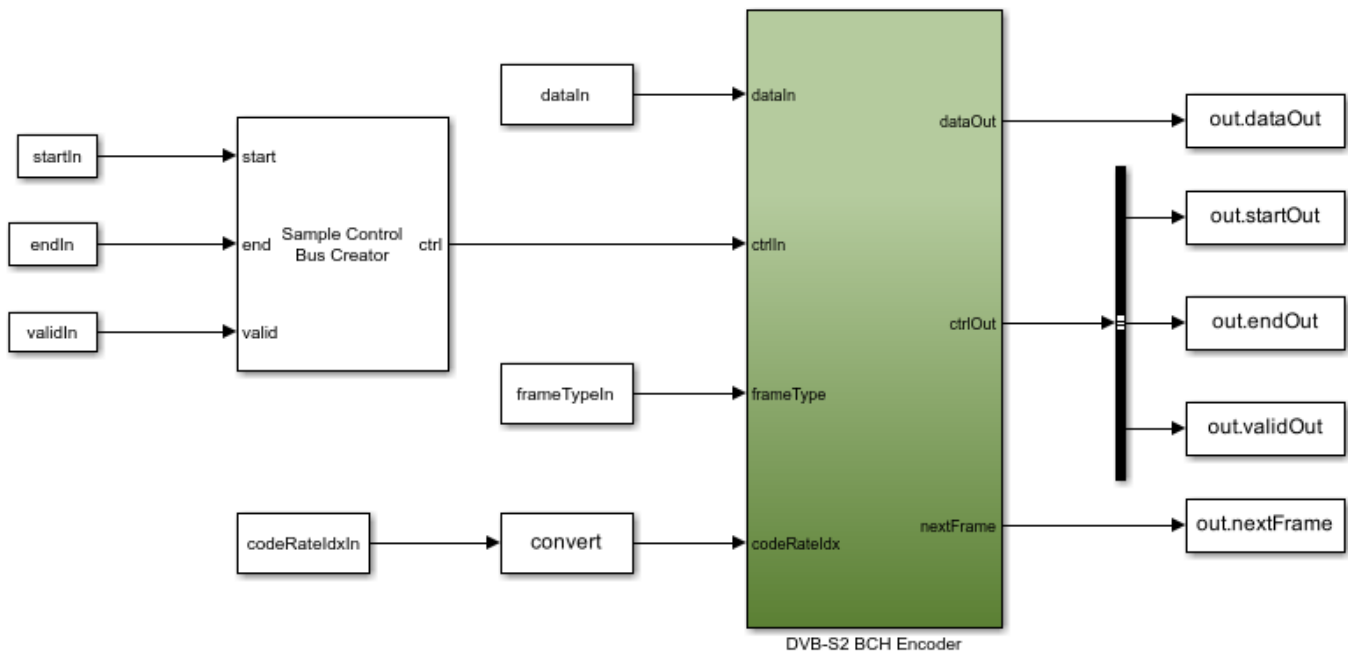
Model Architecture

This figure shows the architecture block diagram of the DVB-S2 BCH Encoder example implementation.
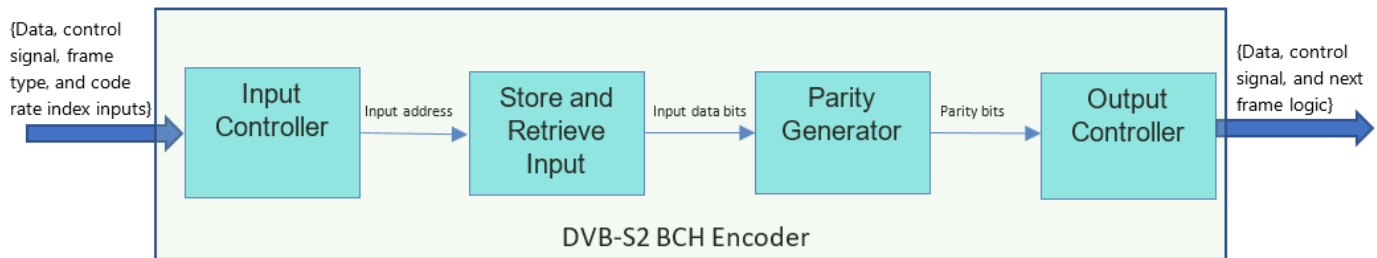
This figure shows the top-level structure of the `dvbs2hdlBCHEncoder` model. You can generate the HDL code for the `DVB-S2 BCH Encoder` subsystem in the model.

```
modelName = 'dvbs2hdlBCHEncoder';
open_system(modelName);
set_param(modelName,'Open','on');
```


Copyright 2022 The MathWorks, Inc.

DVB-S2 BCH Encoder

The `DVB-S2 BCH Encoder` subsystem accepts input data, a control signal, a frame type, and a code rate index. The subsystem samples *frameTypeIn* and *codeRateIdxIn* signals at the start of the frame. The `Input Controller` subsystem controls the write and read operation of the input data to and from the RAM, respectively, inside the `Store and Retrieve Input` subsystem. The `DVB-S2 BCH Encoder` subsystem performs the BCH encoding after writing the whole frame to the RAM. Use tables 5a, 5b, 6a, and 6b in section 5.3 of [1] to calculate the generator polynomial. Using the `Parity Generator` subsystem, generate parity bits using the generator polynomial. This subsystem also stores and retrieves parity bits using RAM and performs parity indexing. Using the `Output`

`Controller` subsystem, append the input data with the generated parity bits and generate output control signals.

```
set_param(modelName,'Open','off');
open_system([modelName '/DVB-S2 BCH Encoder']);
```
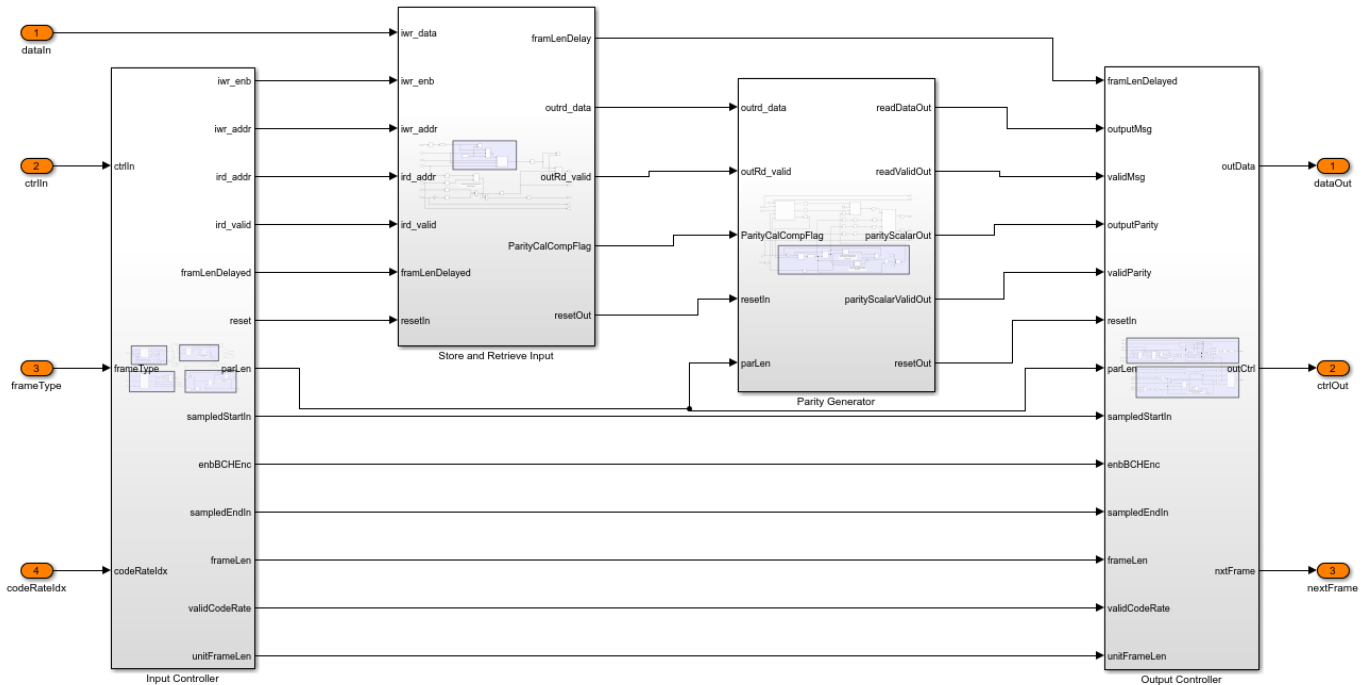

Input Controller

The `Input Controller` subsystem consists of these sections:

- The **Sample Bus Controller** section accepts the input sample control bus and identifies the start, end, and valid signals. This section also generates the reset signal.

- The **Sampler** section samples the *frameType* and *codeRateIdx* using *start* signal and outputs the corresponding input frame length and parity length. This section also validates frame parameters and returns the result in the *validateFrameFlag* indicator.

- The **Write Address Generator** section has a write counter that runs till the input frame length and generates the address at which it stores the input bits inside the RAM.

- The **Read Address Generator** section has a read counter that runs till the input frame length and generates the address for reading the stored input bits from the RAM. *validFrameFlag* signal enables read address generation only after storing the entire input frame inside the RAM.
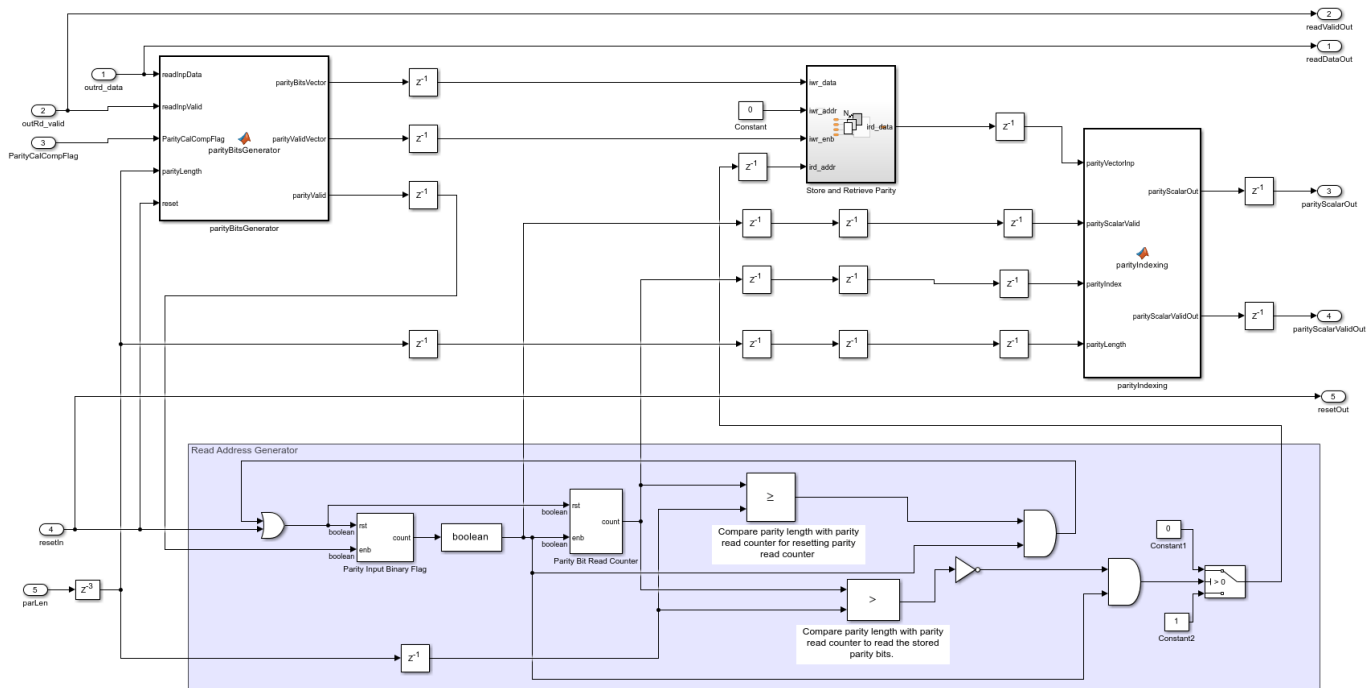
```
set_param([modelName '/DVB-S2 BCH Encoder'],'Open','off');
open_system([modelName '/DVB-S2 BCH Encoder/Input Controller']);
```
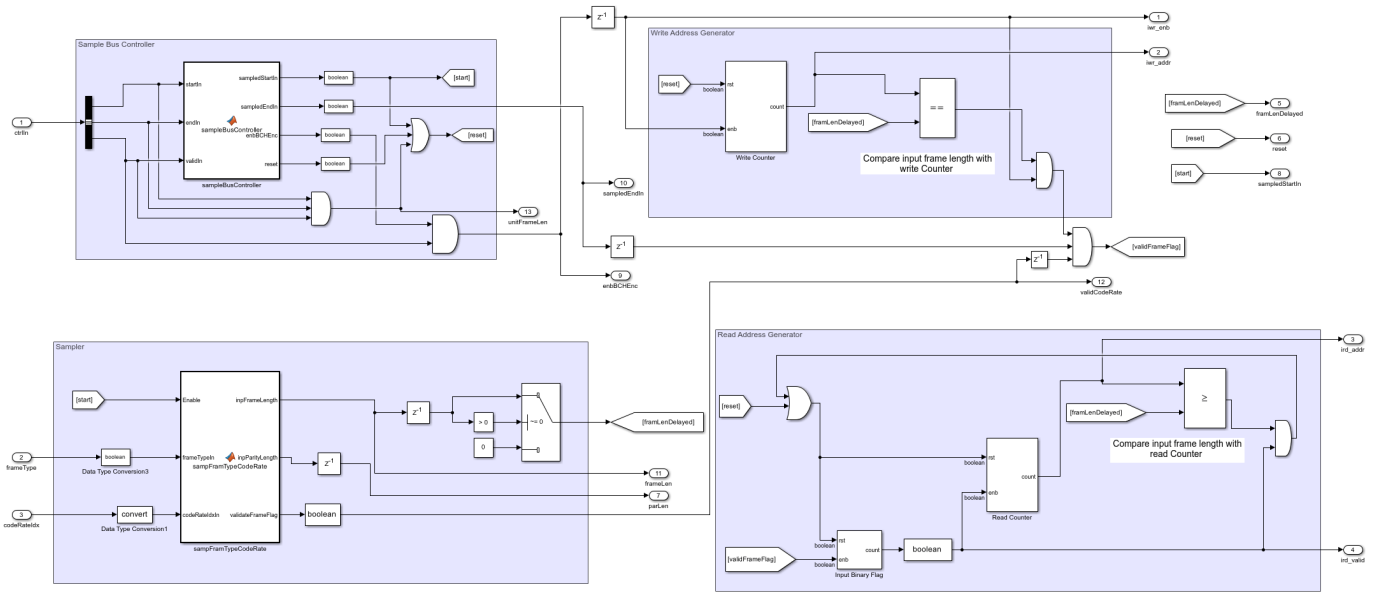
Store and Retrieve Input

After generating the read and write address, the `Store and Retrieve Input` subsystem uses the Data RAM block to read and write the bits. For each high value of the *outRd_valid* signal, this subsystem reads a data bit from the Data RAM block and feeds it into the `Parity Generator` subsystem. The `Store and Retrieve Input` subsystem also compares the read addresses with the input frame length and indicates the end of the parity calculation using *ParityCalCompFlag* signal.

```
set_param([modelName '/DVB-S2 BCH Encoder'],'Open','off');
open_system([modelName '/DVB-S2 BCH Encoder/Store and Retrieve Input']);
```
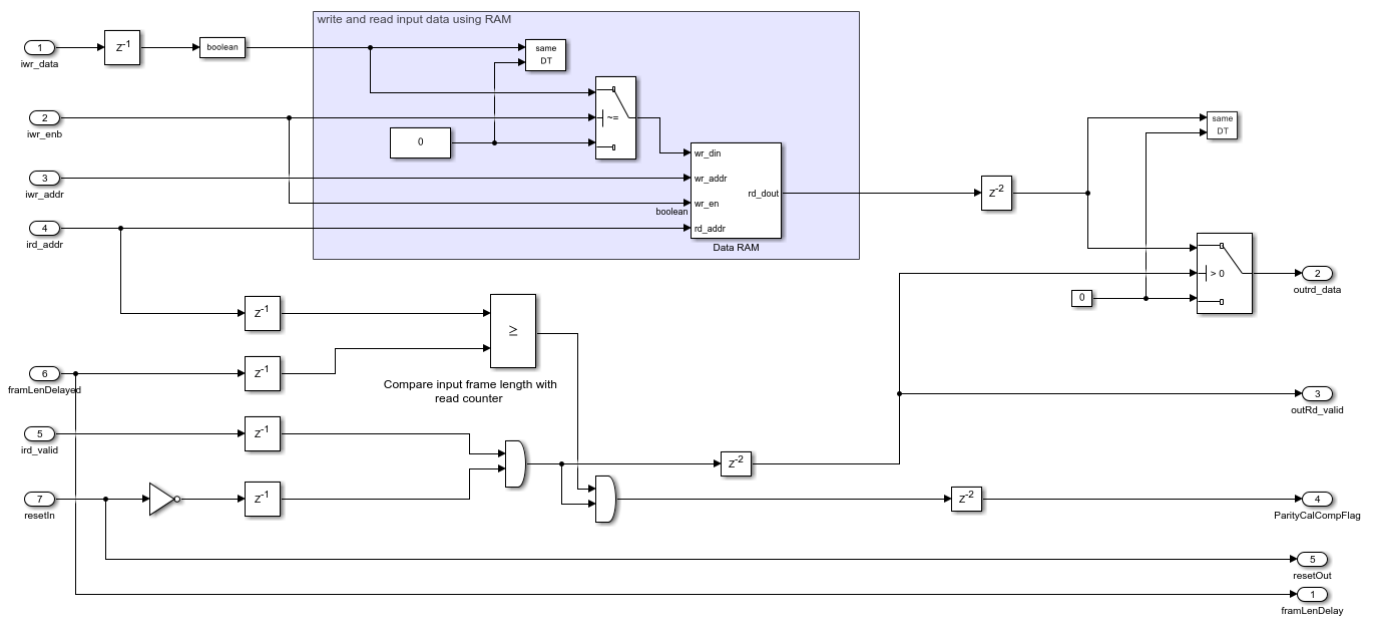

Parity Generator

The `parityBitsGenerator` function generates a parity bit vector of length 192 for every input data bit according to the section 5.3.1 in [1]. For the parity lengths 168, 160, and 128, this function generates parity bit vectors of corresponding lengths appends zeros to create a vector of the length 192. The write address **iwr_addr** is always 0. The **Read Address Generator** section generates the read address. The `Store and Retrieve Parity` subsystem stores and retrieves the vector parity bits. The `parityIndexing` function block outputs the scalar parity bits from the parity vector based on the parity index that the Parity Bit Read Counter block generates.

```
set_param([modelName '/DVB-S2 BCH Encoder'],'Open','off');
open_system([modelName '/DVB-S2 BCH Encoder/Parity Generator']);
```


Output Controller

The `Output Controller` subsystem consists of these sections:

- The **Next Frame Generator** generates the next frame logic using the previous state of *nxtFrame*, *sampledStartIn*, *sampledEndIn*, *endOut*, *enbBCHEnc*, *unitFrameLen*, *validCodeRate*, and *frameLen* signals.

- The **Output Generator** section multiplexes the message and parity to generate the output data. This subsystem also generates the output sample control bus using an Output Counter block that runs till the output frame length.

```
set_param([modelName '/DVB-S2 BCH Encoder'],'Open','off');
open_system([modelName '/DVB-S2 BCH Encoder/Output Controller']);
```
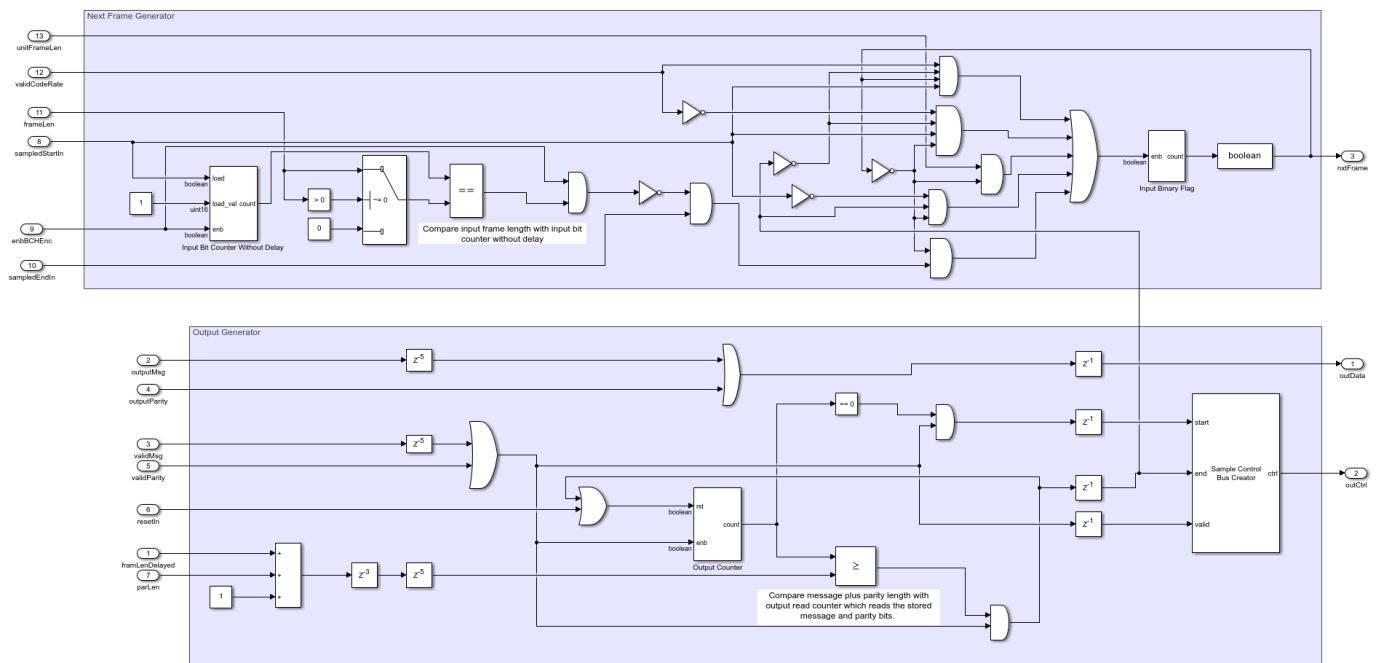
Set Up Input Variables

Choose input values for the FEC frame type and the corresponding code rates according to the DVB-S2 standard. You can change the variable values in this section based on your requirements. Specify the `codeRateIdx` values as integers in the range [0, 10] that correspond to the `codeRateSet` values `'1/4'`, `'1/3'`, `'2/5'`, `'1/2'`, `'3/5'`, `'2/3'`, `'3/4'`, `'4/5'`, `'5/6'`, `'8/9'`, and `'9/10'`.

```
fecFrameSet = {'Normal','Short'};
codeRateSet = {'1/4','1/3','2/5','1/2','3/5','2/3','3/4','4/5','5/6','8/9','9/10'};

frameType = [1 0];    % FEC frame type
codeRateIdx = [1 0]; % Code rate index
numFrames = 2;
```

Generate Input Data

Generate inputs for the `bchEncode` helper function with the specified frame type and code rate variables. Create vectors of the frame type and code rate index using the `frameTypeIn` and `codeRateIdxIn` variables, respectively. Convert the frames of input data to samples with a control bus signal that indicates the frame boundaries. Provide these vectors and the control bus as inputs to the `DVB-S2 BCH Encoder` subsystem.

The `encFrameGap` variable accommodates the latency of the `DVB-S2 BCH Encoder` subsystem for the specified block length and code rate.

```
fecFrameType = fecFrameSet(frameType+1);
codeRate = codeRateSet(codeRateIdx+1);
msg = {numFrames};
refOut = cell(1,numFrames);

encSampleIn = [];
encStartIn = [];
encEndIn = [];
```

```matlab
encValidIn = [];
fFrameIn = [];
codeRateIn = [];

for framIdx = 1:numFrames
    fFrame =  fecFrameType{framIdx};

    % Input and code word length calculation
    if strcmpi(fFrame,'Normal')
        cwLen = 64800;
        inpFrameLenList = [16008 21408 25728 32208 38688 43040 48408 51648 53840 57472 58192];
        inpRateList = [1/4 1/3 2/5 1/2 3/5 2/3 3/4 4/5 5/6 8/9 9/10];
        frameLenArr = inpFrameLenList(inpRateList == str2num(codeRate{framIdx}));
        frameLen = frameLenArr(1);
    else
        cwLen = 16200;
        inpFrameLenList = [3072  5232  6312  7032  9552 10632 11712 12432 13152 14232];
        inpRateList = [1/4 1/3 2/5 1/2 3/5 2/3 3/4 4/5 5/6 8/9];
        frameLenArr = inpFrameLenList(inpRateList == str2num(codeRate{framIdx}));
        frameLen = frameLenArr(1);
    end
    inpLen = frameLen;

    % Input bits generation
    msg{framIdx} = (randi([0 1],inpLen,1));

    % BCH encoding
    refOut{framIdx} = satcom.internal.dvbs.bchEncode(int8(msg{framIdx}),inpLen,cwLen);

    % Select the value cwLen*2 to accommodate the maximum latency of the
    % block considering different frame type and code rate configurations
    encFrameGap = cwLen*2;

    encSampleIn = [encSampleIn msg{framIdx}' zeros(1,encFrameGap)]; %#ok<*AGROW>
    encStartIn = logical([encStartIn 1 zeros(1,inpLen-1) zeros(1,encFrameGap)]);
    encEndIn = logical([encEndIn zeros(1,inpLen-1) 1 zeros(1,encFrameGap)]);
    encValidIn = logical([encValidIn ones(1,inpLen) zeros(1,encFrameGap)]);
    fFrameIn = logical([fFrameIn repmat(frameType(framIdx),1,inpLen) zeros(1,encFrameGap)]);
    codeRateIn = [codeRateIn repmat(codeRateIdx(framIdx),1,inpLen) zeros(1,encFrameGap)];
end

dataIn = logical(encSampleIn');
startIn = encStartIn;
endIn = encEndIn;
validIn = encValidIn;
frameTypeIn = fFrameIn;
codeRateIdxIn = codeRateIn;

simTime = length(encValidIn);
```

Run Simulink Model

The `DVB-S2 BCH Encoder` subsystem contains the implementation of the DVB-S2 BCH Encoder. Run the model to import the input signal variables `dataIn`, `startIn`, `endIn`, `validIn`, `frameTypeIn`, `codeRateIdxIn`, and `simTime` to the example from the script. The model exports a stream of encoded output samples `encOut` and a control bus containing `startOut`, `endOut`, and `validOut` signals to the MATLAB® workspace.

```
enc = sim(modelName);
```

Compare Block Output with Function Output

Convert the streaming output data of the `DVB-S2 BCH Encoder` subsystem to frames. Compare the frames with the output of the `bchEncode` helper function.

```
startIdx = find(squeeze(enc.startOut));
endIdx = find(squeeze(enc.endOut));
encData = squeeze(enc.dataOut);

encHDL = {numFrames};
for framIdx = 1:numFrames
    idx = startIdx(framIdx):endIdx(framIdx);
    encHDL{framIdx} = encData(idx);
    HDLOutput = double(encHDL{framIdx}(1:length(refOut{framIdx})));
    error = sum(abs(double(refOut{framIdx})-HDLOutput(:)));
    fprintf('Encoded %s FEC frame and code rate %s: Output data differs by %d bits\n',fecFrameTyp
end

Encoded Short FEC frame and code rate 1/3: Output data differs by 0 bits
Encoded Normal FEC frame and code rate 1/4: Output data differs by 0 bits
```
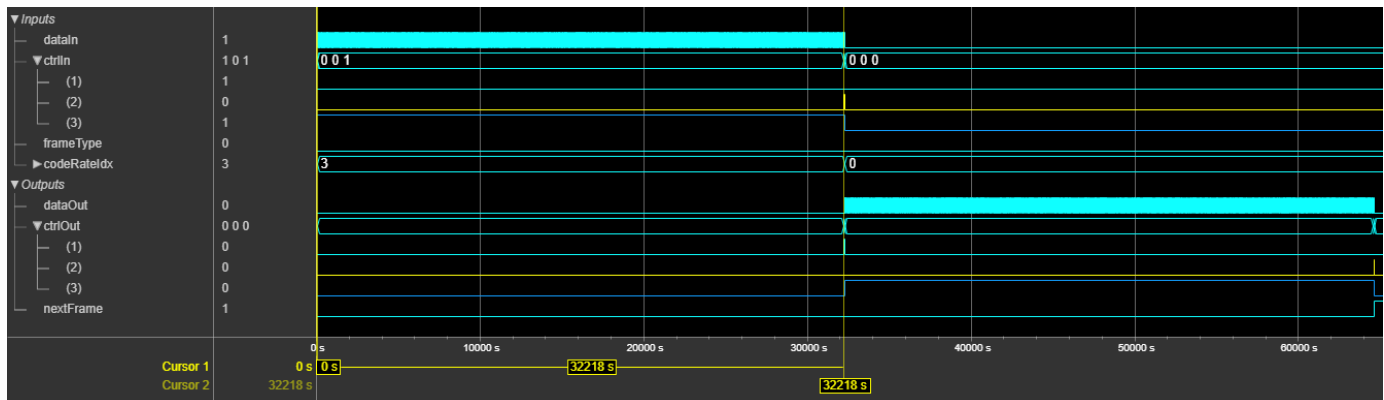
Calculate Latency

The latency of the DVB-S2 BCH Encoder in this example is $K_{bch} + 10$ clock cycles, where K_{bch} is input frame length of the BCH encoder.

This figure shows the latency of the DVB-S2 BCH Encoder example when you set the **frameType** input port to 0 and the **codeRateIdx** input port to 3.

Generate HDL Code

To check and generate HDL code for this example, you must have an HDL Coder™ license. Use the `makehdl` and `makehdltb` commands to generate the HDL code and test bench for the `DVB-S2 BCH Encoder` subsystem.

Synthesize the `DVB-S2 BCH Encoder` subsystem on a Xilinx® Xilinx Zynq® UltraScale+ RFSoC xczu29dr-ffvf1760-2-e. This table shows the resource utilization results.

```
F = table(...
    categorical({'Slice LUT';'Slice Registers';'RAMB36';'DSP'; ...
```

```
      'Max. Frequency (MHz)'}) ,...
      categorical({'1401';'1032';'2';'0';'602.71'}), ...
      'VariableNames',{'Resources','Values'});
```

```
disp(F);
```

```
          Resources          Values
      _____    _____

      Slice LUT               1401
      Slice Registers         1032
      RAMB36                  2
      DSP                     0
      Max. Frequency (MHz)    602.71
```

References

1 ETSI 302 307-1 "Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications; Part 1: DVB-S2." Digital Video Broadcasting. https://www.etsi.org

Blocks

DVB-S2 BCH Decoder

Reference Applications

NR HDL Reference Applications Overview

Wireless HDL Toolbox contains several reference applications that implement and verify parts of a 5G NR downlink receiver. This page illustrates a workflow for designing and verifying complex algorithms for hardware, explains how the examples relate to each other, and shows which parts of the downlink receiver algorithm the examples cover.

Family of Examples

The "5G Reference Applications" page shows a family of examples that describe a workflow for designing and deploying an algorithm to hardware. Different examples describe different parts of the workflow. This diagram shows the complete workflow.
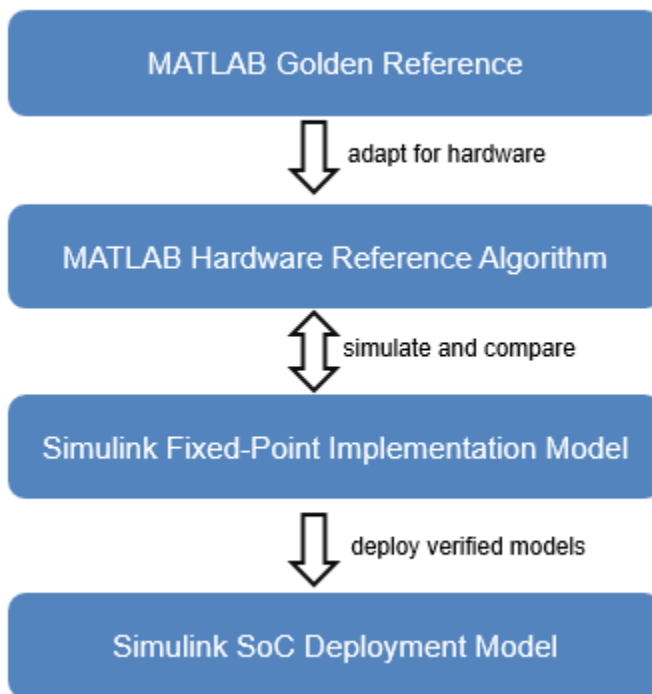
Each step in this workflow is demonstrated by one or more related examples.

1. The *MATLAB Golden Reference Algorithm* step consists of the "NR Cell Search and MIB and SIB1 Recovery" (5G Toolbox) example, which shows the floating-point golden reference algorithm.

2. The *MATLAB Hardware Reference Algorithm* step consists of the "NR HDL Downlink Receiver MATLAB Reference" on page 5-68 example, which models hardware friendly algorithms and generates test waveforms. This MATLAB code covers cell search, MIB recovery, and SIB1 recovery, and bridges the gap between a mathematical algorithm and its hardware implementation. This code models the data flow and sample rate used in the hardware implementation, operates on vectors and matrices of floating-point data samples, and does not support HDL code generation. For the success rate of the hardware reference algorithm, see "NR HDL Receiver Performance" on page 5-5.

3. The *Simulink Fixed-Point Implementation Model* step consists of multiple examples that cover sections of the downlink receiver chain. These models operate on fixed-point data and are

optimized for HDL code generation. The algorithms in these models are verified against the golden and reference design scripts, and have been tested on boards to ensure that they decode over-the-air waveforms. They are ready for integration into your own designs and deploying to boards.

- The "NR HDL Cell Search" on page 5-88 example demonstrates a 5G cell search Simulink subsystem that uses the same algorithm as the MATLAB reference.
- The "NR HDL MIB Recovery" on page 5-56 example builds on the cell search example and adds a broadcast channel decoding and MIB recovery subsystem.
- The "Hardware Accelerators for NR SIB1 Recovery" on page 5-23 example shows the SIB1 grid recovery, CORESET0 decoding, and LDPC decoding sections of the SIB1 recovery algorithm implemented for hardware.
- The "NR HDL SIB1 Recovery" on page 5-8 example builds on the MIB recovery example and integrates the SIB1 hardware accelerators to form a complete SIB1 recovery system.
- The "NR HDL SIB1 Recovery for FR2" on page 5-44 example shows cell search, MIB, and SIB1 recovery models that are extended to support FR2.

The block diagram shows the 5G NR downlink receiver algorithm as implemented for hardware. The algorithm detects, demodulates, and decodes 5G NR synchronization signal blocks (SSBs) and recovers SIB1. It is a hardware-friendly version of the corresponding steps in the "NR Cell Search and MIB and SIB1 Recovery" (5G Toolbox) example. At the top level, the algorithm consists of a search controller, an SSB detector, an SSB decoder, SIB1 grid demodulator, and SIB1 decoder. The SIB1 decoder includes PDCCH and PDSCH decoder algorithms that use hardware polar decoder and LDPC decoder blocks. The shaded areas show which examples implement which parts of the downlink receiver.
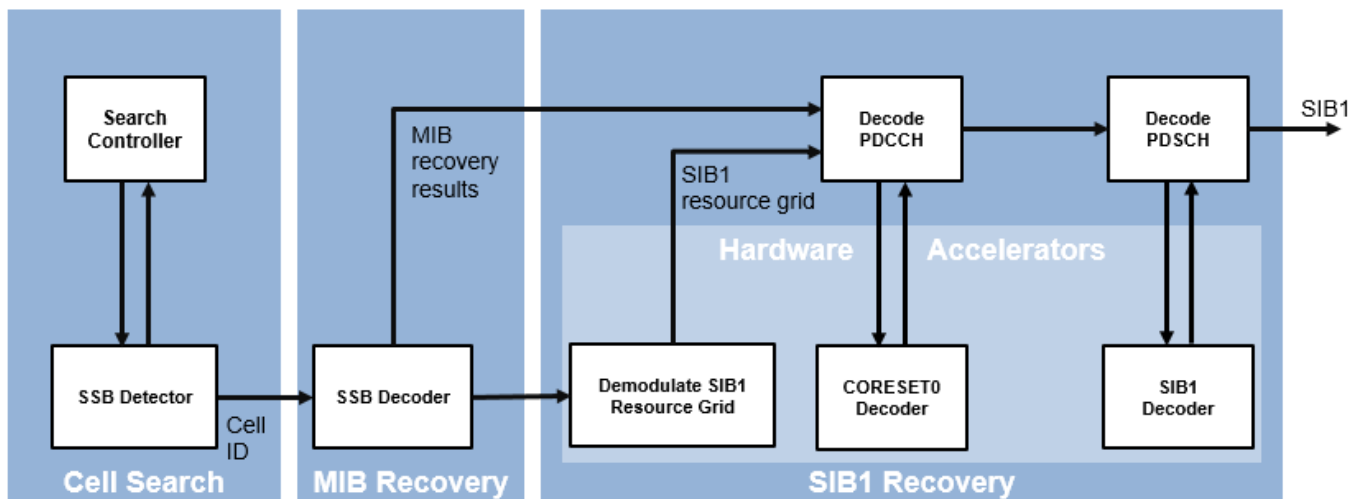
4   The *Simulink SoC Deployment Model* step consists of the "Deploy NR HDL Reference Applications on FPGAs and SoCs" on page 5-105 examples, which build on the fixed-point implementation models and use hardware support packages to deploy the algorithms on hardware. For the success rate of the algorithm running on hardware, see "NR HDL Receiver Performance" on page 5-5.

For a general description of how MATLAB and Simulink can be used together to develop deployable models, see "Wireless Communications Design for ASICs, FPGAs, and SoCs".

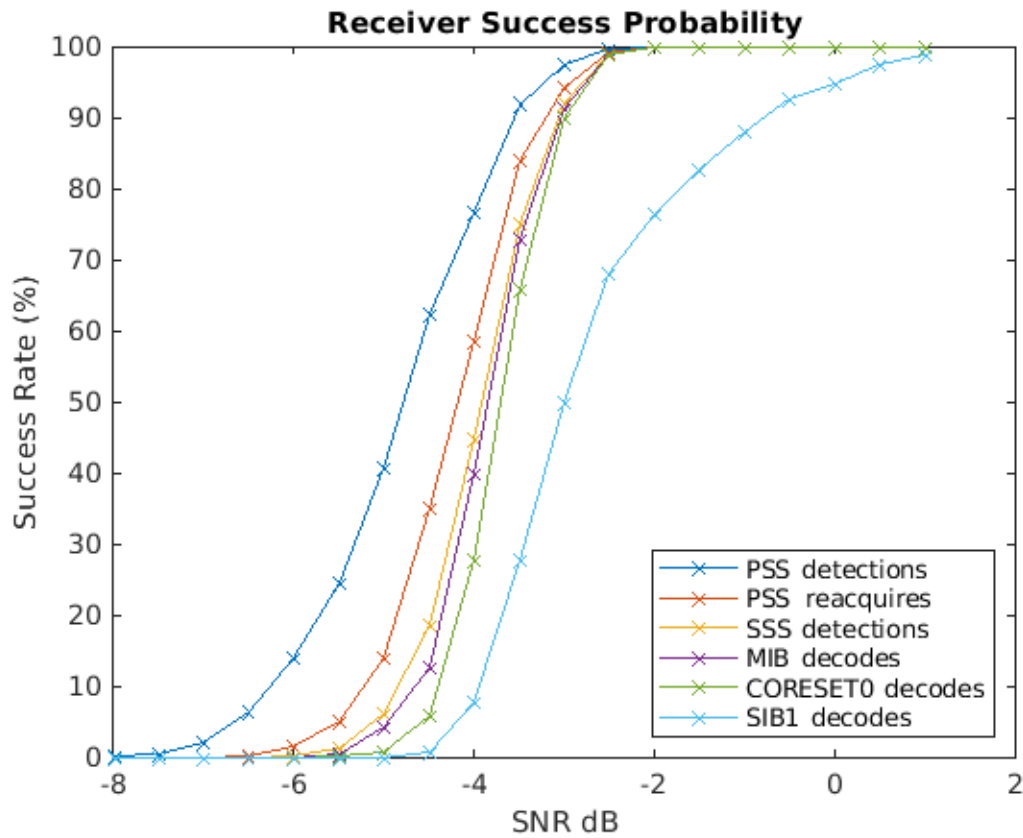## See Also

Related Examples
- "NR Cell Search and MIB and SIB1 Recovery" (5G Toolbox)
- "NR HDL Downlink Receiver MATLAB Reference" on page 5-68
- "NR HDL Cell Search" on page 5-88
- "NR HDL MIB Recovery" on page 5-56
- "Hardware Accelerators for NR SIB1 Recovery" on page 5-23
- "NR HDL SIB1 Recovery" on page 5-8
- "NR HDL SIB1 Recovery for FR2" on page 5-44
- "Deploy NR HDL Reference Applications on FPGAs and SoCs" on page 5-105
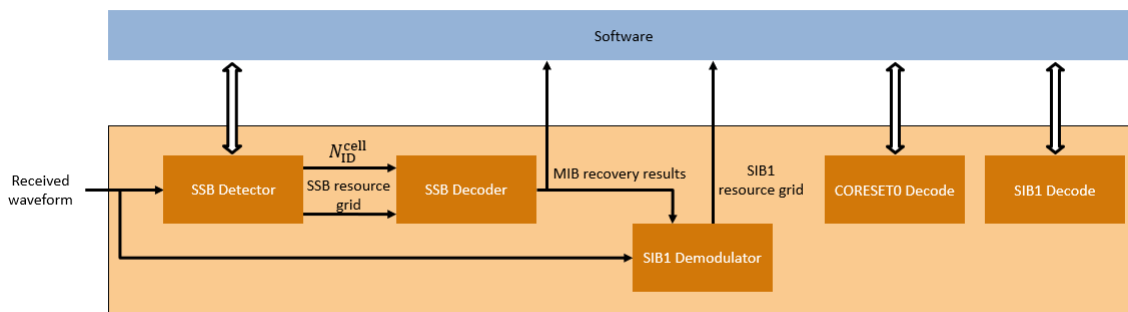
NR HDL Receiver Performance

Wireless HDL Toolbox 5G NR reference applications implement and verify parts of a 5G NR downlink receiver. For information about the reference applications, see "NR HDL Reference Applications Overview" on page 5-2. This page shows the performance of the stages of the receiver over a range of signal conditions for both the MATLAB reference design and the hardware implementation running on a board. For both MATLAB and hardware performance characterization, the design runs 1000 times at each SNR level, in 0.5 dB increments from –8 dB to 1 dB. The table shows the waveform and receiver configuration parameters.

Parameter	Setting
SSB Pattern	Case C
SCS SSB	30
SCS Common	30
PDCCH Config SIB1	164
Lmax	8
kSSB	0
Minimum channel bandwidth	5

The first graph shows the success probability of the MATLAB reference design for the receiver. This reference design models the data flow and sample rate used in the hardware implementation and operates on vectors and matrices of floating-point data samples. For more information, see "NR HDL Downlink Receiver MATLAB Reference" on page 5-68. The plot shows the success rate at each receiver stage for a range of signal-to-noise ratios. The input waveforms are created by using 5G Toolbox™ functions to generate a 5G FR1 waveform that contains one SSB burst and the corresponding SIB1 transmission.

The second graph shows the success probability of the receiver when deployed on hardware, for the same range of signal-to-noise ratios. The receiver described in the "5G NR SIB1 Recovery for FR1 and FR2 Using Xilinx RFSoC Device" (SoC Blockset Support Package for Xilinx Devices) example is deployed to a Xilinx Zynq UltraScale+™ RFSoC ZCU111 board. The characterization script writes the generated input waveforms to DDR memory on the board, and the deployed receiver reads the input signals from the memory.
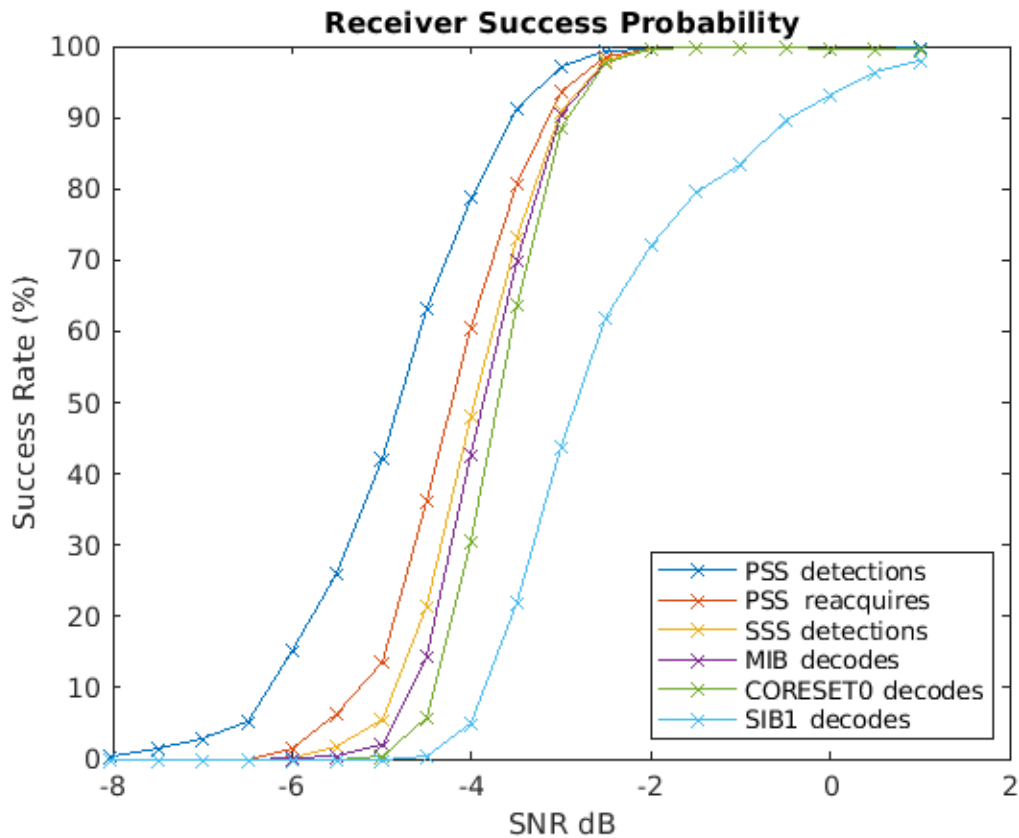
Receiver Success Probability

See Also

Related Examples

- "NR HDL Downlink Receiver MATLAB Reference" on page 5-68
- "NR HDL Cell Search" on page 5-88
- "NR HDL MIB Recovery" on page 5-56
- "NR HDL SIB1 Recovery" on page 5-8
- "Deploy NR HDL Reference Applications on FPGAs and SoCs" on page 5-105

More About

- "NR HDL Reference Applications Overview" on page 5-2

NR HDL SIB1 Recovery

This example shows how to design a 5G NR system information block type 1 (SIB1) recovery model optimized for HDL code generation and hardware implementation.

Introduction

SIB1 recovery requires cell search, master information block (MIB) decoding, recovery of the SIB1 grid (the area of the resource grid containing CORESET0 and SIB1), and decoding of the CORESET0 PDCCH and SIB1 PDSCH from the SIB1 grid. The process of Cell Search and MIB recovery are described in the "NR HDL Cell Search" on page 5-88 and "NR HDL MIB Recovery" on page 5-56 examples respectively. The additonal models used to implement SIB1 grid recovery, CORESET0 decoding, and SIB1 decoding are described in the "Hardware Accelerators for NR SIB1 Recovery" on page 5-23 example. This example focuses on the SIB1 Recovery Simulink model and uses the MATLAB reference to generate test input and verify the behavior of the model.

The Simulink® models described in this example are fixed-point HDL optimized implementations of SIB1 recovery for 5G NR frequency range 1 (FR1). This example is one of a related set, for more information see "NR HDL Reference Applications Overview" on page 5-2.

File Structure

This example uses these files.

Simulink models

- `nrhdlSIB1Recovery.slx`: This Simulink model combines the processing of the SSB detector, SSB decoder, SIB1 demodulator, CORESET0 decoder, and SIB1 decoder into an integrated model illustrating the complete SIB1 grid recovery process. This model references the `nrhdlDDCFR1Core`, `nrhdlSSBDetectionFR1Core`, `nrhdlSSBDecodingCore`, `nrhdlPolarDecodingChainCore`, `nrhdlSIB1DemodulationCore`, `nrhdlCORESET0DecodingCore`, and `nrhdlLDPCDecodingChainCore` models.

- `nrhdlDDCFR1Core.slx`: This model implements a DDC to create sample streams for SIB1 and SSBs.

- `nrhdlSSBDetectionFR1Core.slx`: This model implements the SSB detection algorithm.

- `nrhdlSSBDecodingCore.slx`: This model implements the SSB decoding algorithm.

- `nrhdlPolarDecodingChainCore.slx` : This model implements the common polar decoding chain.
- `nrhdlSIB1DemodulationCore.slx`: This model implements the SIB1 Demodulation algorithm.
- `nrhdlCORESET0DecodingCore.slx` : This model implements the CORESET0 decoding algorithm.
- `nrhdlLDPCDecodingChainCore.slx` : This model implements the SIB1 LDPC decoding algorithm.
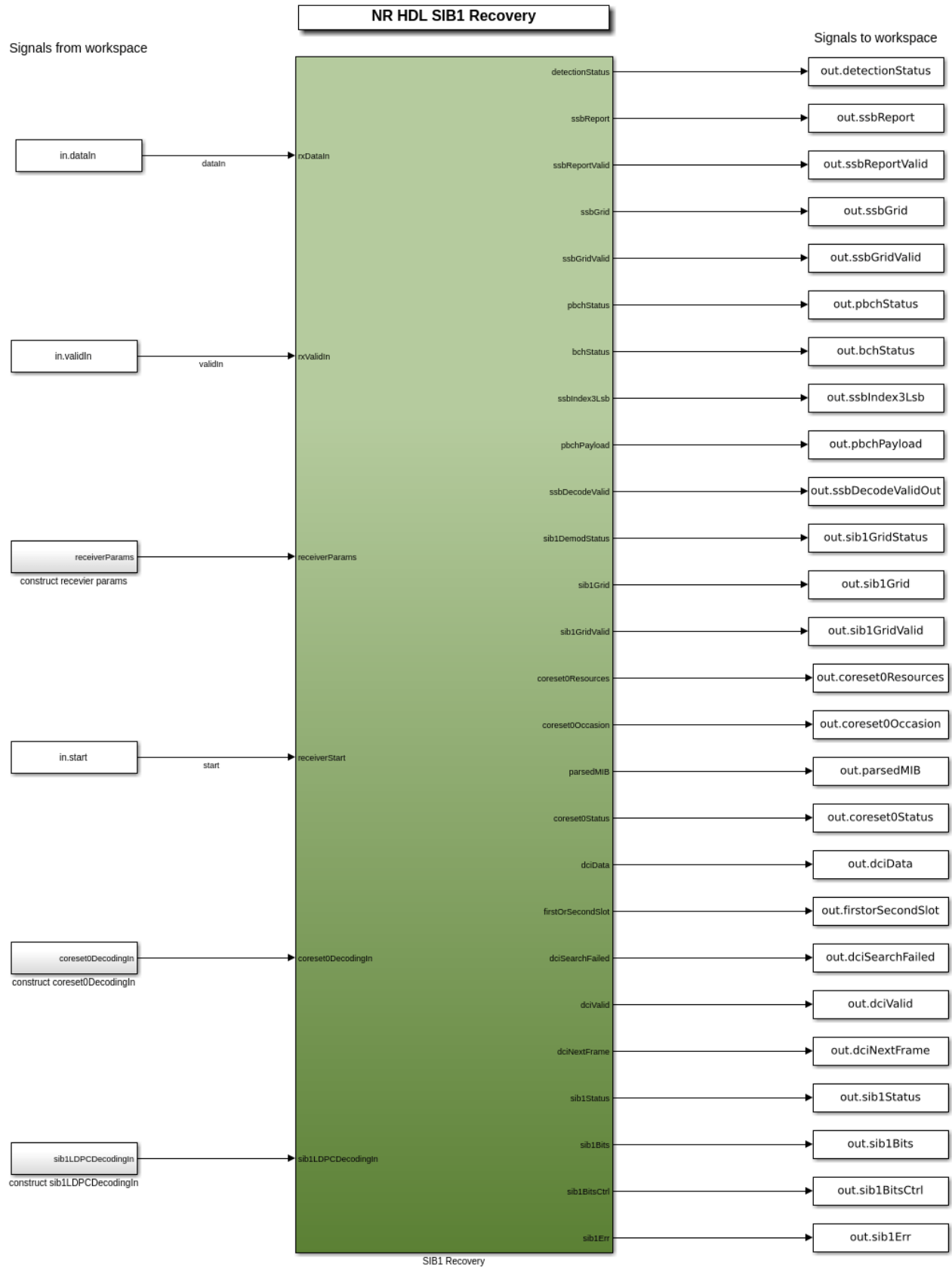
Simulink data dictionary

- `nrhdlReceiverData.sldd`: This Simulink data dictionary contains bus objects that define the buses contained in the example models.

MATLAB code

- `runSIB1RecoveryModel.m`: This script uses the MATLAB reference to perform the search mode of the SSB detection algorithm, then runs the `nrhdlSIB1Recovery` Simulink model to demodulate and decode the SSB, and then demodulate the SIB1 grid. The script performs CORESET0 and SIB1 decoding using either MATLAB code designed for embedded software or the hardware accelerators in the `nrhdlSIB1Recovery` model.
- `nrhdlexamples`: Package containing the MATLAB reference code and utility functions for verifying the implementation models.
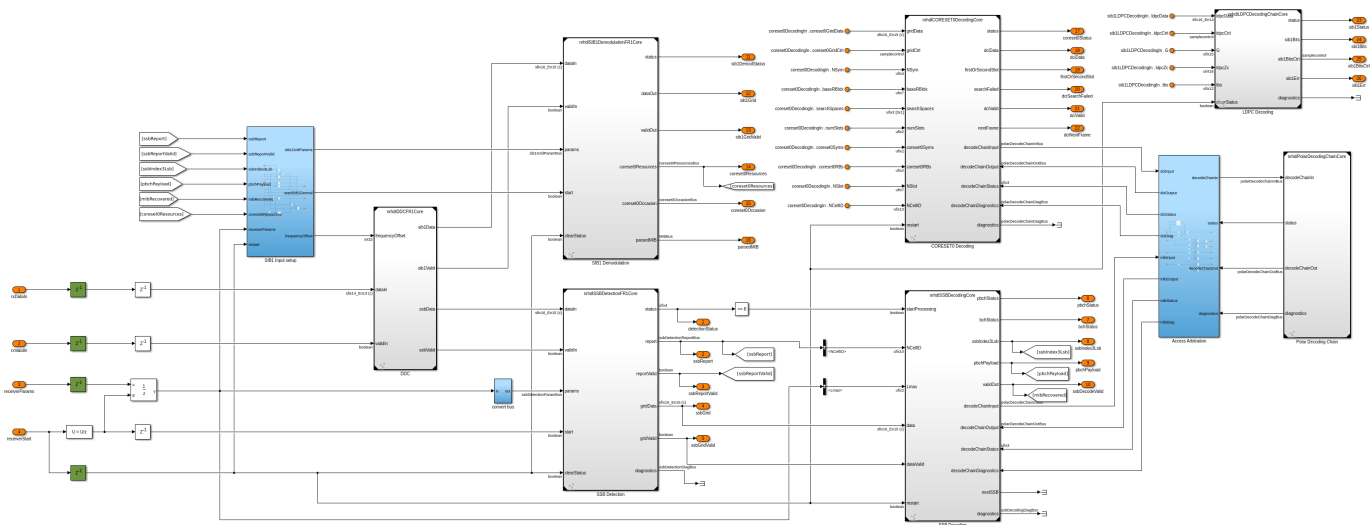
NR HDL SIB1 Recovery

This figure shows the `nrhdlSIB1Recovery` model. The top level of the model reads the signals from the MATLAB base workspace, passes them to the SIB1 Recovery subsystem, and writes the outputs back to the workspace. The model implements SIB1 recovery through a set of hardware accelerators which are controlled from software when deployed to an SoC device. The design operates on a baseband 5G waveform and performs initial access up to the decoding of the SIB1.

NR HDL SIB1 Recovery

Signals from workspace

Signals to workspace

Input	Block	Output
in.dataIn → dataIn	rxDataIn	detectionStatus → out.detectionStatus
		ssbReport → out.ssbReport
		ssbReportValid → out.ssbReportValid
		ssbGrid → out.ssbGrid
		ssbGridValid → out.ssbGridValid
		pbchStatus → out.pbchStatus
in.validIn → validIn	rxValidIn	bchStatus → out.bchStatus
		ssbIndex3Lsb → out.ssbIndex3Lsb
		pbchPayload → out.pbchPayload
		ssbDecodeValid → out.ssbDecodeValidOut
receiverParams	receiverParams	sib1DemodStatus → out.sib1GridStatus
construct recevier params		sib1Grid → out.sib1Grid
		sib1GridValid → out.sib1GridValid
		coreset0Resources → out.coreset0Resources
		coreset0Occasion → out.coreset0Occasion
in.start → start	receiverStart	parsedMIB → out.parsedMIB
		coreset0Status → out.coreset0Status
		dciData → out.dciData
		firstOrSecondSlot → out.firstorSecondSlot
coreset0DecodingIn	coreset0DecodingIn	dciSearchFailed → out.dciSearchFailed
construct coreset0DecodingIn		dciValid → out.dciValid
		dciNextFrame → out.dciNextFrame
		sib1Status → out.sib1Status
sib1LDPCDecodingIn	sib1LDPCDecodingIn	sib1Bits → out.sib1Bits
construct sib1LDPCDecodingIn		sib1BitsCtrl → out.sib1BitsCtrl
		sib1Err → out.sib1Err

SIB1 Recovery

SIB1 Recovery Subsystem

The `SIB1 Recovery` subsystem references models and combines them to create the full SIB1 recovery design. The appendix of this example contains a full decription of the subsystem interface. The subsystem can be operated in four modes, the software control loop co-ordinates the process to setup inputs and monitor the outputs for each stage.

1 Search: This operation searches for SSBs at a given frequency offset and subcarrier spacing. it performs three correlations, one for each PSS sequence. By running repeated search operations a subcarrier spacing sweep and coarse frequency search algorithm can be performed in software to create a list of the SSBs at a selected carrier frequency.

2 Demodulate: This operation reacquires and OFDM demodulates a single SSB selected from the those found during the search step. Each detected SSB has a unique timing reference and PSS sequence so can be reacquired on a repeat transmission. Once the SSB is demodulated the SSB is decoded to obtain the MIB. If *sib1En* is set and a SIB1 transmission is scheduled the SIB1 grid corresponding to the reacquired SSB will be OFDM demodulated and output to the software.

3 CORESET0 Decode: This operation decodes CORESET0 to recover the SIB1 DCI by performing a blind search across each search space and monitored slot. The algorithm operates on data extracted from the SIB1 grid recovered in the previous step. The process of extracting this data is performed in software.

4 SIB1 Decode: This operation performs LDPC decoding, code block desegmentation, and CRC decoding to recover the final SIB1 payload. The input data is extracted from the SIB1 grid in software using the DCI from the previous step to select the allocated symbols.

More information on each model referenced by the SIB1 Recovery subsystem can be found in these examples.

The "NR HDL Cell Search" on page 5-88 example details:

- `nrhdlDDCFR1Core`
- `nrhdlSSBDetectionFR1Core`

The "NR HDL MIB Recovery" on page 5-56 example details:

- `nrhdlSSBDecodingCore`
- `nrhdlPolarDecodingChainCore`

The "Hardware Accelerators for NR SIB1 Recovery" on page 5-23 example details:

- `nrhdlSIB1DemodulationCore`
- `nrhdlCORESET0DecodingCore`
- `nrhdlLDPCDecodingChainCore`

SIB1 Recovery Simulation Setup

The block diagram shows the simulation setup implemented by this example. The orange blocks highlight the comparison points between the MATLAB reference and the Simulink HDL implementation. The runSIB1RecoveryModelFR2 script runs the simulation, the MATLAB code represents the software control algorithm and the Simulink simulations perform the FPGA processing. 5G Toolbox™ functions are used to generate a test waveform. MATLAB reference code is used to perform the SSB search stage in place of running the Simulink simulation. The MATLAB reference provides equivalent results and improves simulation speed because it runs faster than the Simulink simulation. The results of the MATLAB SSB search is passed to both MATLAB and Simulink implementations of SIB1 recovery, and the output grids are directly compared. The Simulink SIB1 grid is decoded by one of two methods. The default option uses the `nrhdlSIB1Recovery` model to simulate the hardware accelerators for CORESET0 and SIB1 decoding. The second option uses a MATLAB only decode algorithm. When the design is deployed to an SoC the first option reduces the computations performed by the embedded processor by offloading the calculations to the FPGA. The second option performs all processing in software allowing for the algorithm to be easily modified and updated without rebuilding the FPGA bitstream.

SIB1 Recovery Simulation

Use the `runSIB1RecoveryModel` script to run a SIB1 recovery simulation. The script displays its progress at the MATLAB command prompt, and produces plots of inputs and outputs for analysis. The script also supports multiple simulation cases. The full set of cases, and their parameters, are shown.

Simulation Case	SSB Pattern	Subcarrier Spacing Common	PDCCH Config SIB1	SNR dB
"SimCase 1"	"Case C"	30	164	50
"SimCase 2"	"Case B"	15	100	6
"SimCase 3"	"Case A"	30	4	20
"SimCase 4"	"Case A"	15	84	7

This example shows the results of running "SimCase 1". The resource grids produced by MATLAB and Simulink are displayed along with their relative mean squared error (MSE). This comparison verifies that the Simulink implementation closely matches the MATLAB reference. The grid plots are labelled to highlight the decoded PDCCH and PDSCH. The final stage of the script decodes CORESET0, displays the DCIs, and decodes SIB1. The result of the SIB1 decode is displayed, and the SIB1 bits from MATLAB and Simulink are comapred to verify that they match.

runSIB1RecoveryModel;

```
Generating test waveform.
Searching for SSBs using MATLAB reference.
    NCellID2    timingOffset    pssCorrelation    pssEnergy    frequencyOffset

    _____    _____    _____    _____    _____

       0             4416          0.81822          0.90289         5059
       0            17568          0.65152          0.71764         4996
       0            35136           1.6417           1.8089         5019
       0            48288           1.2927           1.4275         5036
       0            65856           5.1698           5.6955         4943
       0            79008           1.1552           1.2723         5005
       0            96576            2.065           2.2723         5015
       0         1.0973e+05         1.0295           1.1339         4993


Recover the SIB1 grid using MATLAB reference.
Decoding the SSB using the MATLAB reference.
Recovering the SIB1 grid using the MATLAB reference.
Recover the SIB1 grid using Simulink model.
Running nrhdlSIB1Recovery.slx
### Starting serial model reference simulation build.
### Model reference simulation target for nrhdlCORESET0DecodingCore is up to date.
### Model reference simulation target for nrhdlDDCFR1Core is up to date.
### Model reference simulation target for nrhdlLDPCDecodingChainCore is up to date.
### Model reference simulation target for nrhdlPolarDecodingChainCore is up to date.
### Model reference simulation target for nrhdlSIB1DemodulationFR1Core is up to date.
### Model reference simulation target for nrhdlSSBDecodingCore is up to date.
### Model reference simulation target for nrhdlSSBDetectionFR1Core is up to date.


Build Summary

0 of 7 models built (7 models already up to date)
Build duration: 0h 0m 1.0194s
..........
MATLAB and Simulink grids relative MSE : -62.0132 dB

Extracting CORESET0 candidates from the SIB1 grid.
Decoding CORESET0 candidates using MATLAB reference.
Decoding CORESET0 candidates using Simulink.
Running nrhdlSIB1Recovery.slx
### Starting serial model reference simulation build.
```

```
### Model reference simulation target for nrhdlCORESET0DecodingCore is up to date.
### Model reference simulation target for nrhdlDDCFR1Core is up to date.
### Model reference simulation target for nrhdlLDPCDecodingChainCore is up to date.
### Model reference simulation target for nrhdlPolarDecodingChainCore is up to date.
### Model reference simulation target for nrhdlSIB1DemodulationFR1Core is up to date.
### Model reference simulation target for nrhdlSSBDecodingCore is up to date.
### Model reference simulation target for nrhdlSSBDetectionFR1Core is up to date.

Build Summary

0 of 7 models built (7 models already up to date)
Build duration: 0h 0m 0.8094s
..........
DCI from MATLAB:
                      RIV: 528
                 TDDIndex: 0
     VRBToPRBInterleaving: 0
                ModCoding: 0
                       RV: 0
              SIIndicator: 0
                 Reserved: 0

DCI from Simulink:
                      RIV: 528
                 TDDIndex: 0
     VRBToPRBInterleaving: 0
                ModCoding: 0
                       RV: 0
              SIIndicator: 0
                 Reserved: 0

DCI successfully decoded from Simulink grid with hardware acceleration

Extracting LDPC codeword from the SIB1 grid.
Decoding SIB1 using MATLAB reference.
Decoding SIB1 using Simulink.
Running nrhdlSIB1Recovery.slx
### Starting serial model reference simulation build.
### Model reference simulation target for nrhdlCORESET0DecodingCore is up to date.
### Model reference simulation target for nrhdlDDCFR1Core is up to date.
### Model reference simulation target for nrhdlLDPCDecodingChainCore is up to date.
### Model reference simulation target for nrhdlPolarDecodingChainCore is up to date.
### Model reference simulation target for nrhdlSIB1DemodulationFR1Core is up to date.
### Model reference simulation target for nrhdlSSBDecodingCore is up to date.
### Model reference simulation target for nrhdlSSBDetectionFR1Core is up to date.

Build Summary

0 of 7 models built (7 models already up to date)
Build duration: 0h 0m 0.85057s
..........
SIB1 successfully decoded from Simulink grid with hardware acceleration
SIB1 bits from MATLAB and Simulink match
```

SIB1 Grid from MATLAB Reference

SIB1 Grid from Simulink
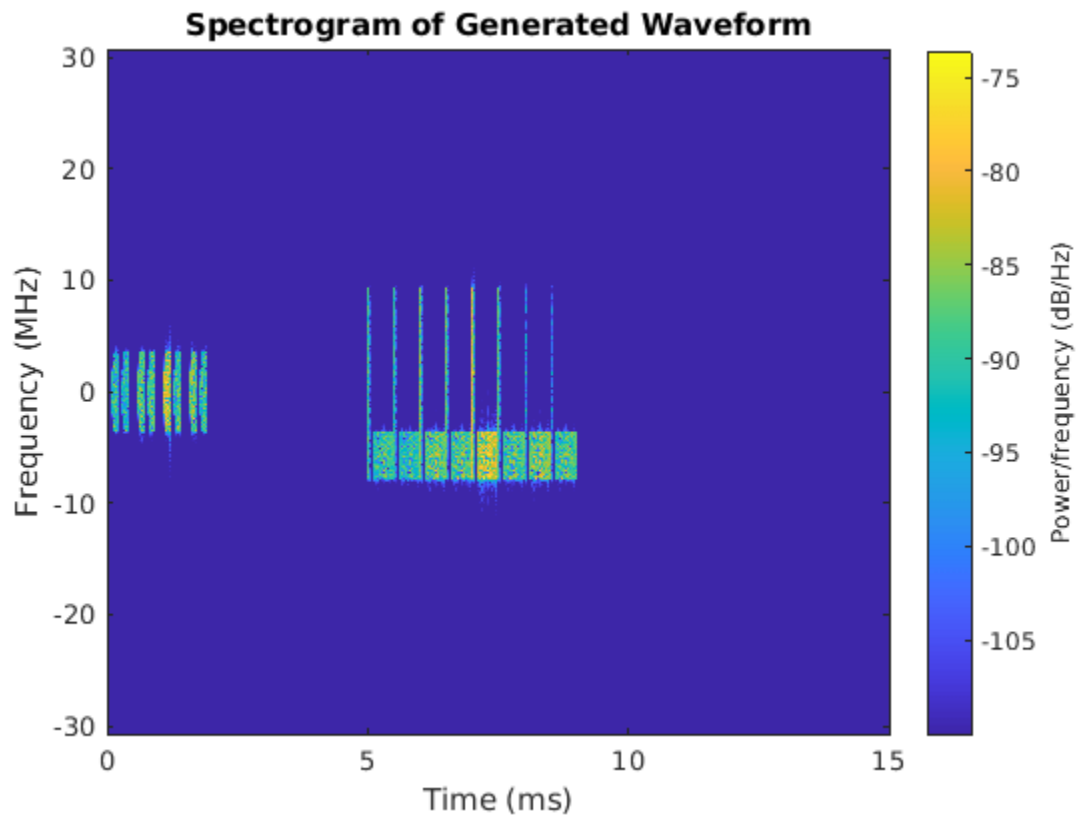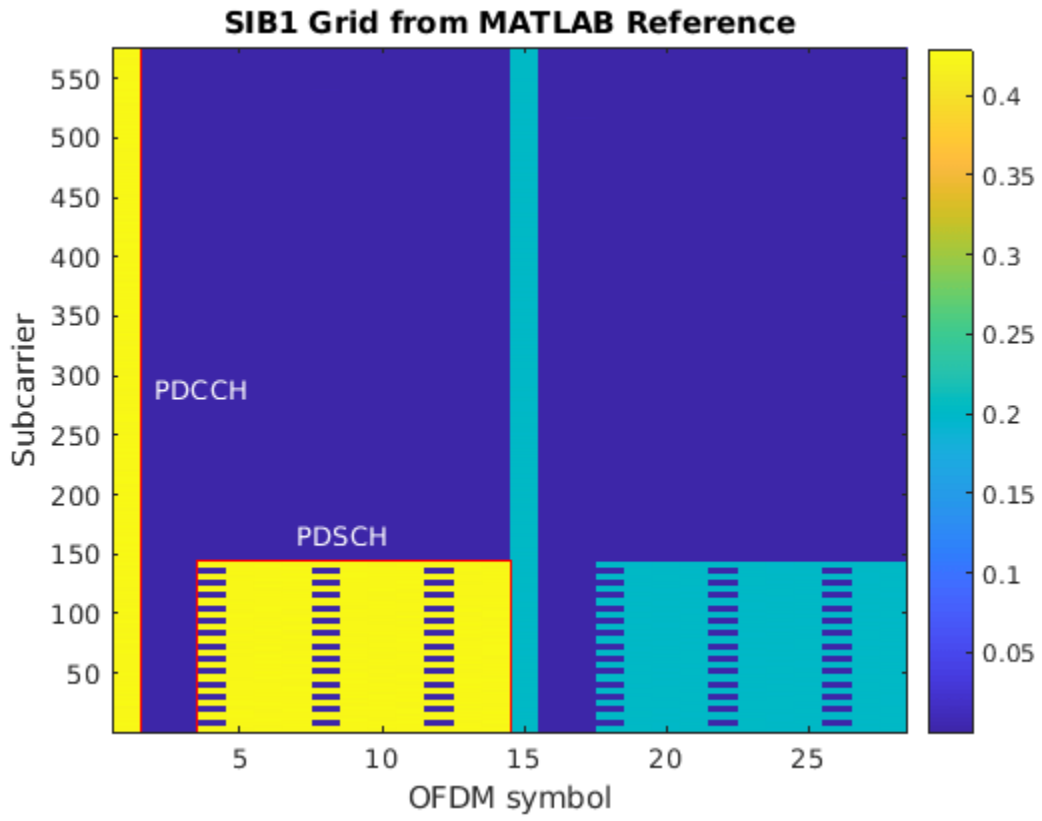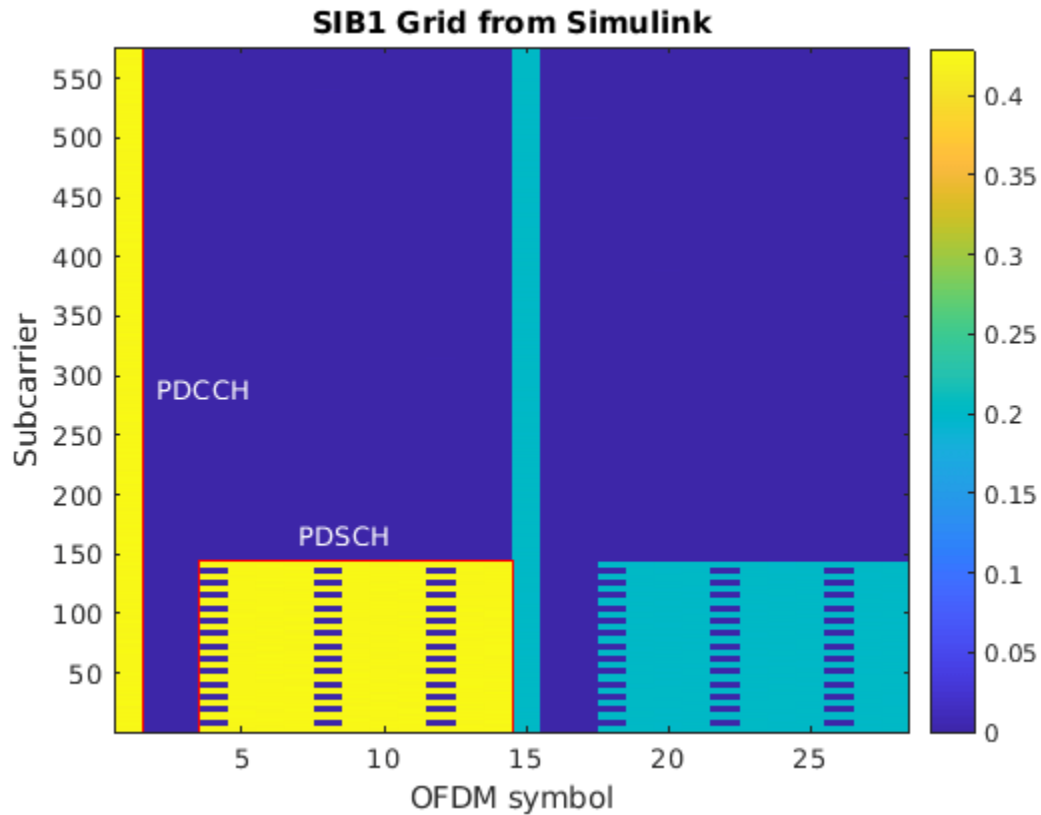
HDL Code Generation and Implementation Results

To generate the HDL code for this example, you must have the HDL Coder™ product. Use the `makehdl` and `makehdltb` commands to generate HDL code and an HDL test bench for `nrhdlSIB1Recovery/SIB1 Recovery` subsystems. The resulting HDL code was synthesized for a Xilinx® Zynq® UltraScale+ RFSoC ZCU111 evaluation board. The table shows the post place and route resource utilization results. The design meets timing with a clock frequency of 245.76 MHz.

Resource utilization for nrhdlSIB1dRecovery model:

Resource	Usage
Slice Registers	116600
Slice LUTs	79117
RAMB18	409
RAMB36	17
DSP48	275

To deploy the nrhdlSIB1Recovery model to a hardware platform and recover SIB1 from off the air signals, see the "Deploy NR HDL Reference Applications on FPGAs and SoCs" on page 5-105 example.

Appendix

SIB1 Recovery Interface

Inputs

- *dataIn*: 14-bit signed complex-valued signal, sampled at 61.44 Msps.
- *validIn*: 1-bit control signal to validate *dataIn*.
- *receiverParams*: Bus signal containing parameter values used for SSB search, demodulation, and SIB1 grid recovery.
- *receiverStart*: 1-bit control signal used to start a search or demodulation operation.
- *coreset0DecodingIn*: Bus signal containing the input data used for CORESET0 decoding.
- *sib1LDPCDecodingIn*: Bus signal containing the input data used for SIB1 LDPC decoding.

receiverParams Bus

- *frequencyOffset*: 32-bit signed value specifying the frequency offset to be corrected. This signal is connected to an NCO with a 32-bit accumulator. Use this equation to convert the value to Hz: *frequencyOffset_Hz* = *frequencyOffset* * 61.44e6 / 2^32.
- *subcarrierSpacing*: 2-bit unsigned value specifying the subcarrier spacing. Set this signal to 0 to select 15kHz, or 1 to select 30kHz.
- *mode*: 1-bit unsigned value specifying the operation mode. Set this signal to 0 for search mode, or 1 for demod mode.
- *timingOffset*: 21-bit unsigned value specifying the timing offset of the start of the SSB to be demodulated. Specify the timing offset in samples at 61.44 Msps, from 0 to 1228799. This parameter applies only for demod mode.
- *NCellID2*: 2-bit unsigned value specifying the PSS (0, 1, or 2) of the SSB to be demodulated. This parameter applies only for demod mode.
- *Lmax*: 2-bit unsigned number which indicates the maximum number of SSBs in a burst. A value of 0 indicates 4 SSBs, a value of 1 indicates 8 SSBs, a value of 2 indicates 64 SSBs.
- *sib1En*: 1-bit unsgined number which enables SIB1 grid recovery after a successfull MIB decode.
- *minChanBW*: 2-bit unsigned value specifying the minimum channel bandwidth. A value of 0 indicates 5 MHz, 1 indicates 10 MHz, and 2 indicates 40 MHz.
- *ssbPattern*: 2-bit unsigned value specifying the SSB pattern. A value of 0 indicates 'Case A', 1 indicates 'Case B', and 2 indicates 'Case C'.

coreset0DecodingIn Bus

- *gridDataIn*: 16-bit signed CORESET0 candidate OFDM grid data.
- *gridCtrlIn*: Sample control bus signal to validate *gridDataIn*.
- *NSym*: 4-bit OFDM symbol number for the current resource element group (REG).
- *baseRBIdx*: 7-bit base CORESET0 resource block index for the current REG.
- *searchSpaces*: 3-bit unsigned vector of length 3 indicating the number of search spaces at aggregation levels 4, 8, and 16.
- *coreset0Syms*: 2-bit unsigned value that is the number of OFDM symbols CORESET0 spans.
- *coreset0RBs*: 2-bit unsigned value specifying the number of resource blocks. A value of 0 indicates 24, 1 indicates 48, and 2 indicates 96.
- *NSlot*: 7-bit unsigned value that specifies the slot number for the first monitored CORESET0 slot.
- *NCellID*: 10-bit unsigned value that is the cell ID of the demodulated SSB.

sib1LDPCDecodingIn Bus

- *ldpcDta*: 16-bit signed LDPC codeword LLR data.
- *ldpcCtrl*: Sample control bus for validating *ldpcData*.
- *G*: 15-bit length of the input codeword.
- *ldpcZc*: 16-bit unsigned value indicating the lifting size used for the LDPC codeword.
- *tbs*: 12-bit unsigned value indicating the length of the decoded output data.

Outputs

- *detectionStatus*: 4-bit unsigned value that indicates the progress of the current SSB detection operation. See the next section for the possible values of this signal.
- *ssbReport*: Bus of type ssbDetectionReportBus.
- *reportValid*: 1-bit control signal which validates the *ssbReport* output.
- *ssbGrid*: 16-bit signed complex-values that are the SSB resource grid data.
- *ssbGridValid*: 1-bit control signal that validates the *ssbGrid* output.
- *pbchStatus*: 2-bit unsigned value indicating the progress of the PBCH decoding operation. See below for more information on the possible values of this signal.
- *bchStatus*: 3-bit unsigned value indicating the progress of the BCH decoding operation. See below for more information on the possible values of this signal.
- *ssbIndex3Lsb*: 3-bit unsigned value that is the 3 least significant bits of the SSB index calculated by the DMRS search process and Lmax.
- *pbchPayload*: 32-bit unsigned value that contains the MIB and additional PBCH timing data.
- *ssbDecodeValid*: 1-bit control signal to validate *ssbIndex3Lsb* and *pbchPayload*.
- *sib1DemodStatus*: 2-bit unsigned value indicating the progress of the SIB1 grid demodulation operation.
- *sib1Grid*: 16-bit signed complex-valued SIB1 resource grid data.
- *sib1GridValid*: 1-bit control signal that validates the *sib1Grid* output.
- *coreset0Resources*: Bus of type coreset0ResourcesBus.
- *coreset0Occasion*: Bus of type coreset0OccasionBus.
- *parsedMIB*: Bus of type MIBBus.
- *coreset0Status*: 3-bit unsigned value indicating the progress of the CORESET0 decoding process.
- *dciData*: 41-bit unsigned data that contains the final decoded DCI.
- *firstOrSecondSlot*: 1-bit value indicating if the decoded DCI was found in the first (0) or second (1) monitored slot.
- *dciSearchFailed*: 1-bit value indicating that the CORESET0 DCI search failed.
- *dciValid*: 1-bit value indicating the search is complete.
- *dciNextFrame*: 1-bit signal to provide back pressure to signal when the next candidate can be input.
- *sib1Status*: 3-bit unsigned value indicating the progress of the SIB1 decoding process.
- *sib1Bits*: 1-bit data that is the final decoded SIB1 payload.
- *sib1BitsCtrl*: Sample control bus for validating *sib1Bits*
- *sib1Err*: 1-bit value indicating if the SIB1 CRC failed.

ssbDetectionReportBus

- *NCellID2*: 2-bit unsigned value that is the PSS (0, 1 or 2) of the detected SSB.
- *timingOffset*: 21-bit unsigned value that is the timing offset of the detected SSB. The timing offset is in samples at 61.44 Msp from 0 to 1228799.
- *frequencyOffset*: 32-bit signed value that is the frequency offset of the detected SSB. This signal has the same units as the *frequencyOffset* input.
- *pssCorrelation*: 32-bit unsigned value that is the strength of the PSS correlation.
- *pssThreshold*: 32-bit unsigned value that is the threshold value when PSS was detected.
- *sssCorrelation*: 32-bit unsigned value that is the SSS correlation strength. This signal is returned only in demod mode.
- *sssThreshold*: 32-bit unsigned value that is the SSS threshold. This value is returned only in demod mode.
- *NCellID*: 10-bit unsigned value that is the cell ID of the demodulated SSB. This value is returned only in demod mode.

coreset0ResourcesBus

- *resourceBlocks*: 2-bit unsigned value specifying the number of resource blocks. A value of 0 indicates 24, 1 indicates 48, and 2 indicates 96.
- *ofdmSymbols*: 2-bit unsigned value that is the number of OFDM symbols CORESET0 spans.
- *frequencyOffset*: 32-bit signed value specifying the relative frequency offset from the SSB to CORESET0. This signal is connected to an NCO with a 32-bit accumulator. Use this equation to convert the value to Hz: $frequencyOffset\_Hz = frequencyOffset * 61.44e6 / 2^{32}$.
- *muxPattern*: 2-bit unsigned value specifying the CORESET0 multiplexing pattern.

coreset0OccasionBus

- *slotOffset*: 5-bit unsigned value that is the slot offset from the even frame head to the first monitored slot.
- *firstSymbol*: 3-bit unsigned value specifying the first occupied OFDM symbol in the slot.

MIBBus

- *sfn*: 10-bit unsigned value that is the system frame number (SFN).
- *scsCommon*: 1-bit unsigned value specifying the common subcarrier spacing. A value of 0 indicates 15 kHz, and 1 indicates 30 kHz.
- *Kssb*: 5-bit unsigned value that is the offset between the SSB and the overall resource block grid.
- *drmsTypeAPos*: 1-bit unsigned value specifying the position of the DMRS symbol for PDSCH allocation type A, where 0 represents position 2 and 1 indicates position 3.
- *pdcchConfigSIB1*: 8-bit unsigned value containing the configuration for CORESET0
- *cellBarred*: 1-bit value indicating whether the cell is barred.
- *intraFreqReselection*: 1-bit value indicating whether intra frequency reselection is allowed.
- *hrf*: 1-bit value that is the half frame bit.
- *ssbIdx*: 6-bit value that is the index of the SSB.

Detection Status Signal States

- `0`: Idle -- Initial state. Waiting for first start pulse.

- 1: Search mode -- Searching for PSS.
- 2: Search mode -- Operation complete, no PSS found.
- 3: Search mode -- Operation complete, found one or more PSSs.
- 4: Demod mode -- Waiting for specified PSS timing offset.
- 5: Demod mode -- Operation complete, PSS not found.
- 6: Demod mode -- Found specified PSS. Demodulating the resource grid and looking for SSS.
- 7: Demod mode -- Operation complete, no SSS found. Returned demodulated resource grid.
- 8: Demod mode -- Operation complete, found SSS. Returned demodulated resource grid.

PBCH Status Signal States

- 0: Idle
- 1: Reading in data for SSB grid
- 2: Performing DMRS search
- 3: Performing PBCH symbol demodulation

BCH Status Signal States

- 0: Idle
- 1: Performing rate recovery
- 2: Performing polar decoding
- 3: CRC error
- 4: CRC pass, MIB detected

SIB1 Demod Status Signal States

- 0: Initial state. Waiting for start pulse.
- 1: Waiting for the CORESET0 timing occasion.
- 2: OFDM demodulating and outputting the SIB1 grid data.

CORESET0 Decoding Status Signal States

- 0: Initial state. Waiting for start pulse.
- 1: Performing channel estimation, equalization, symbol demodulation and descrambling.
- 2: Performing polar rate recovery.
- 3: Performing polar and CRC decoding.
- 4: Candidate decode failed, waiting for next attempt.
- 5: Decoded all candidates with no successes.
- 6: Successfully decoded the DCI from a candidate.

SIB1 Decoding Status Signal States

- 0: Initial state. Waiting for start pulse.
- 1: Performing LDPC rate recovery.
- 2: Performing LDPC decoding.

- 3: Performing CRC decoding.
- 4: Failed to decode SIB1.
- 5: Successfully decoded SIB1.

See Also

Related Examples
- "NR HDL Cell Search" on page 5-88
- "NR HDL MIB Recovery" on page 5-56

More About
- "NR HDL Receiver Performance" on page 5-5

Hardware Accelerators for NR SIB1 Recovery

This example shows the design of 5G SIB1 accelerators optimized for HDL code generation and hardware implementation.

Introduction

The Simulink® models described in this example are fixed-point HDL optimized implementations of hardware accelerators for SIB1 recovery for 5G NR. The example details three individual hardware accelerators that perform:

1 SIB1 grid recovery

2 CORESET0 decoding

3 SIB1 LDPC decoding

The SIB1 grid recovery algorithm is designed and optimized for frequency range 1 (FR1). The CORESET0 and SIB1 LDPC decoding algorithms support both frequency ranges. The "NR HDL SIB1 Recovery" on page 5-8 example shows how to integrate hardware accelerators with SSB detection and decoding designs to implement a complete SIB1 recovery model. A design supporting both FR1 and FR2 is implemented in the "NR HDL SIB1 Recovery for FR2" on page 5-44 example.

This example is one of a related set, for more information see "NR HDL Reference Applications Overview" on page 5-2.

File Structure

This example uses these files.

Simulink models

- `nrhdlSIB1Demodulation.slx`: This Simulink model references the `nrhdlDDCFR1Core` and `nrhdlSIB1DemodulationFR1Core` models to simulate the SIB1 grid demodulation step of SIB1 recovery.

- `nrhdlCORESET0Decoding.slx` : This model references the `nrhdlCORESET0DecodingCore` and `nrhdlPolarDecodingChainCore` models to simulate the CORESET0 decoding step of SIB1 recovery.

- `nrhdlSIB1LDPCDecoding.slx` : This model references the `nrhdlLDPCDecodingChainCore` model to simulate the SIB1 LDPC decoding step of SIB1 recovery.

- `nrhdlDDCFR1Core.slx`: This model implements a DDC to create sample streams for SIB1 and SSBs.

- `nrhdlSIB1DemodulationFR1Core.slx`: This model implements the SIB1 demodulation algorithm.

- `nrhdlCORESET0DecodingCore.slx` : This model implements the CORESET0 decoding algorithm.

- `nrhdlPolarDecodingChainCore.slx` : This model implements the common polar decoding chain.

- `nrhdlLDPCDecodingChainCore.slx` : This model implements the SIB1 LDPC decoding algorithm.

Simulink data dictionary

- `nrhdlReceiverData.sldd`: This Simulink data dictionary contains bus objects that define the buses contained in the example models.

MATLAB code

- `runSIB1AcceleratorModels.m`: This script uses the MATLAB reference to implement the MIB recovery algorithm, then runs the `nrhdlSIB1Demodulation`, `nrhdlCORESET0Decoding`, and `nrhdlSIB1LDPCDecoding` Simulink models. The script verifies the operation of the model using 5G Toolbox™ and the MATLAB reference code.
- `nrhdlexamples`: Package containing the MATLAB reference code and utility functions for verifying the implementation models.

NR HDL SIB1 Demodulation

This figure shows the `nrhdlSIB1Demodulation` model. The top level of the model reads the signals from the MATLAB base workspace, passes them to the SIB1 Demodulation subsystem, and writes the outputs back to the workspace.

```
models.DemodulationTop = 'nrhdlSIB1Demodulation';
open_system(models.DemodulationTop);
```

NR HDL SIB1 Demodulation

Signals from workspace

Signals to workspace
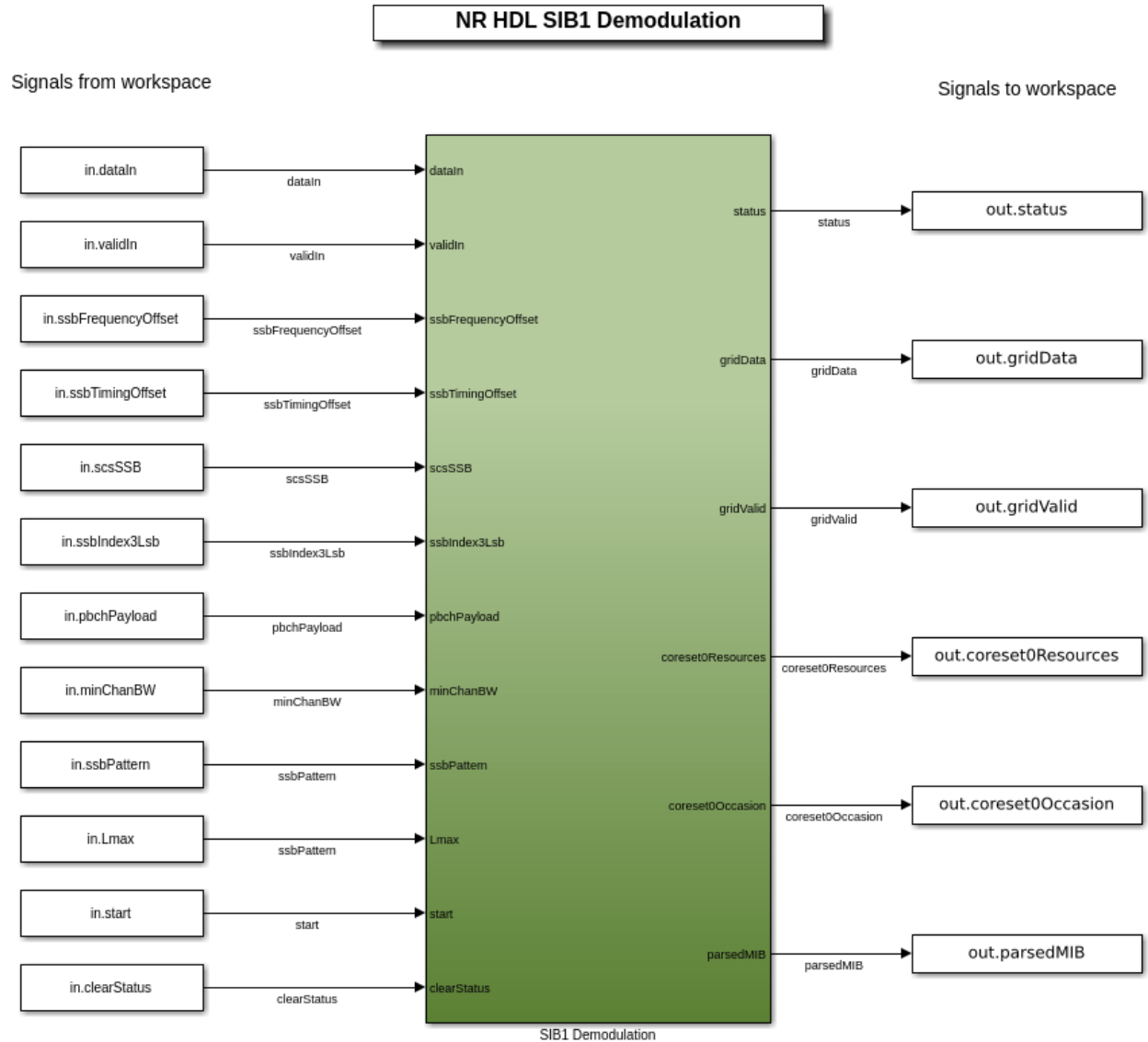
Copyright 2021-2022 The MathWorks, Inc.

SIB1 Demodulation Subsystem

The SIB1 Demodulation subsystem references the `nrhdlDDCFR1Core` and `nrhdlSIB1DemodulationFR1Core` models. The algorithm of the `nrhdlSIB1DemodulationFR1Core` model is described in the next section. For details about the `nrhdlDDCFR1Core` model see the "NR HDL Cell Search" on page 5-88 example. The output of the DDC is the input to the SIB1 Demodulation algorithm. The frequency offset applied to the DDC combines the SSB frequency offset estimation term, from a successful MIB recovery, with the SSB to SIB1 offset. The combined frequency offset centers CORESET0 in the received waveform.

```
set_param([models.DemodulationTop '/SIB1 Demodulation'],'Open','on');
```
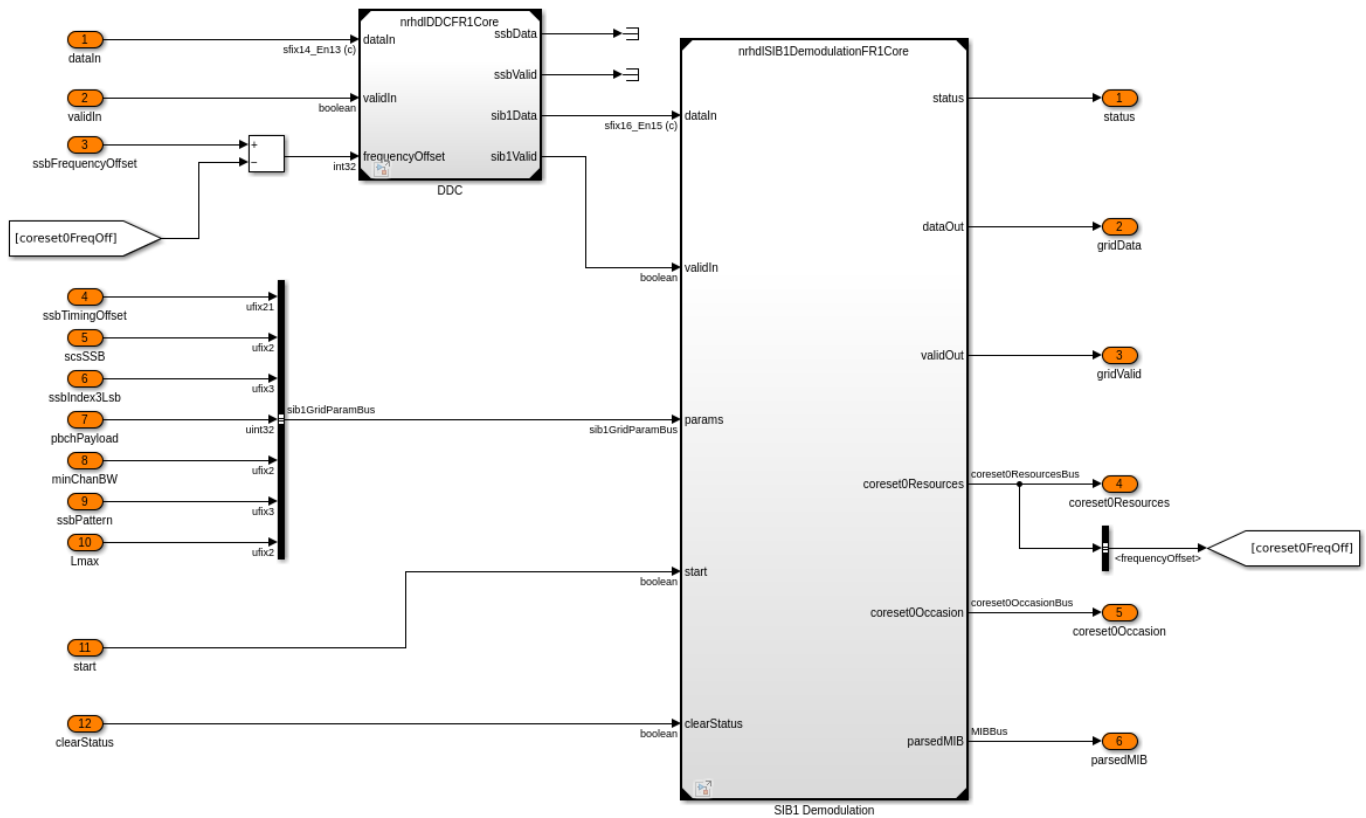
Inputs

- *dataIn*: 14-bit signed complex-valued signal, sampled at 61.44 Msps.
- *validIn*: 1-bit control signal to validate *dataIn*.
- *paramsIn*: Bus of type SIB1GridParamBus.
- *start*: 1-bit control signal used to start a SIB1 grid recovery operation.
- *clearStatus*: 1-bit control signal used to clear the state of the status signal.

SIB1GridParamBus

- *ssbTimingOffset*: 21-bit unsigned value that is the timing offset of the detected SSB. The timing offset is in samples at 61.44 Msps from 0 to 1228799.
- *scsSSB*: 2-bit unsigned value specifying the subcarrier spacing (SCS) of the detected SSB. Set this signal to 0 to select 15 kHz, or 1 to select 30 kHz.
- *ssbIndex3Lsb*: 3-bit unsigned value that is the 3 least significant bits of the SSB index.
- *pbchPayload*: 32-bit unsigned value that contains the MIB and additional PBCH timing data.
- *minChanBW*: 2-bit unsigned value specifying the minimum channel bandwidth. A value of 0 indicates 5 MHz, 1 indicates 10 MHz, and 2 indicates 40 MHz.
- *ssbPattern*: 2-bit unsigned value specifying the SSB pattern. A value of 0 indicates 'Case A', 1 indicates 'Case B', and 2 indicates 'Case C'.
- *Lmax*: 2-bit unsigned number which indicates the maximum number of SSBs in a burst. A value of 0 indicates 4 SSBs, a value of 1 indicates 8 SSBs, a value of 2 indicates 64 SSBs.

Outputs

- *status*: 2-bit unsigned value indicating the progress of the SIB1 demodulation operation.
- *dataOut*: 16-bit signed complex-valued SIB1 resource grid data. The algorithm outputs the 28 OFDM symbols of the SIB1 grid, one resource element (RE) per cycle.
- *validOut*: 1-bit control signal that validates the *dataOut* output.
- *coreset0Resources*: Bus of type coreset0ResourcesBus.
- *coreset0Occasion*: Bus of type coreset0OccasionBus.
- *parsedMIB*: Bus of type MIBBus.

coreset0ResourcesBus

- *resourceBlocks*: 2-bit unsigned value specifying the number of resource blocks. A value of 0 indicates 24, 1 indicates 48, and 2 indicates 96.
- *ofdmSymbols*: 2-bit unsigned value that is the number of OFDM symbols CORESET0 spans.
- *frequencyOffset*: 32-bit signed value specifying the relative frequency offset from the SSB to CORESET0. This signal is connected to an NCO with a 32-bit accumulator. Use this equation to convert the value to Hz: *frequencyOffset_Hz = frequencyOffset* * 61.44e6 / 2^32.
- *muxPattern*: 2-bit unsigned value specifying the CORESET0 multiplexing pattern.

coreset0OccasionBus

- *slotOffset*: 8-bit unsigned value that is the slot offset from the even frame head to the first monitored slot.
- *firstSymbol*: 4-bit unsigned value specifying the first occupied OFDM symbol in the slot.

MIBBus

- *sfn*: 10-bit unsigned value that is the system frame number (SFN).
- *scsCommon*: 1-bit unsigned value specifying the common subcarrier spacing. A value of 0 indicates 15 kHz, and 1 indicates 30 kHz.
- *Kssb*: 5-bit unsigned value that is the offset between the SSB and the overall resource block grid.
- *drmsTypeAPos*: 1-bit unsigned value specifying the position of the DMRS symbol for PDSCH allocation type A, where 0 represents position 2 and 1 indicates position 3.
- *pdcchConfigSIB1*: 8-bit unsigned value containing the configuration for CORESET0
- *cellBarred*: 1-bit value indicating whether the cell is barred.
- *intraFreqReselection*: 1-bit value indicating whether intra frequency reselection is allowed.
- *hrf*: 1-bit value that is the half frame bit.
- *ssbIdx*: 3-bit value that is the index of the SSB.

Status Signal States

- `0`: Initial state. Waiting for start pulse.
- `1`: Waiting for the CORESET0 timing occasion.
- `2`: OFDM demodulating and outputting the SIB1 grid data.

SIB1 Demodulation Model

This diagram shows the top level of the `nrhdlSIB1DemodulationFR1Core` model. The design operates on IQ data at 61.44 MHz sample rate and requires parameters from a successful MIB

recovery. The start signal begins a SIB1 demodulation operation, set this signal to 1 when all input ports on the paramsIn bus are valid. You must hold the values on paramsIn constant for the duration of the demodulation.

```
models.DemodulationCore = 'nrhdlSIB1DemodulationFR1Core';
load_system(models.DemodulationCore);
set_param(models.DemodulationCore,'SimulationCommand','Update');
set_param([models.DemodulationCore '/SIB1DemodulationCore'],'Open','on');
```
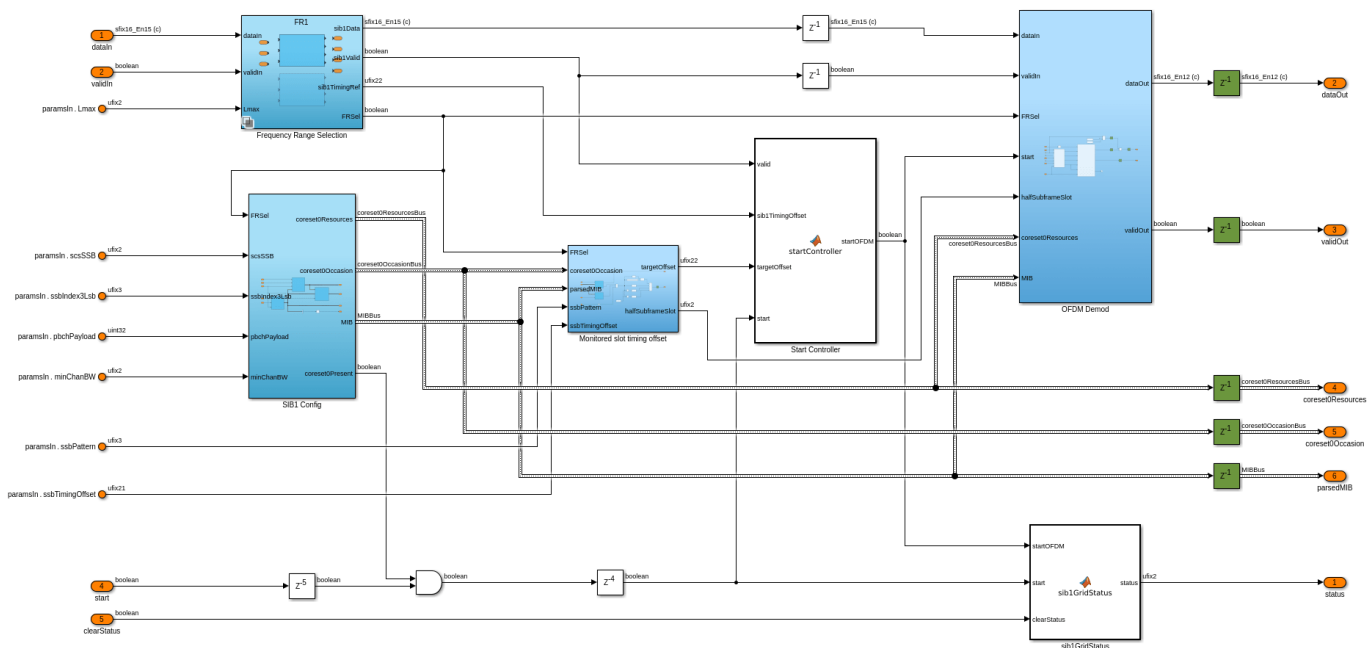

When the `OFDM Demod` subsystem recives a start signal, it OFDM demodulates the input data and outputs the SIB1 grid. The subsystem computes a variable size FFT, with configurable cyclic prefix length and number of guard subcarriers. The FFT size is selected depending on the subcarrier spacing of the SIB1 grid. For SCS 15 a 2048-point FFT is used, and for SCS 30 a 1024-point FFT is used. These values correspond to the sample rate of 30.72 MHz. The cyclic prefix length varies during OFDM demodulation of the SIB1 grid to account for the longer cyclic prefix symbols present at the halfsubframe boundaries. The number of guard subcarriers is used to extract the SIB1 grid from the full demodulation bandwidth.

The `OFDM Demod` subsystem requires configuration values to successfully demodulate the SIB1 grid. These values are computed from the results of MIB recovery. The `SIB1 Config` subsystem constructs the MIB from the PBCH payload and parses the `pdcchSIB1Config` field to determine `coreset0Resources` and `coreset0TimingOccasion`. The `coreset0Resources` signal contains the frequency offset from the SSB to CORESET0 and the bandwidth of the CORESET0 resource grid. The `coreset0TimingOccasion` signal contains the slot offset from the even SFN frame head to the first monitored slot i.e. the slot within 2 system frames. The monitored slot timing offset subsystem converts the `coreset0TimingOccasion` slot offset to a timing reference count. The SSB pattern, SSB index, and SSB timing reference are used to compute the timing reference of the even SFN frame head, from which the timing reference value of CORESET0 is computed. A timing reference counter, synchronized with the SSB detection references, is used to track the SIB1 data stream. Once the start signal is asserted, the startController waits for the SIB1 timing reference to reach the target offset computed by the monitored slot timing offset subsystem and then triggers the OFDM demodulation to begin. The diagram shows an example of the timing references for the monitored

SSB, the even frame head, and the CORESET0 timing occasion. These offsets depend on the configuration of the transmitting cell.
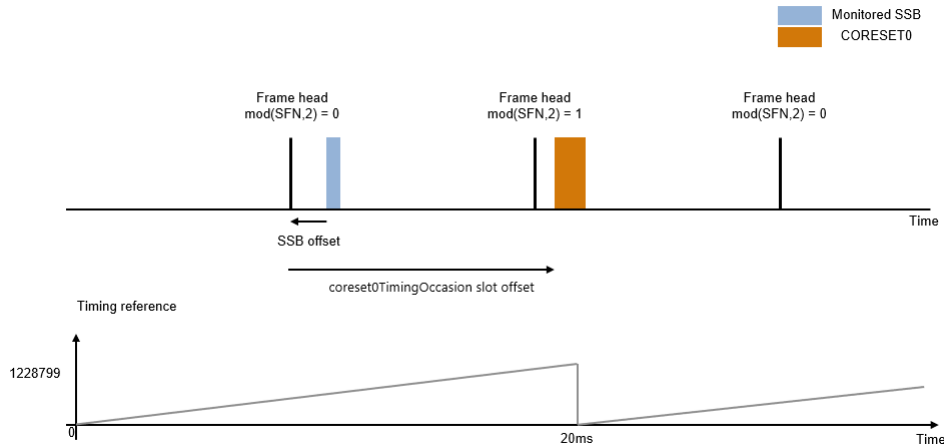
NR HDL CORESET0 Decoding

This figure shows the `nrhdlCORESET0Decoding` model. The top level of the model reads the signals from the MATLAB base workspace, passes them to the CORESET0 Decoding subsystem, and writes the outputs back to the workspace.

```
models.CORESET0Top = 'nrhdlCORESET0Decoding';
open_system(models.CORESET0Top);
```
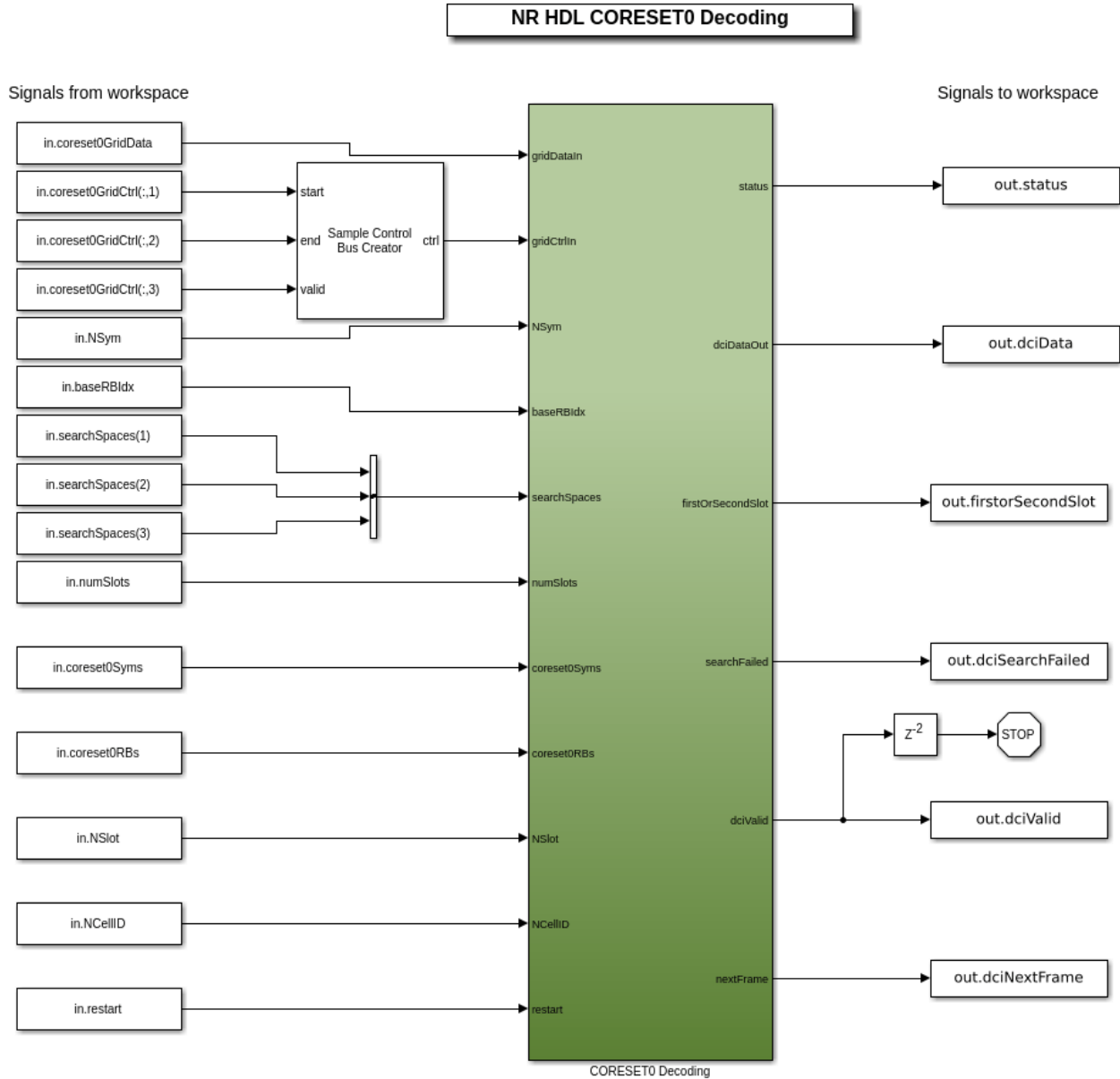
CORESET0 Decoding Subsystem

The CORESET0 Decoding subsystem references the `nrhdlCORESET0DecodingCore` and `nrhdlPolarDecodingChainCore` models. The algorithm of the `nrhdlCORESET0DecodingCore` model is described in the next section. The `nrhdlPolarDecodingChainCore` model is covered in the "NR HDL MIB Recovery" on page 5-56 example. The subsystem performs channel estimation and equalization, QPSK symbol demodulation, descrambling, rate recovery, polar decoding, and CRC decoding of CORESET0 candidates. The design provides back pressure with the nextFrame signal to indicate when it can accept a new candidate. The processing is split over two models to allow for the `nrhdlPolarDecodingChainCore` to be shared between the SSB decoding and SIB1 CORESET0

decoding in the "NR HDL SIB1 Recovery" on page 5-8 example. This section describes the inputs and outputs for the subsystem.

```
set_param([models.CORESET0Top '/CORESET0 Decoding'],'Open','on');
```


Inputs

- *gridDataIn*: 16-bit signed CORESET0 candidate OFDM grid data.
- *gridCtrlIn*: Sample control bus signal to validate *gridDataIn*.
- *NSym*: 4-bit OFDM symbol number for the current resource element group (REG).
- *baseRBIdx*: 7-bit base CORESET0 resource block index for the current REG.
- *searchSpaces*: 3-bit unsigned vector of length 3 indicating the number of search spaces at aggregation levels 4, 8, and 16.
- *coreset0Syms*: 2-bit unsigned value that is the number of OFDM symbols CORESET0 spans.
- *coreset0RBs*: 2-bit unsigned value specifying the number of resource blocks. A value of 0 indicates 24, 1 indicates 48, and 2 indicates 96.
- *NSlot*: 7-bit unsigned value that specifies the slot number for the first monitored CORESET0 slot.
- *NCellID*: 10-bit unsigned value that is the cell ID of the demodulated SSB.
- *restart*: 1-bit control signal to restart the processing.

Outputs

- *status*: 3-bit unsigned value indicating the progress of the CORESET0 decoding process.
- *dciData*: 41-bit unsigned data that contains the final decoded DCI.
- *firstOrSecondSlot*: 1-bit value indicating if the decoded DCI was found in the first (0) or second (1) monitored slot.
- *searchFailed*: 1-bit value indicating that the CORESET0 DCI search failed.
- *dciValid*: 1-bit value indicating the search is complete.
- *nextFrame*: 1-bit signal to provide back pressure to signal when the next candidate can be input.

Status Signal States

- `0`: Initial state. Waiting for start pulse.
- `1`: Performing channel estimation, equalization, symbol demodulation and descrambling.
- `2`: Performing polar rate recovery.
- `3`: Performing polar and CRC decoding.
- `4`: Candidate decode failed, waiting for next attempt.
- `5`: Decoded all candidates with no successes.
- `6`: Successfully decoded the DCI from a candidate.

CORESET0 Decoding Model

This diagram shows the top level of the `nrhdlCORESET0DecodingCore` model. To decode CORESET0 a blind search is performed over multiple candidates. The correct candidate is determined by the CRC remainder equaling the SIB1 RNTI 65535. The design coordinates each search step and generates the required parameters to decode the candidate with the `nrhdlPolarDecodingChainCore` model. After a candidate is decoded, the CRC result is checked. If the CRC passes, the decoded data is output on the *dciData* port. If the CRC fails, the algorithm signals with *nextFrame* that it is ready for the next candidate. If all candidates are decoded with no success then *searchFailed* is set high. The number of search candidates per slot is determined from the *searchSpaces* input, this signal is a vector of 3 values correpsonding to the number of candidates for the three possible aggregation values [4 8 16]. The design expects candidates to be input in decreasing order of aggregation level. The total number of searches is twice the search spaces since two slots are monitored for decoding. The *firstOrSecondSlot* output signals which slot the DCI was decoded in. The status signal can be used to monitor the progress of the decoding. Each candidate failure is indicated by state `4` and the final result is signalled by either state `5` (failure) or state `6` (success).

```
models.coreset0Decoding = 'nrhdlCORESET0DecodingCore';
load_system(models.coreset0Decoding);
set_param(models.coreset0Decoding,'SimulationCommand','Update');
set_param(models.coreset0Decoding,'Open','on');
```
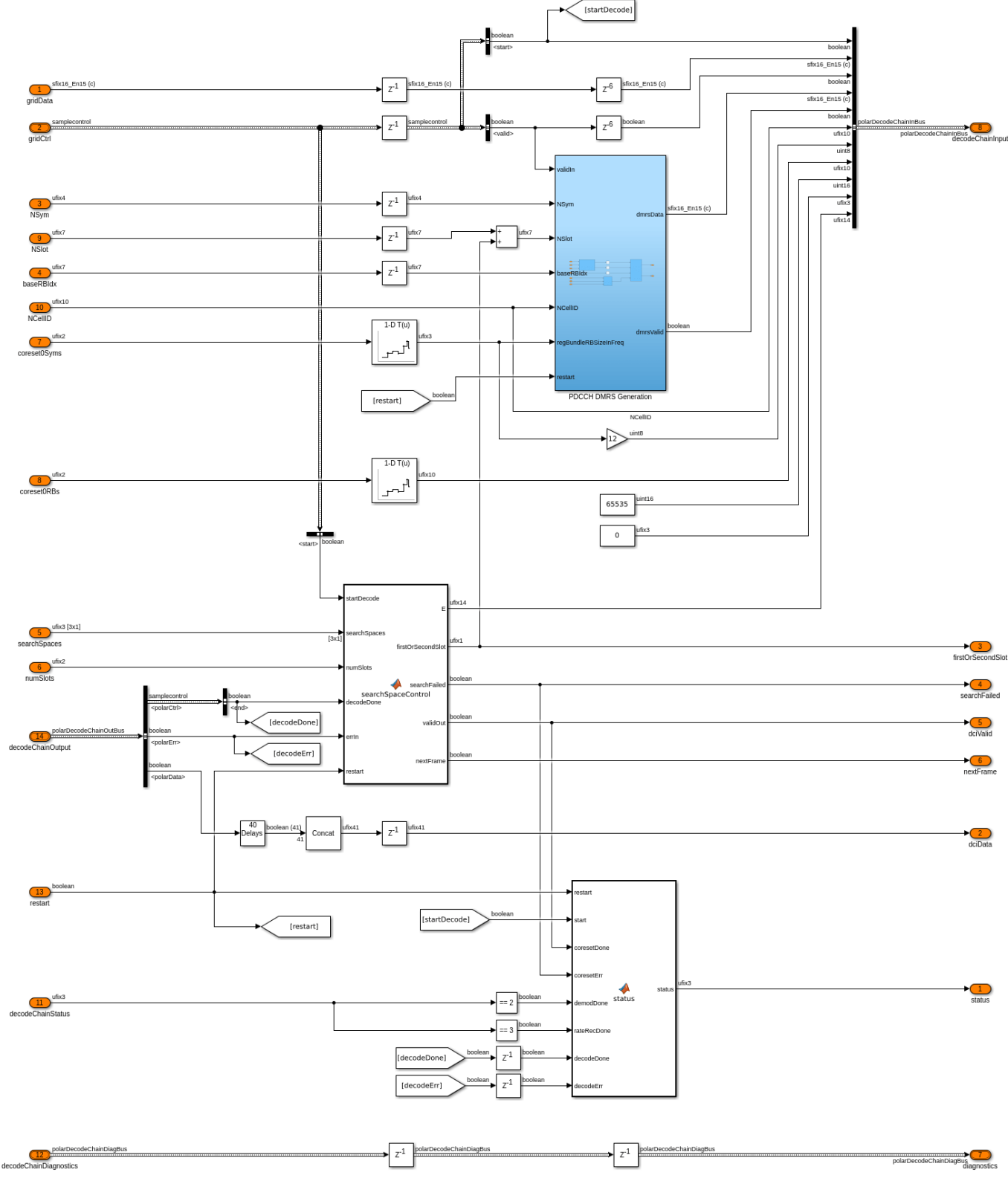
NR HDL SIB1 LDPC Decoding

This figure shows the `nrhdlSIB1LPDCDecoding` model. The top level of the model reads the signals from the MATLAB base workspace, passes them to the SIB1 LDPC Decoding subsystem, and writes the outputs back to the workspace.

```
models.LDPCTop = 'nrhdlSIB1LDPCDecoding';
open_system(models.LDPCTop);
```
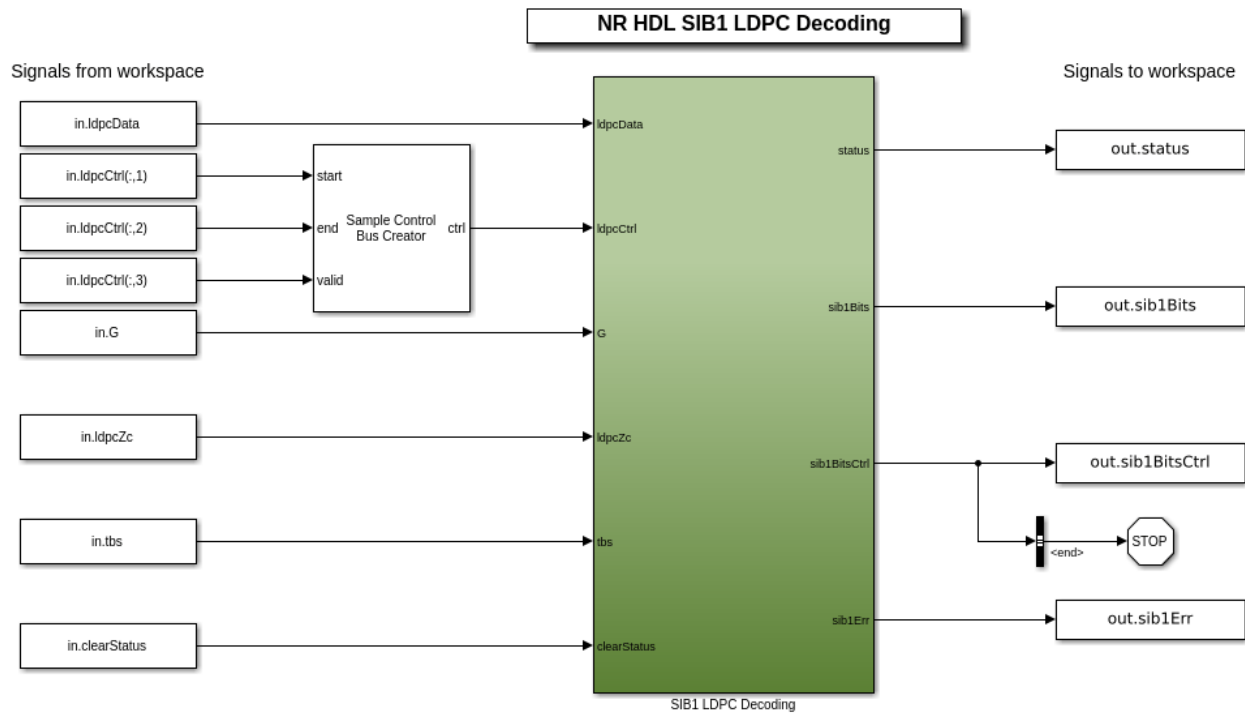

Copyright 2021-2022 The MathWorks, Inc.

SIB1 LDPC Decoding Subsystem

The `SIB1 LDPC Decoding` subsystem references the `nrhdlLDPCDecodingChainCore` model. The subsystem performs LDPC decoding, codeblock desegmentation, and CRC decoding. This section describes the inputs and outputs for the subsystem.

```
set_param([models.LDPCTop '/SIB1 LDPC Decoding'],'Open','on');
```
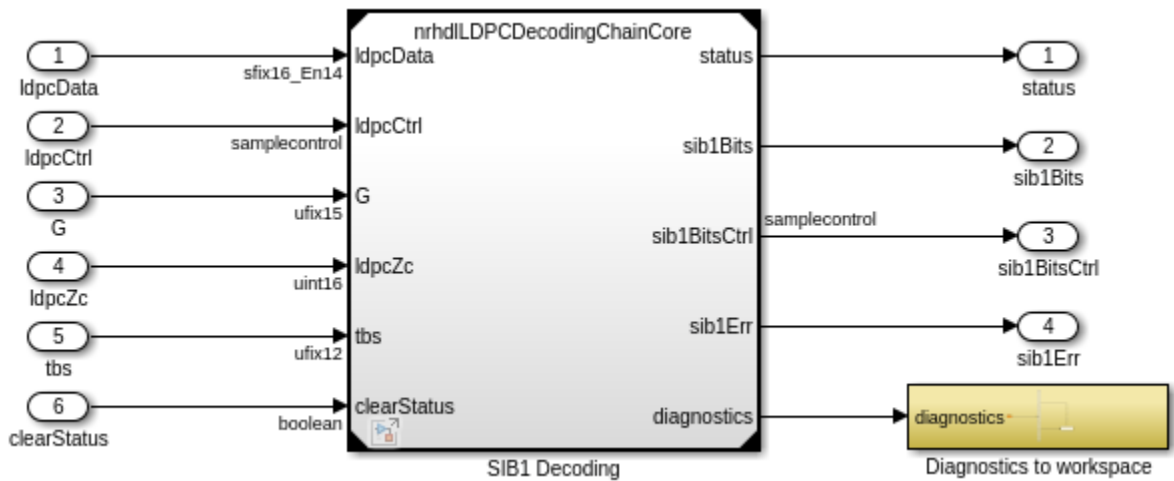
SIB1 Decoding

Inputs

- *ldpcDta*: 16-bit signed LDPC codeword LLR data.
- *ldpcCtrl*: Sample control bus for validating *ldpcData*.
- *G*: 15-bit length of the input codeword.
- *ldpcZc*: 16-bit unsigned value indicating the lifting size used for the LDPC codeword.
- *tbs*: 12-bit unsigned value indicating the length of the decoded output data.
- *clearStatus*: 1-bit control signal used to clear the state of the status signal.

Outputs

- *status*: 3-bit unsigned value indicating the progress of the SIB1 decoding process.
- *sib1Bits*: 1-bit data that is the final decoded SIB1 payload.
- *sib1BitsCtrl*: Sample control bus for validating *sib1Bits*.
- *sib1Err*: 1-bit value indicating if the SIB1 CRC failed.
- *diagnostics*: Bus containing diagnostic signals.

Status Signal States

- `0`: Initial state. Waiting for start pulse.
- `1`: Performing LDPC rate recovery.
- `2`: Performing LDPC decoding.
- `3`: Performing CRC decoding.
- `4`: Failed to decode SIB1.
- `5`: Successfully decoded SIB1.

SIB1 LDPC Decoding Model

This diagram shows the top level of the `nrhdlLDPCDecodingChainCore` model. The design accepts input LLRs along with a sample control bus and additional constants. The first stage performs LDPC rate recovery, this includes signal scaling and wordlength reduction to prepare the data for LDPC decoding.The second stage decodes the LDPC data using min-sum layered belief propagation with

lifting factor from input port. For the SIB1 use case only base graph 2 is supported so the **bgn** input is constant. The algorithm then performs codeblock desegmentation. For SIB1 this only requires the removal of the padding bits from the end of the data to produce a length of **tbs** + 16 CRC bits . The final stage performs CRC decoding using CRC16 with no scrambling. The decoded SIB1 bits and the error status are output. Additionally, a status port is provided to show the algorithms progress. The output from the rate recovery stage is provided on the diagnostic port.

```
models.ldpcDecoding = 'nrhdlLDPCDecodingChainCore';
load_system(models.ldpcDecoding);
set_param(models.ldpcDecoding,'SimulationCommand','Update');
set_param([models.ldpcDecoding '/LDPC Decoding Chain'],'Open','on');
```
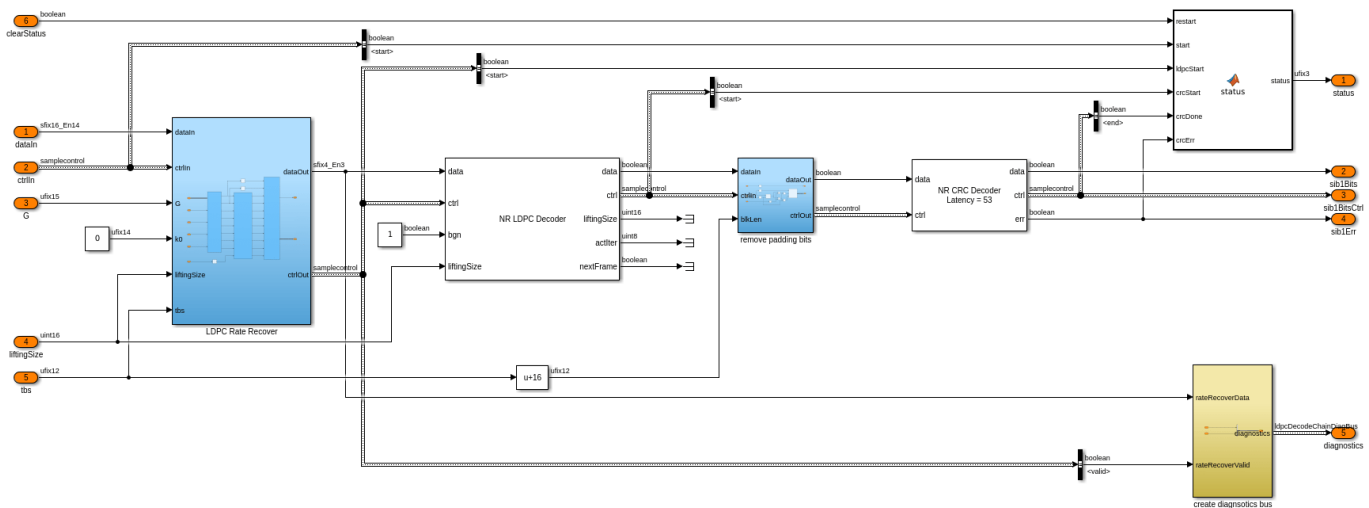

SIB1 Accelerators Simulation Setup

The diagram shows the simulation setup implemented by this example. 5G Toolbox functions are used to generate a test waveform. MATLAB reference code is then used to perform the steps required for MIB recovery - SSB search, demodulation, and decoding. The results provide the input data for the SIB1 demodulation stage. The same input is passed to both MATLAB and Simulink implementations of SIB1 demodulation, and the output grids are directly compared. The CORESET0 candidates are extracted from the grid and decoded using MATLAB and Simulink. The DCI result is used to extract the SIB1 LDPC codeword from the resource grid and the final decode is performed in MATLAB and Simulink. At each stage MATLAB and Simulink results are compared to confirm their equivalence.
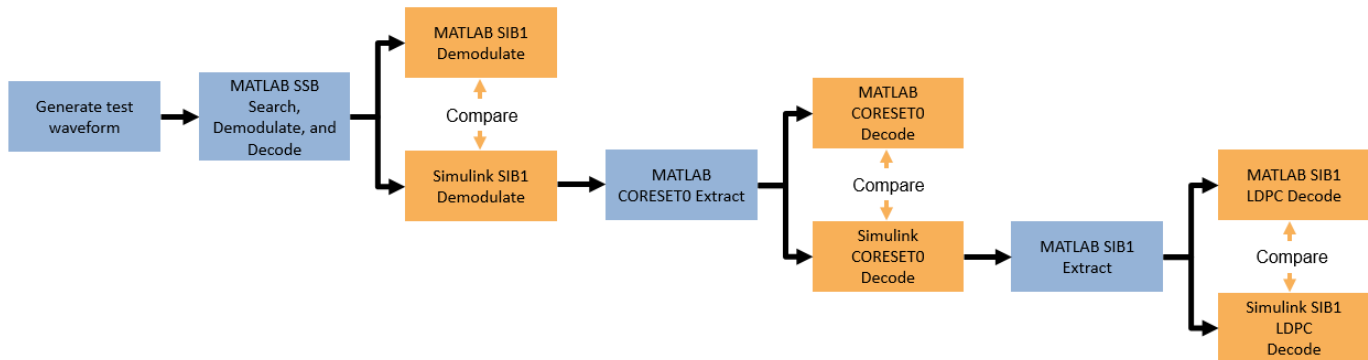
SIB1 Accelerators Simulation

Use the `runSIB1AcceleratorModels` script to run a SIB1 recovery simulation using the hardware accelerators. The script displays its progress at the MATLAB command prompt, and produces plots of inputs and outputs for analysis. The test bench supports multiple simulation cases. The full set of cases, and their parameters, are shown. This example shows the results of running "SimCase 1". The resource grids produced by MATLAB and Simulink from the SIB1 demodulation are displayed along with the difference between them and their relative mean squared error (MSE). The grid plots are labelled to highlight the decoded PDCCH and PDSCH. The DCI fields from CORESET0 decoding are displayed and the final SIB1 bits are compared. This comparison verifies that the Simulink implementations closely matches the MATLAB reference.

`disp(nrhdlexamples.generateFR1RxWaveform('list'));`

Simulation Case	SSB Pattern	Subcarrier Spacing Common	PDCCH Config SIB1	SNR dB
"SimCase 1"	"Case C"	30	164	50
"SimCase 2"	"Case B"	15	100	6
"SimCase 3"	"Case A"	30	4	20
"SimCase 4"	"Case A"	15	84	7

`runSIB1AcceleratorModels;`

`runSIB1AcceleratorModels;`

```
Generating test waveform.
Searching for SSBs using MATLAB reference.
Demodulating the strongest SSB using MATLAB reference.
Decoding the demodulated SSB using MATLAB reference.
Demodulating the SIB1 grid using MATLAB reference.
Demodulating the SIB1 grid using Simulink model.
Running nrhdlSIB1Demodulation.slx
### Starting serial model reference simulation build
### Model reference simulation target for nrhdlDDCFR1Core is up to date.
### Model reference simulation target for nrhdlSIB1DemodulationFR1Core is up to date.

Build Summary

0 of 2 models built (2 models already up to date)
```

```
Build duration: 0h 0m 0.76361s
..................
MATLAB and Simulink grids relative MSE : -62.453 dB

Extracting CORESET0 candidates from the SIB1 grid.
Decoding CORESET0 candidates using MATLAB reference.
Decoding CORESET0 candidates using Simulink.
Running nrhdlCORESET0Decoding.slx
### Starting serial model reference simulation build
### Model reference simulation target for nrhdlCORESET0DecodingCore is up to date.
### Model reference simulation target for nrhdlPolarDecodingChainCore is up to date.

Build Summary

0 of 2 models built (2 models already up to date)
Build duration: 0h 0m 0.92363s
..........
DCI from MATLAB:
                      RIV: 528
                 TDDIndex: 0
     VRBToPRBInterleaving: 0
                ModCoding: 0
                       RV: 0
              SIIndicator: 0
                 Reserved: 0

DCI from Simulink:
                      RIV: 528
                 TDDIndex: 0
     VRBToPRBInterleaving: 0
                ModCoding: 0
                       RV: 0
              SIIndicator: 0
                 Reserved: 0

DCI successfully decoded from Simulink grid with hardware acceleration

Extracting LDPC codeword from the SIB1 grid.
Decoding SIB1 using MATLAB reference.
Decoding SIB1 using Simulink.
Running nrhdlSIB1LDPCDecoding.slx
### Starting serial model reference simulation build
### Model reference simulation target for nrhdlLDPCDecodingChainCore is up to date.

Build Summary

0 of 1 models built (1 models already up to date)
Build duration: 0h 0m 0.61133s
..........
SIB1 successfully decoded from Simulink grid with hardware acceleration
SIB1 bits from MATLAB and Simulink match
```

Spectrogram of Generated Waveform

Difference between MATLAB and Simulink grids (real)

Difference between MATLAB and Simulink grids (imag)

SIB1 Grid from Simulink

SIB1 Grid from MATLAB Reference
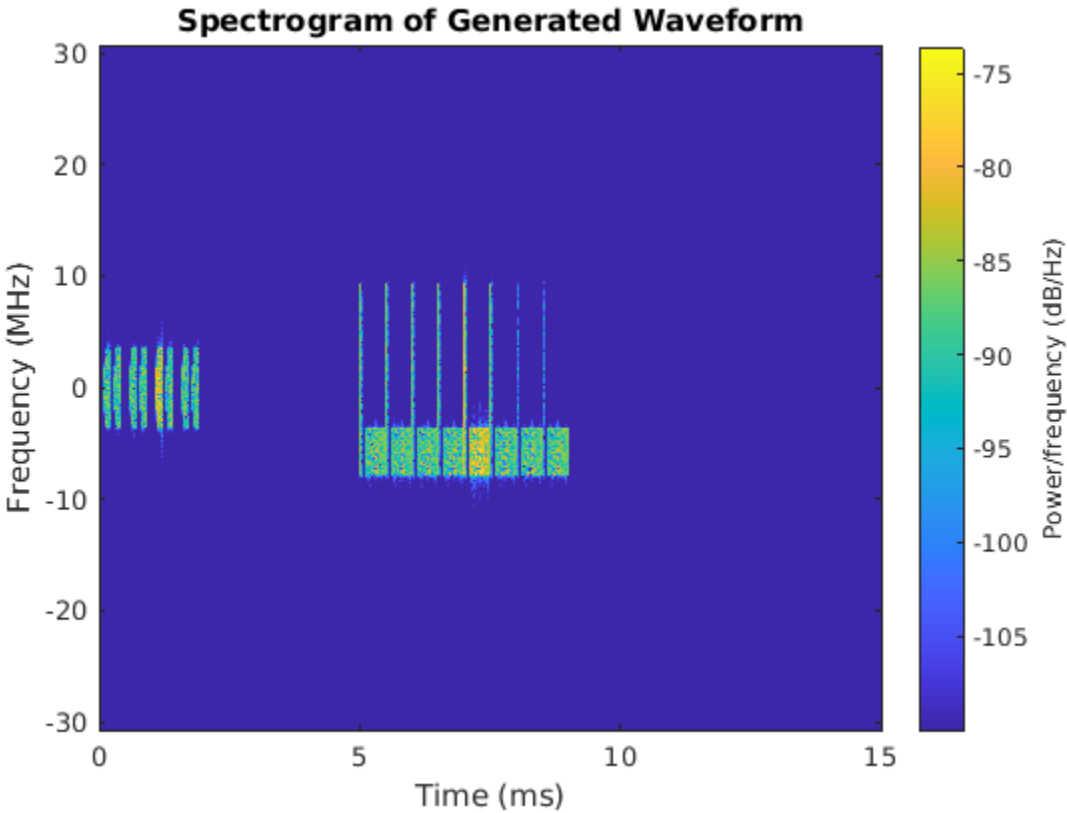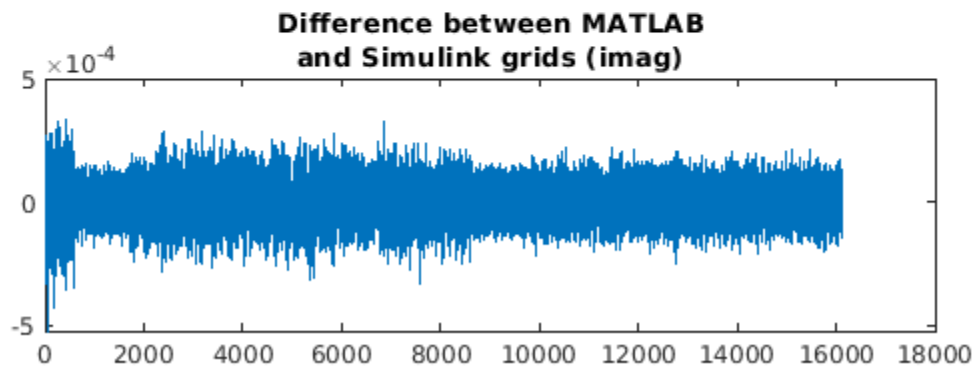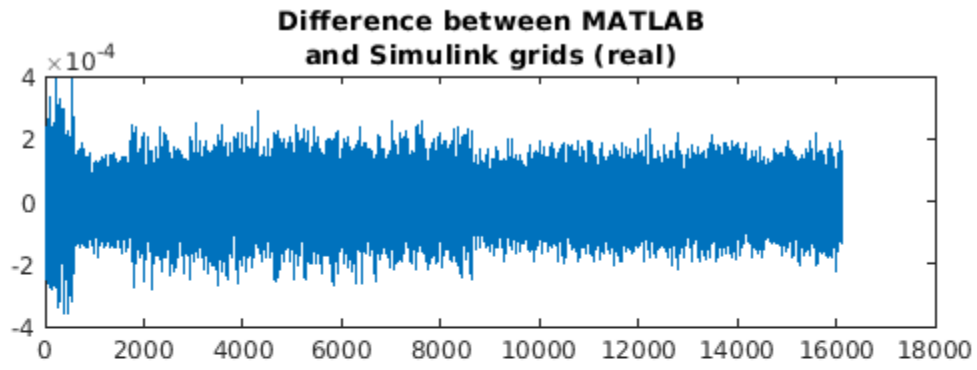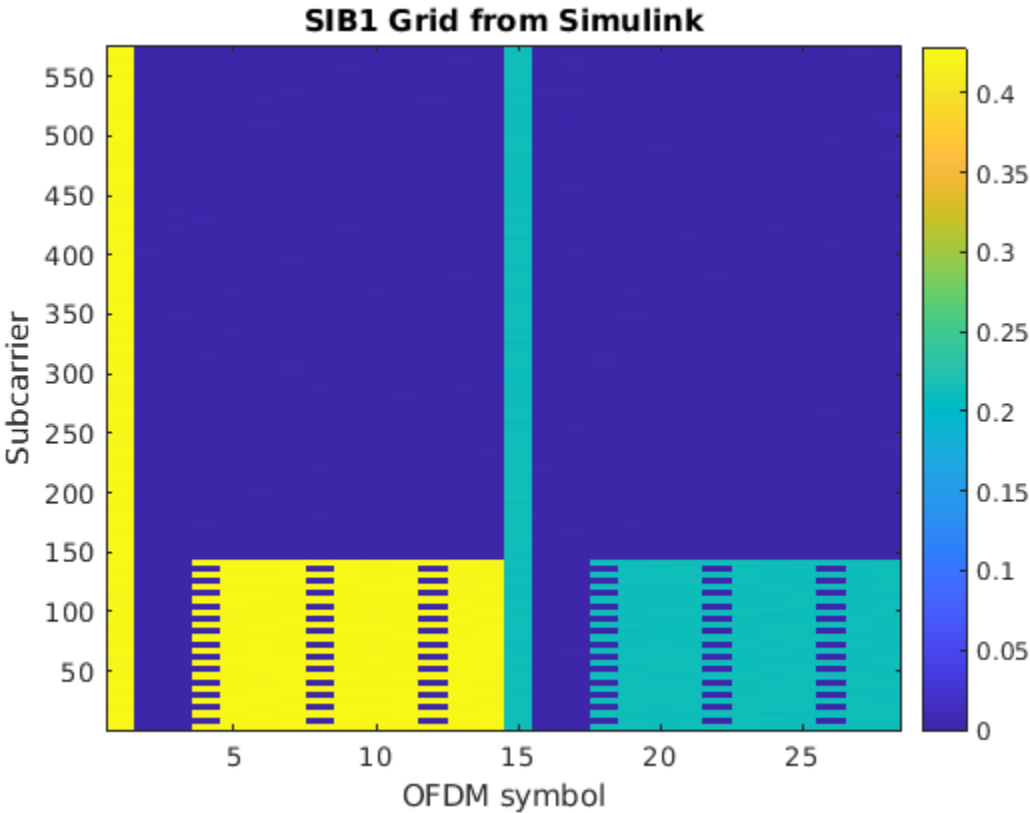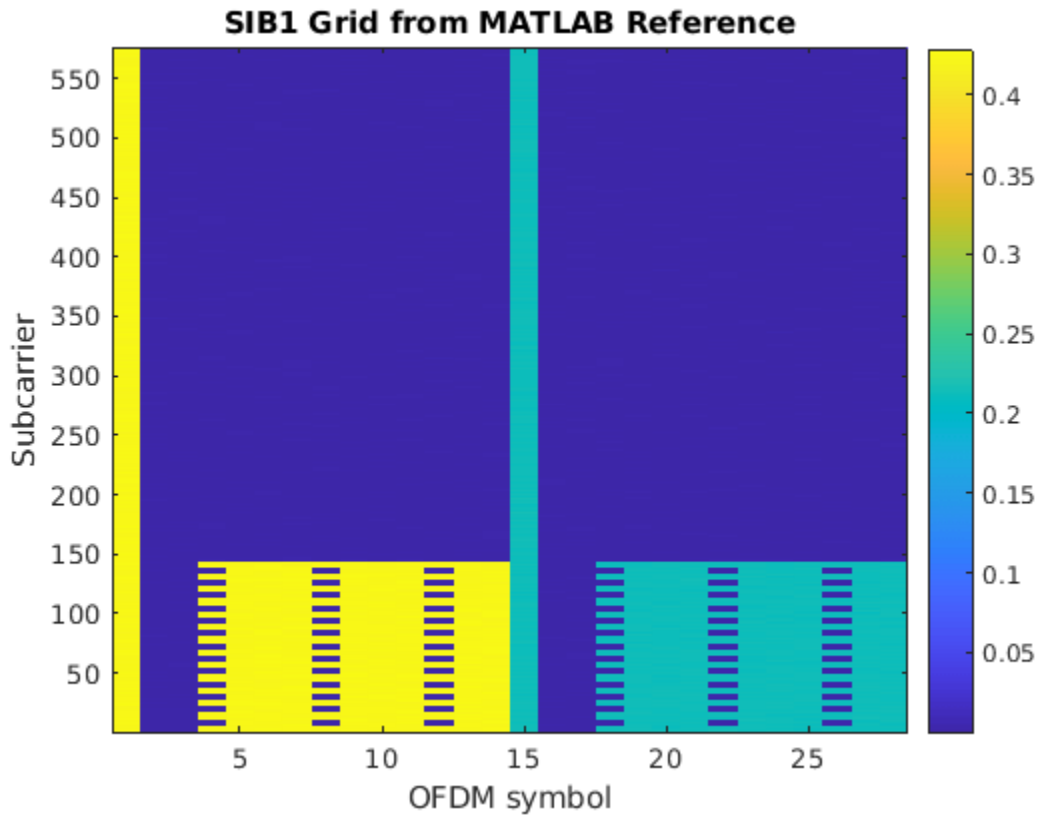
HDL Code Generation and Implementation Results

To generate the HDL code for this example, you must have an HDL Coder™ license. Use the `makehdl` and `makehdltb` commands to generate HDL code and an HDL test bench for the `nrhdlSIB1Demodulation/SIB1 Demodulation`, `nrhdlCORESET0Decoding/CORESET0 Decoding` or `nrhdlSIB1LDPCDecoding/SIB1 LDPC Decoding` subsystems. The resulting HDL code was synthesized for a Xilinx® Zynq®-7000 ZC706 evaluation board. The table shows the post place and route resource utilization results. The design meets timing with a clock frequency of 150 MHz.

Resource utilization for nrhdlSIB1Demodulation model:

```
T = table(...
    categorical({'Slice Registers'; 'Slice LUTs'; 'RAMB18'; 'RAMB36'; 'DSP48'}),...
    [12932; 7338; 19; 10; 35],...
    'VariableNames',{'Resource','Usage'});
```

```
disp(T);
```

```
     Resource        Usage
    _____  _____

    Slice Registers   12932
    Slice LUTs         7338
    RAMB18               19
    RAMB36               10
```

```
    DSP48                      35
```

Resource utilization for nrhdlCORESET0Decoding model:

```
T = table(...
    categorical({'Slice Registers'; 'Slice LUTs'; 'RAMB18'; 'RAMB36'; 'DSP48'}),...
    [8291; 11073; 8; 4; 16],...
    'VariableNames',{'Resource','Usage'});

disp(T);

       Resource          Usage
    _____      _____

    Slice Registers       8291
    Slice LUTs           11073
    RAMB18                   8
    RAMB36                   4
    DSP48                   16
```

Resource utilization for nrhdlSIB1LDPCDecoding model:

```
T = table(...
    categorical({'Slice Registers'; 'Slice LUTs'; 'RAMB18'; 'RAMB36'; 'DSP48'}),...
    [61821; 42092; 289; 20; 3],...
    'VariableNames',{'Resource','Usage'});

disp(T);

fields = fieldnames(models);
for k=1:length(fields)
    close_system(models.(fields{k}),0);
end

       Resource          Usage
    _____      _____

    Slice Registers      61821
    Slice LUTs           42092
    RAMB18                 289
    RAMB36                  20
    DSP48                    3
```

See Also

Related Examples

- "NR HDL SIB1 Recovery" on page 5-8

More About

- "NR HDL Receiver Performance" on page 5-5

NR HDL SIB1 Recovery for FR2

This example shows how to design a 5G NR system information block 1 (SIB1) recovery model that is optimized for HDL code generation and hardware implementation and that supports frequency range 1 (FR1) and frequency range 2 (FR2).

Introduction

5G cell towers can operate in either FR1 or FR2 frequency bands. FR1 covers frequencies up to 6 GHz, and FR2 covers frequencies above 6 GHz, including the millimeter wave band. SIB1 recovery requires cell search, master information block (MIB) decoding, recovery of the SIB1 grid (the area of the resource grid containing CORESET0 and SIB1), and decoding of the CORESET0 PDCCH and SIB1 PDSCH from the SIB1 grid. The process of FR1 Cell Search and MIB recovery are described in the "NR HDL Cell Search" on page 5-88 and "NR HDL MIB Recovery" on page 5-56 examples respectively. The additonal models used to implement FR1 SIB1 grid recovery, CORESET0 decoding, and SIB1 decoding are described in the "Hardware Accelerators for NR SIB1 Recovery" on page 5-23 example. This example introduces functionality that is required to support FR2 and the process of upgrading an existing FR1 design.
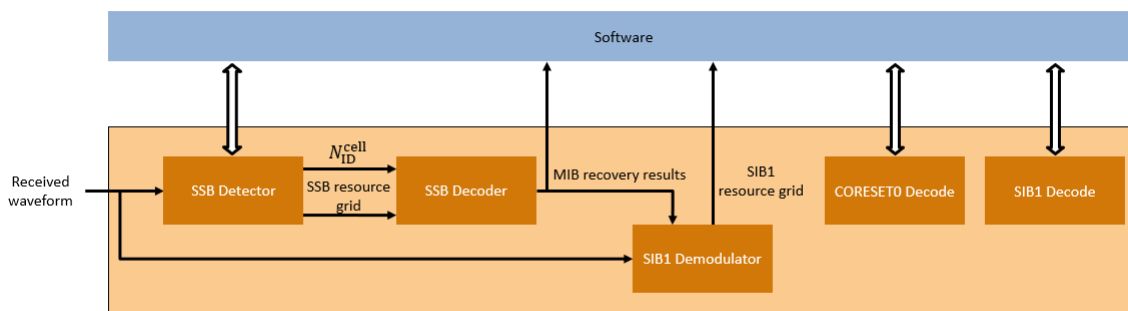
The Simulink® models described in this example are fixed-point HDL-optimized implementations of SIB1 recovery for 5G NR FR1 and FR2. This example is one of a related set, for more information see "NR HDL Reference Applications Overview" on page 5-2.

File Structure

This example uses these files.

Simulink Models

- `nrhdlSIB1Recovery.slx`: This Simulink model combines the processing of the SSB detector, SSB decoder, SIB1 demodulator, CORESET0 decoder, and SIB1 decoder into an integrated model illustrating the complete SIB1 grid recovery process. This model references the `nrhdlDDCCore`, `nrhdlSSBDetectionCore`, `nrhdlSSBDecodingCore`, `nrhdlPolarDecodingChainCore`, `nrhdlSIB1DemodulationCore`, `nrhdlCORESET0DecodingCore`, and `nrhdlLDPCDecodingChainCore` models.

- `nrhdlDDCCore.slx`: This model implements a DDC to create sample streams for SIB1 and SSBs.

- `nrhdlSSBDetectionCore.slx`: This model implements the SSB detection algorithm.

- `nrhdlSSBDecodingCore.slx`: This model implements the SSB decoding algorithm.
- `nrhdlPolarDecodingChainCore.slx` : This model implements the common polar decoding chain.
- `nrhdlSIB1DemodulationCore.slx`: This model implements the SIB1 Demodulation algorithm.
- `nrhdlCORESET0DecodingCore.slx` : This model implements the CORESET0 decoding algorithm.
- `nrhdlLDPCDecodingChainCore.slx` : This model implements the SIB1 LDPC decoding algorithm.

Simulink Data Dictionary

- `nrhdlReceiverData.sldd`: This Simulink data dictionary contains bus objects that define the buses contained in the example models.

MATLAB Code

- `runSIB1RecoveryModelFR2.m`: This script uses the MATLAB reference to perform the search mode of the SSB detection algorithm, then runs the `nrhdlSIB1Recovery` Simulink model to demodulate and decode the SSB, and then demodulate the SIB1 grid. The script performs CORESET0 and SIB1 decoding using either MATLAB code designed for embedded software or the hardware accelerators in the `nrhdlSIB1Recovery` model.
- `nrhdlexamples`: This package contains the MATLAB reference code and utility functions for verifying the implementation models.

SSB Detection

This section describes the changes to the SSB detection MATLAB reference and Simulink implementation in the "NR HDL Cell Search" on page 5-88 example that are required to support FR2.

The SSB detection algorithm performs search and demodulation with a given subcarrier spacing (SCS). The SCS options are 15 kHz or 30 kHz for FR1 and 120 kHz or 240 kHz for FR2. To add FR2 functionality, the new SCS options must be supported. The detector searches for SSBs by downsampling the received signal to one of the rates shown in the table according to the SCS. The signal is then cross-correlated with the PSS sequences.

SCS (kHz)	Sample Rate (MHz)
15	3.84
30	7.68
120	30.72
240	61.44

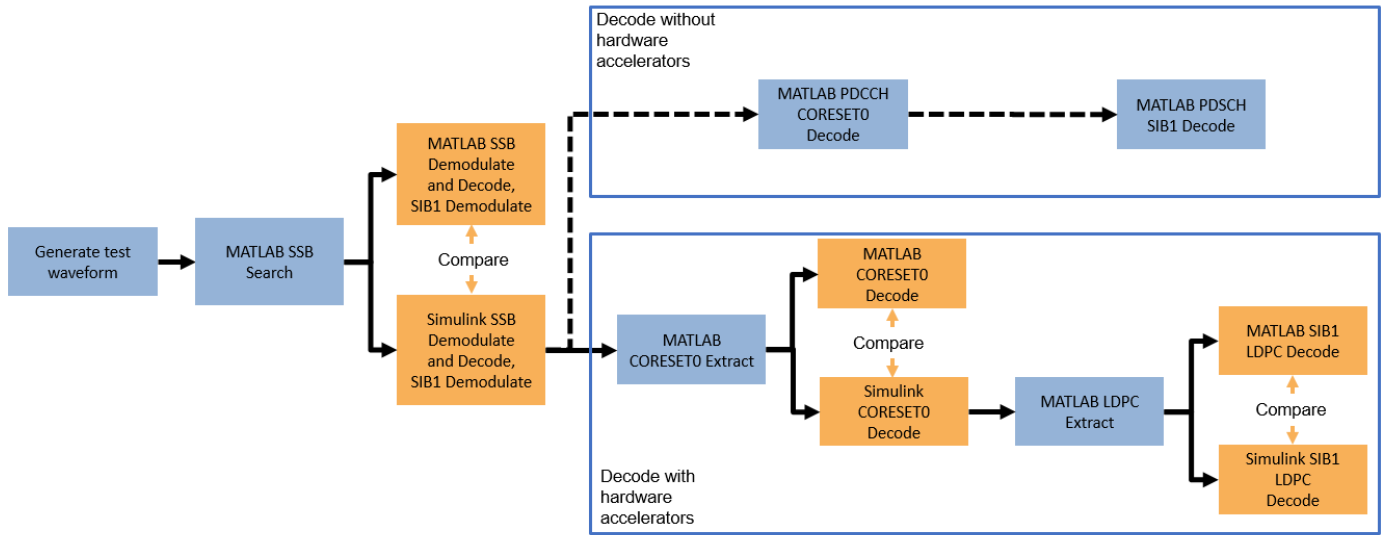To accommodate the increased bandwidth requirement of the SSBs in FR2, an input sampling rate of 61.44 Msps is used for the `nrhdlSSBDetectionCore` model (compared to 7.68 Msps for the FR1 design). The timing reference units are unchanged and still measured in samples at 61.44 Msps. The timing reference counters increment in steps of 16, 8, 2, and 1 for SCS of 15, 30, 120, and 240 kHz, respectively. The `SCS selection` subsystem inside `nrhdlSSBDetection` performs the first processing step for SSB detection. This subsystem creates the data streams for each SCS option by successively downsampling the data with halfband filters. All four streams are aligned, enabling the timing reference to be maintained when switching between different subcarrier spacings. The signal

processing chain implemented in the `nrhdlSSBDetectionCore` is shown. A single SCS data stream is selected for downstream processing.
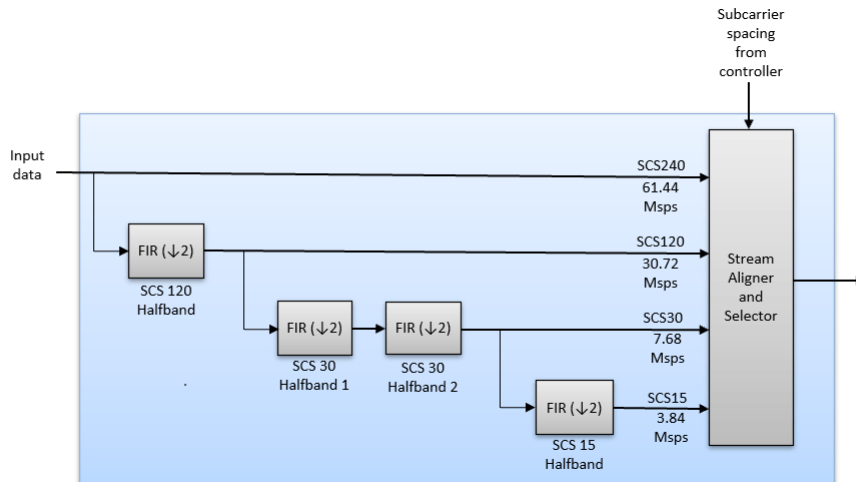
The selected SCS data stream is correlated against each of the three PSS sequences to detect SSBs. The FPGA implementation of these correlators in the time domain uses 144 DSPs. A frequency domain overlap-add method is used to minimize the DSP usage at the expense of a small increase in latency. This figure shows the overlap-add correlation in the `nrhdlSSBDetectionCore` model. The subsystem computes the four stages of the overlap-add method: FFT, multiplication by the three sets of frequency domain coefficients (one for each PSS), IFFT, and overlap and add of subsequent windows. This implementation uses one FFT, three complex multipliers, and three IFFTs, requiring 48 DSP blocks in total.
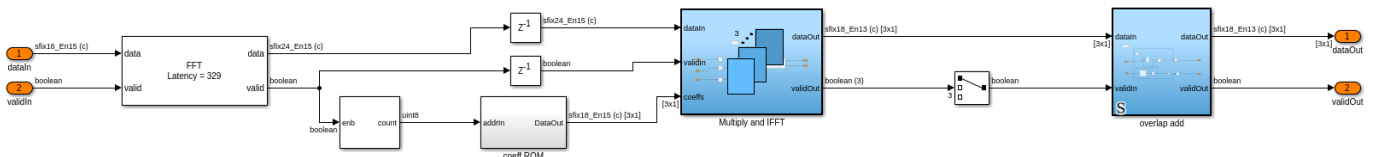
SSB Decoding

This section describes the changes to the SSB decoding MATLAB reference and Simulink implementation in the "NR HDL MIB Recovery" on page 5-56 example that are required to support FR2.

The SSB decoding algorithm decodes the broadcast channel (BCH) contained in the SSB. The decoding process outputs the MIB and the beam index of the detected SSB. In FR1, the maximum number of SSBs that can be independently beamformed is 8. FR2 supports transmitting up to 64 SSBs, each on their own beam. The contents of the MIB between FR1 and FR2 differs to accommodate the increase in the maximum number of beams.

The `nrhdlexamples.ssbDecode` function and `nrhdlSSBDecodingCore` model accept `Lmax` as an input. `Lmax` is the maximum number of beams that can be transmitted by a cell tower, and its value depends on the carrier frequency. Valid settings for `Lmax` are 4 or 8 for FR1 and 64 for FR2. `Lmax` affects the descrambling in the BCH processing subsystem and how the final BCH payload is parsed.
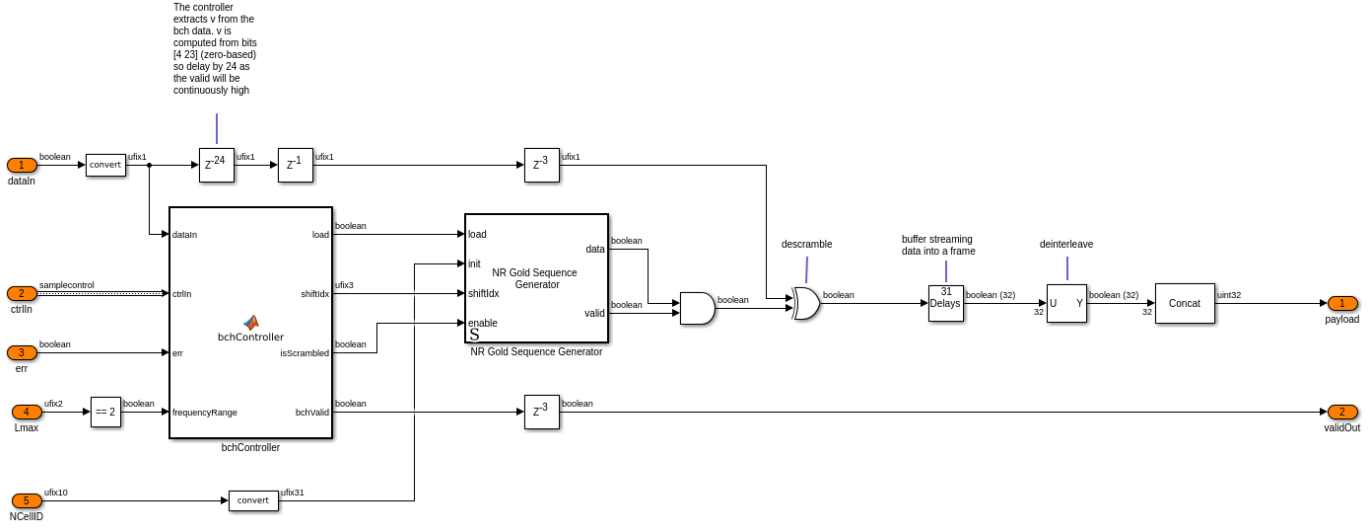
SIB1 Grid Demodulation

This section describes the changes to the SIB1 demodulation MATLAB reference and Simulink implementation in the "Hardware Accelerators for NR SIB1 Recovery" on page 5-23 example that are required to support FR2.

The SIB1 demodulation algorithm performs OFDM demodulation of the SIB1 grid, and includes the lookup tables required to identify the time and frequency resources which contain the SIB1. The SIB1 grid is modulated with a SCS set by the scsCommon field in the MIB. The SCS options are 15 kHz or 30 kHz for FR1 and 60 kHZ or 120 kHz for FR2. To incorporate the wider FR2 bandwidth the FR2 algorithm requires an input sample rate of 122.88 Msps. This is the minimum sampling rate which results in a power of 2 FFT size for FR2 configurations.

For SCS 15 and 60 kHz a 2048-point FFT is used, for 30 and 120 kHz a 1024-point FFT is used. To compute the FFT for the FR1 SCS options, the input sample stream must be downsampled from the input rate of 122.88 Msps to 30.72 Msps. The FR2 SCS options use the input data unmodified. The `Frequency Range Selection` subsystem creates the two data streams and corresponding timing references for each frequency range. The `FR1 Downsample` subsystem uses two cascaded halfband filters to create the FR1 data and the input data is used directly for FR2. The output from the subsystem is selected based on `Lmax`, a value of 2 indicates FR2 otherwise FR1 is selected.
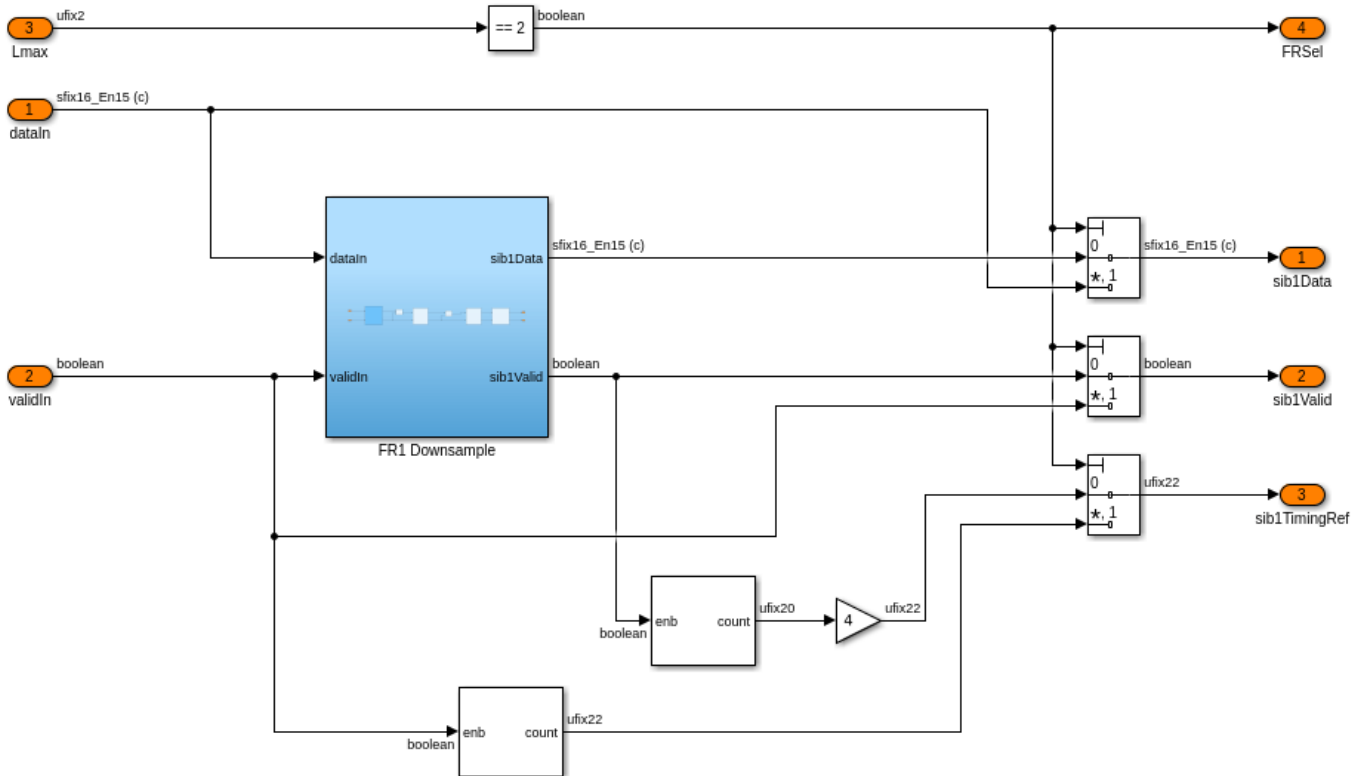
The `SIB1 Config` subsystem is updated to include the additional lookup tables required to generate the `coreset0Resources` and `coreset0TimingOccasion` signals for the FR2 use case.

DDC

This section describes the changes to the DDC MATLAB reference and Simulink implementation in the "NR HDL Cell Search" on page 5-88 example that are required to support FR2.

To accomodate the increased bandwidths of the SSBs and SIB1 in FR2, an input sampling rate of 491.52 Msps is used (compared to 61.44 Msps for the FR1 design). This input rate gives the receiver sufficient bandwidth to sample the SSB and SIB1 subcarriers, provided that the signal is centered on the SSBs. It also includes overhead for the coarse frequency search. This bandwidth allows for MIB and SIB1 recovery without retuning the center frequency of the frontend. To meet the throughput requirements and avoid errors during timing closure, the design uses a clock rate of 245.76 MHz with two samples per clock.

The SIB1 stage creates the SIB1 outputs using two cascaded halfband filters. The first filter operates on the input data stream and downsamples to create an intermediate stream at 245.76 Msps. This downsample reduces the frame size from two samples per clock to one. All downstream processing is performed on scalar data, with the valid duty cycle reducing through each filtering stage. The SSB stage creates the SSB outputs by using a single halfband to implement the filter and downsample from the 122.88 Msps SIB1 rate to the 61.44 Msps rate required for SSB detection.

SIB1 Recovery Simulation Setup

The block diagram shows the simulation setup implemented by this example. The orange blocks highlight the comparison points between the MATLAB reference and the Simulink HDL implementation. The `runSIB1RecoveryModelFR2` script runs the simulation, the MATLAB code represents the software control algorithm and the Simulink simulations perform the FPGA processing. 5G Toolbox™ functions are used to generate a test waveform. MATLAB reference code is used to perform the SSB search stage in place of running the Simulink simulation. The MATLAB reference provides equivalent results and improves simulation speed because it runs faster than the Simulink simulation. The results of the MATLAB SSB search is passed to both MATLAB and Simulink implementations of SIB1 recovery, and the output grids are directly compared. The Simulink SIB1 grid is decoded by one of two methods. The default option uses the `nrhdlSIB1Recovery` model to simulate the hardware accelerators for CORESET0 and SIB1 decoding. The second option uses a MATLAB-only decode algorithm. When the design is deployed to an SoC the first option reduces the computations performed by the embedded processor by offloading the calculations to the FPGA. The second option performs all processing in software allowing for the algorithm to be easily modified and updated without rebuilding the FPGA bitstream.

SIB1 Recovery Simulation

Use the `runSIB1RecoveryModelFR2` script to run a SIB1 recovery simulation. The script displays its progress at the MATLAB command prompt, and produces plots of inputs and outputs for analysis. The script also supports multiple simulation cases. The full set of cases, and their parameters, are shown.

Simulation Case	SSB Pattern	Subcarrier Spacing Common	PDCCH Config SIB1	SNR dB
"SimCase 1"	"Case D"	120	41	10
"SimCase 2"	"Case E"	120	20	2
"SimCase 3"	"Case A"	15	116	5
"SimCase 4"	"Case C"	30	4	20

This example shows the results of running "SimCase 1". The resource grids produced by MATLAB and Simulink are displayed along with their relative mean squared error (MSE). This comparison verifies that the Simulink implementation closely matches the MATLAB reference. The grid plots are labelled to highlight the decoded PDCCH and PDSCH. The final stage of the script decodes CORESET0, displays the DCIs, and decodes SIB1. The result of the SIB1 decode is displayed, and the SIB1 bits from MATLAB and Simulink are compared to verify that they match.

```
runSIB1RecoveryModelFR2;

Generating test waveform.
Searching for SSBs using MATLAB reference.
```

NCellID2	timingOffset	pssCorrelation	pssEnergy	frequencyOffset
0	8800	2.0719	2.3012	5504
0	10992	0.40044	0.59184	2703
0	17568	0.24485	0.44337	5901
0	19760	0.73269	0.93887	6647
0	24144	0.52003	0.7229	3062
0	26336	0.4873	0.72059	5132

```
        0              32944            0.90865            1.1301            5085
        0              35136            0.5211            0.72334            5414

Recover the SIB1 grid using MATLAB reference.
Decoding the SSB using the MATLAB reference.
Recovering the SIB1 grid using the MATLAB reference.
Recover the SIB1 grid using Simulink model.
Running nrhdlSIB1Recovery.slx
### Starting serial model reference simulation build.
### Model reference simulation target for nrhdlCORESET0DecodingCore is up to date.
### Model reference simulation target for nrhdlDDCCore is up to date.
### Model reference simulation target for nrhdlLDPCDecodingChainCore is up to date.
### Model reference simulation target for nrhdlPolarDecodingChainCore is up to date.
### Model reference simulation target for nrhdlSIB1DemodulationCore is up to date.
### Model reference simulation target for nrhdlSSBDecodingCore is up to date.
### Model reference simulation target for nrhdlSSBDetectionCore is up to date.

Build Summary

0 of 7 models built (7 models already up to date)
Build duration: 0h 0m 1.9174s
....................
MATLAB and Simulink grids relative MSE : -56.2924 dB

Extracting CORESET0 candidates from the SIB1 grid.
Decoding CORESET0 candidates using MATLAB reference.
Decoding CORESET0 candidates using Simulink.
Running nrhdlSIB1Recovery.slx
### Starting serial model reference simulation build.
### Model reference simulation target for nrhdlCORESET0DecodingCore is up to date.
### Model reference simulation target for nrhdlDDCCore is up to date.
### Model reference simulation target for nrhdlLDPCDecodingChainCore is up to date.
### Model reference simulation target for nrhdlPolarDecodingChainCore is up to date.
### Model reference simulation target for nrhdlSIB1DemodulationCore is up to date.
### Model reference simulation target for nrhdlSSBDecodingCore is up to date.
### Model reference simulation target for nrhdlSSBDetectionCore is up to date.

Build Summary

0 of 7 models built (7 models already up to date)
Build duration: 0h 0m 2.5059s
....................
DCI from MATLAB:
                    RIV: 528
                TDDIndex: 0
    VRBToPRBInterleaving: 0
              ModCoding: 0
                      RV: 0
              SIIndicator: 0
                Reserved: 0

DCI from Simulink:
                    RIV: 528
                TDDIndex: 0
    VRBToPRBInterleaving: 0
              ModCoding: 0
                      RV: 0
              SIIndicator: 0
```

```
            Reserved: 0

DCI successfully decoded from Simulink grid with hardware acceleration

Extracting LDPC codeword from the SIB1 grid.
Decoding SIB1 using MATLAB reference.
Decoding SIB1 using Simulink.
Running nrhdlSIB1Recovery.slx
### Starting serial model reference simulation build.
### Model reference simulation target for nrhdlCORESET0DecodingCore is up to date.
### Model reference simulation target for nrhdlDDCCore is up to date.
### Model reference simulation target for nrhdlLDPCDecodingChainCore is up to date.
### Model reference simulation target for nrhdlPolarDecodingChainCore is up to date.
### Model reference simulation target for nrhdlSIB1DemodulationCore is up to date.
### Model reference simulation target for nrhdlSSBDecodingCore is up to date.
### Model reference simulation target for nrhdlSSBDetectionCore is up to date.

Build Summary

0 of 7 models built (7 models already up to date)
Build duration: 0h 0m 1.207s
..........
SIB1 successfully decoded from Simulink grid with hardware acceleration
SIB1 bits from MATLAB and Simulink match
```


Spectrogram of Generated Waveform

SIB1 Grid from MATLAB Reference

SIB1 Grid from Simulink
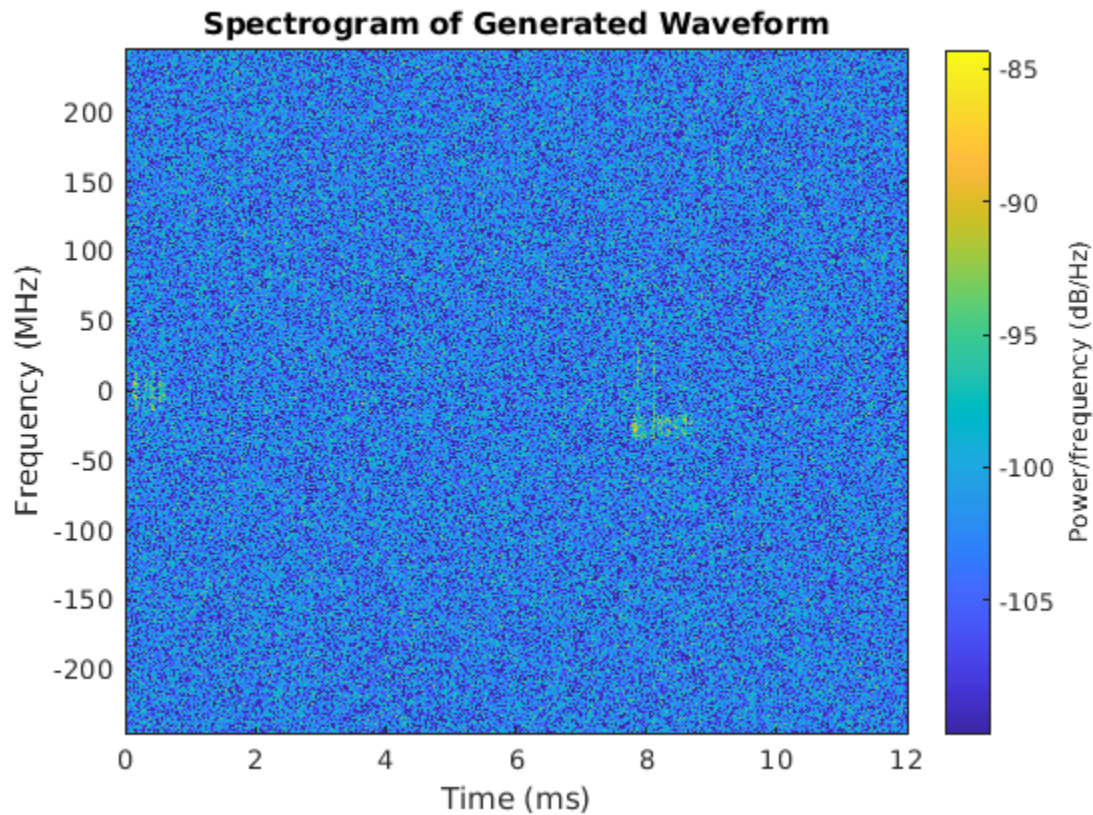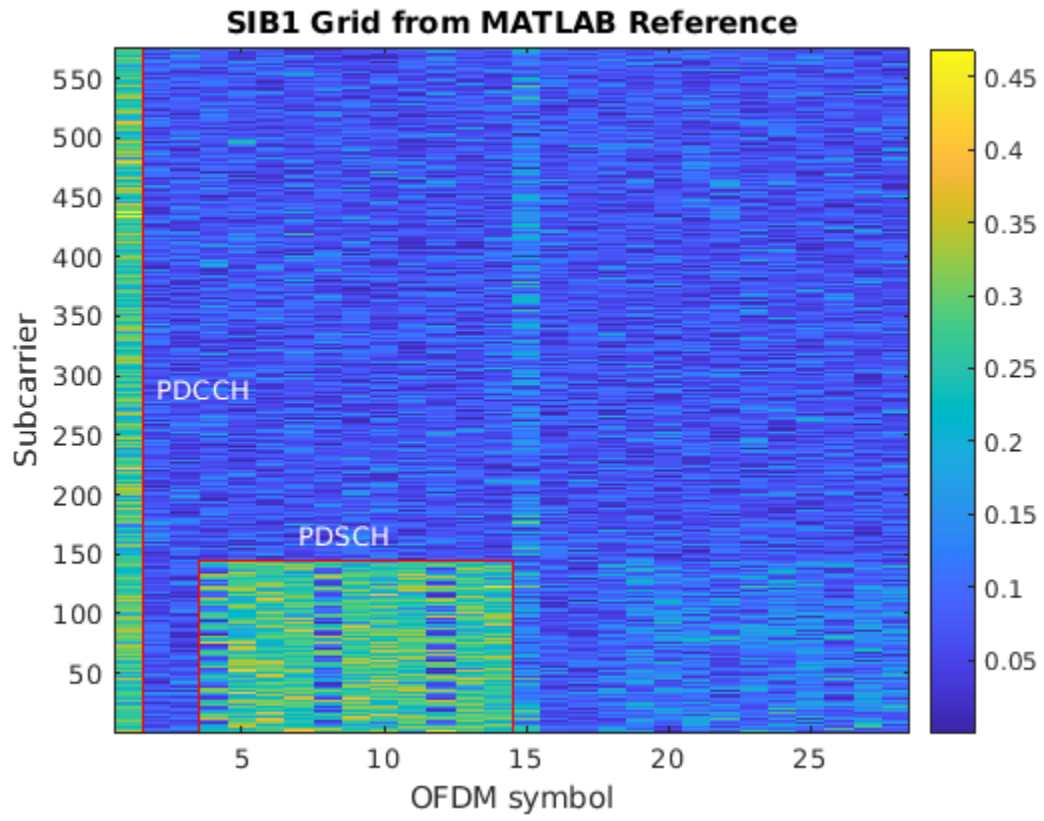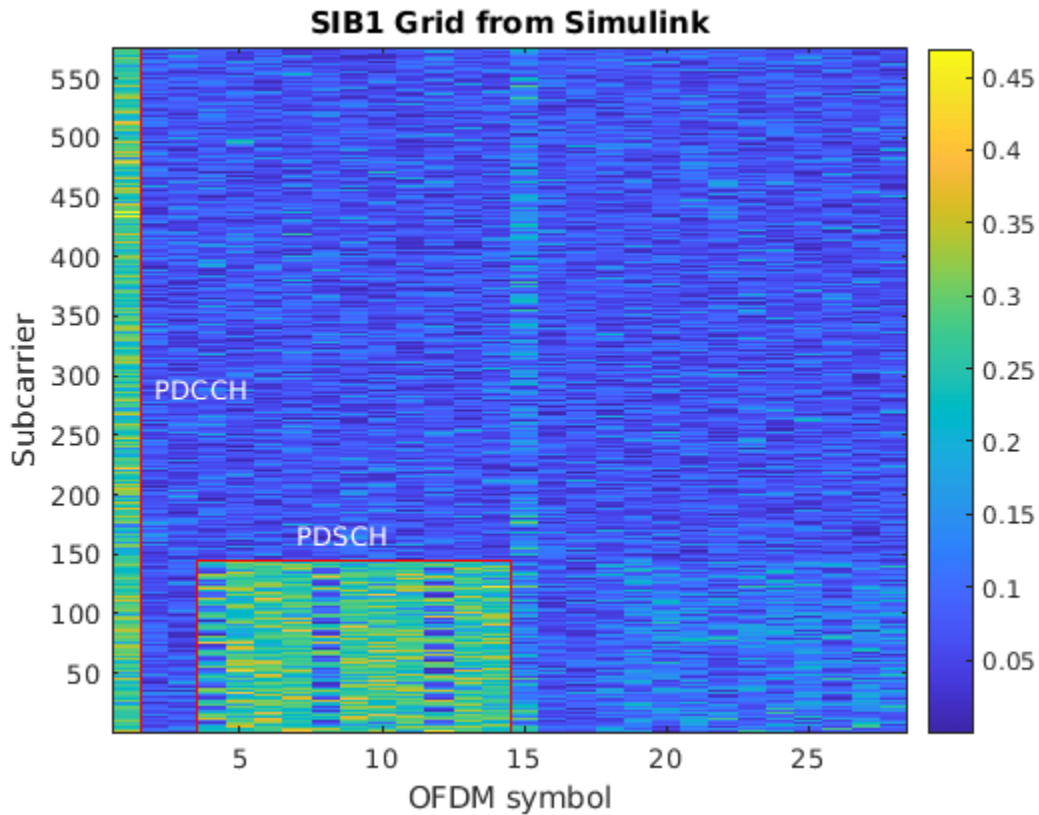
HDL Code Generation and Implementation Results

To generate the HDL code for this example, you must have the HDL Coder™ product. Use the `makehdl` and `makehdltb` commands to generate HDL code and an HDL test bench for the `nrhdlSIB1Recovery/SIB1 Recovery` subsystem. The resulting HDL code is synthesized for a Xilinx® Zynq® UltraScale+ RFSoC ZCU111 evaluation board. The table shows the post place and route resource utilization results. The design meets timing with a clock frequency of 245.76 MHz.

Resource	Usage
Slice Registers	134191
Slice LUTs	93069
RAMB18	418
RAMB36	39
DSP48	216

See Also

Related Examples

- "NR HDL Cell Search" on page 5-88
- "NR HDL MIB Recovery" on page 5-56

More About

- "NR HDL Receiver Performance" on page 5-5

NR HDL MIB Recovery

This example shows how to design a 5G NR synchronization signal block (SSB) decoding and master information block (MIB) recovery model optimized for HDL code generation and hardware implementation.

Introduction

The Simulink® models described in this example are fixed-point HDL optimized implementations of SSB decoding and MIB recovery for 5G NR frequency range 1 (FR1). This example is one of a related set, for more information see "NR HDL Reference Applications Overview" on page 5-2.

MIB recovery requires SSB detection, demodulation, and decoding. This example focuses on SSB decoding. SSB detection and demodulation are described in the "NR HDL Cell Search" on page 5-88 example. This example introduces the SSB decoding Simulink model and uses the MATLAB reference to generate test input and verify the behavior of the model. Then, the example describes a Simulink model that combines SSB detection, demodulation, and decoding to recover MIB from a baseband waveform.

After an SSB has been detected and demodulated, it needs to be decoded to extract the MIB contents. SSB decoding requires demodulation reference signal (DMRS) search, channel estimation and phase equalization, and broadcast channel (BCH) decoding steps as shown in the figure below.
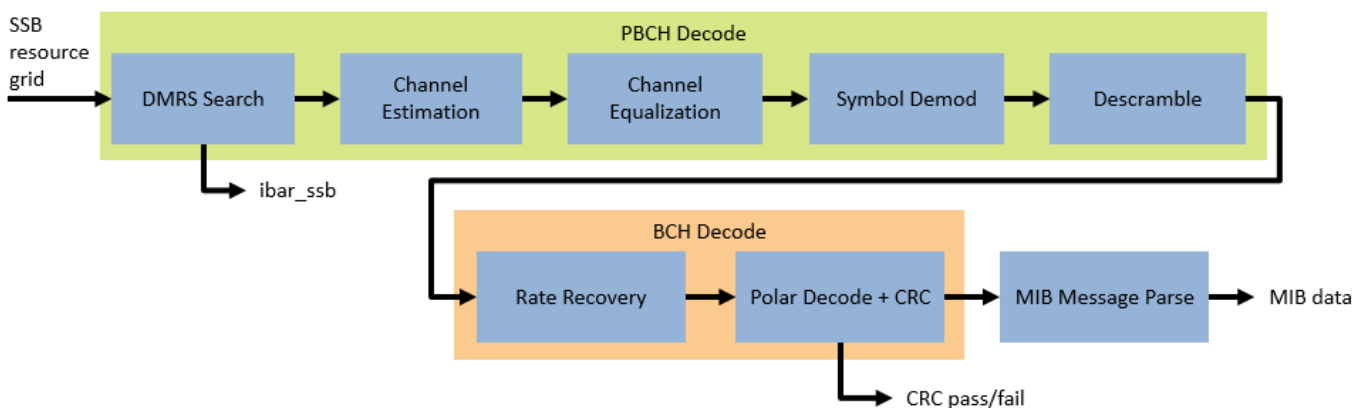
File Structure

This example uses these files.

Simulink models

- `nrhdlSSBDecoding.slx`: This Simulink model simulates the behavior of the SSB decoding step of the MIB recovery process.
- `nrhdlMIBRecovery.slx`: This Simulink model combines the processing of the SSB detector and the SSB decoder into an integrated model illustrating the complete MIB recovery process.
- `nrhdlSSBDecodingCore.slx`: This model implements the SSB decoding algorithm.
- `nrhdlPolarDecodingChainCore.slx`: This model implements the common polar decoding chain.

- `nrhdlSSBDetectionFR1Core.slx`: This model implements the SSB detection algorithm.
- `nrhdlDDCFR1Core.slx`: This model implements a DDC to create sample streams for SIB1 and SSBs.

Simulink data dictionary

- `nrhdlReceiverData.sldd`: This Simulink data dictionary contains bus objects that define the buses contained in the example models.

MATLAB code

- `runSSBDecodingModel.m`: This script uses the MATLAB reference to implement the cell search algorithm, then runs the `nrhdlSSBDecoding` Simulink model. The script verifies the operation of the model using 5G toolbox and the MATLAB reference code.
- `runMIBRecoveryModel.m`: This script uses the MATLAB reference to perform the search mode of the SSB detection algorithm, then runs the `nrhdlMIBRecovery` Simulink model. The script verifies the operation of the model using 5G toolbox and the MATLAB reference code.
- `nrhdlexamples`: Package containing the MATLAB reference code and utility functions for verifying the implementation models.

NR HDL SSB Decoding

This figure shows the `nrhdlSSBDecoding` model. The top level of the model reads the signals from the MATLAB base workspace, passes them to the SSB Decoding subsystem, and writes the outputs back to the workspace. The ParseMIB subsystem takes the pbchPayload and interprets the bit fields to produce the MIB parameter outputs.

Copyright 2020-2021 The MathWorks, Inc.
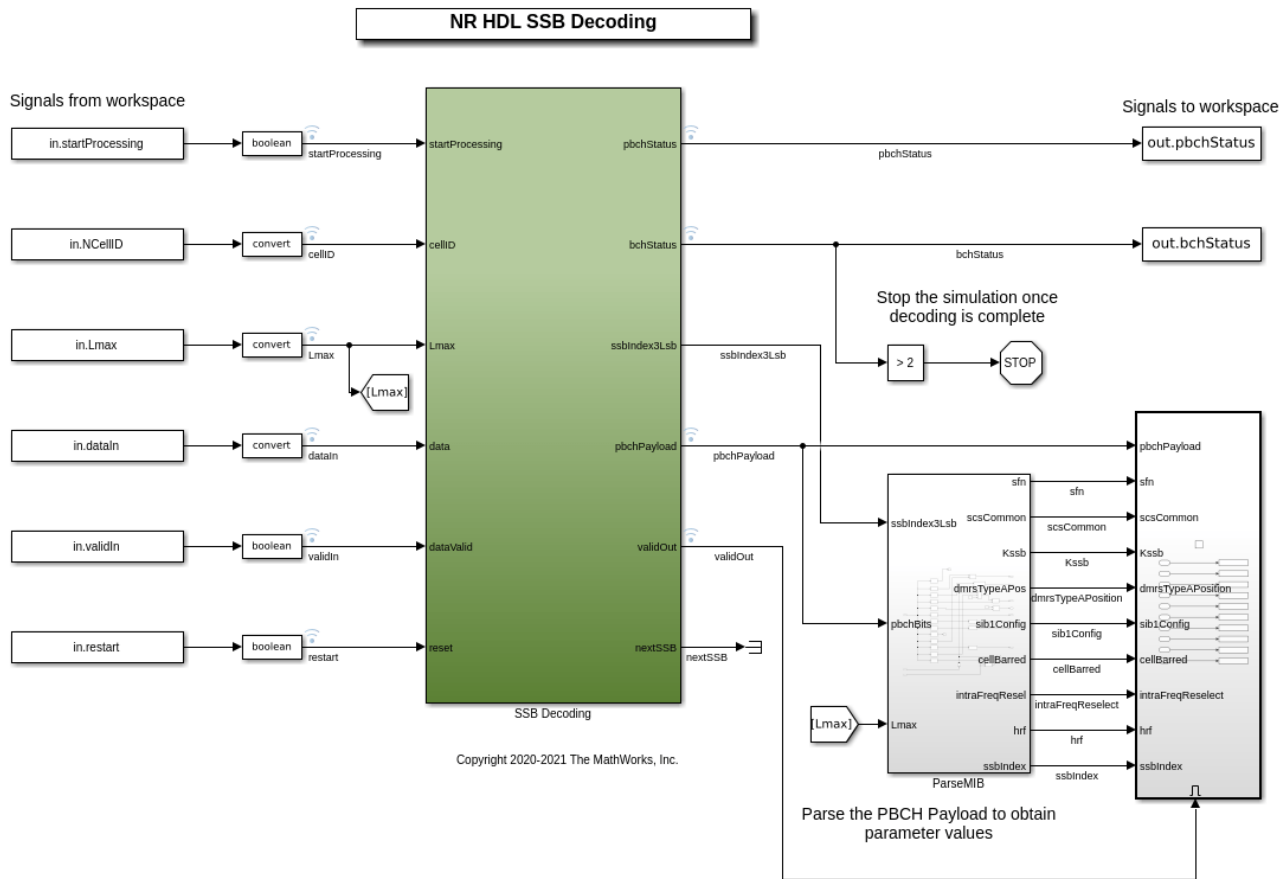
SSB Decoding Subsystem

The `SSB Decoding` subsystem references the `nrhdlSSBDecodingCore` and `nrhdlPolarDecodingChainCore` models. The subsystem performs DMRS search, channel estimation and equalization, QPSK symbol demodulation, descrambling, rate recovery, polar decoding, and CRC decoding. This processing is split over two models to allow for the `nrhdlPolarDecodingChainCore` to be shared between the SSB decoding and SIB1 CORESET0 decoding in the "NR HDL SIB1 Recovery" on page 5-8 example. This section describes the inputs and outputs of that model.
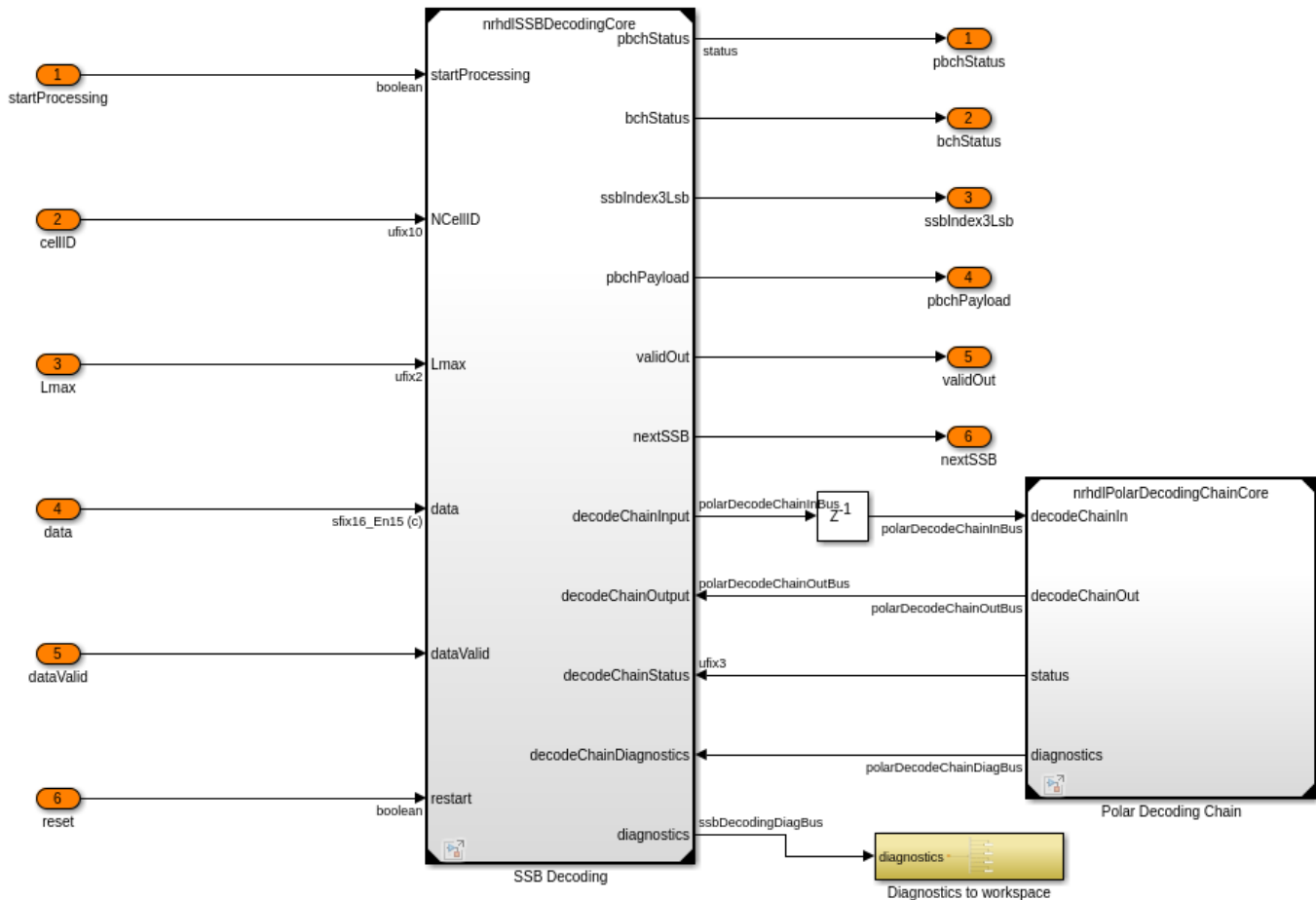
Inputs

- *startProcessing*: 1-bit control signal which indicates when all data has been written and that cellID and Lmax are valid.

- *cellID*: 10-bit unsigned number which provides cell ID number for the detected SSB.

- *Lmax*: 2-bit unsigned number which indicates the maximum number of SSBs in a burst. A value of 0 indicates 4 SSBs and a value of 1 indicates 8 SSBs.

- *data*: 16-bit signed complex-valued signal carrying the 4 OFDM symbols of the SSB.

- *dataValid*: 1-bit control signal to validate data.

- *reset*: 1-bit control signal to reset the processing.

Outputs

- *pbchStatus*: 2-bit unsigned value indicating the progress of the PBCH decoding operation. See below for more information on the possible values of this signal.

- *bchStatus*: 3-bit unsigned value indicating the progress of the BCH decoding operation. See below for more information on the possible values of this signal.

- *ssbIndex3Lsb*: 3-bit unsigned value that is the 3 least significant bits of the SSB index calculated by the DMRS search process and Lmax.

- *pbchPayload*: 32-bit unsigned value that contains the MIB and additional PBCH timing data.
- *validOut*: 1-bit control signal to validate ssbIndex3Lsb and pbchPayload.
- *nextSSB*: 1-bit control signal to indicate when the core can begin processing the next SSB. Can be used to pace inputs for back-to-back SSB decodes.

PBCH Status Signal States

- `0`: idle
- `1`: reading in data for SSB grid
- `2`: performing DMRS search
- `3`: performing PBCH symbol demodulation

BCH Status Signal States

- `0`: idle
- `1`: performing rate recovery
- `2`: performing polar decoding
- `3`: CRC error (end state)
- `4`: CRC pass, MIB detected (end state)

SSB Decoding Model

This diagram shows the top level of the `nrhdlSSBDecodingCore` model. The input data is 4 OFDM symbols containing the synchronization signal block (SSB), with the values scaled within the range +/-1. The model starts processing when all of the SSB data has been input to the model and startProcessing is asserted. The startProcessing signal also indicates that the NCellID and Lmax inputs are valid.

Copyright 2020-2021 The MathWorks, Inc.
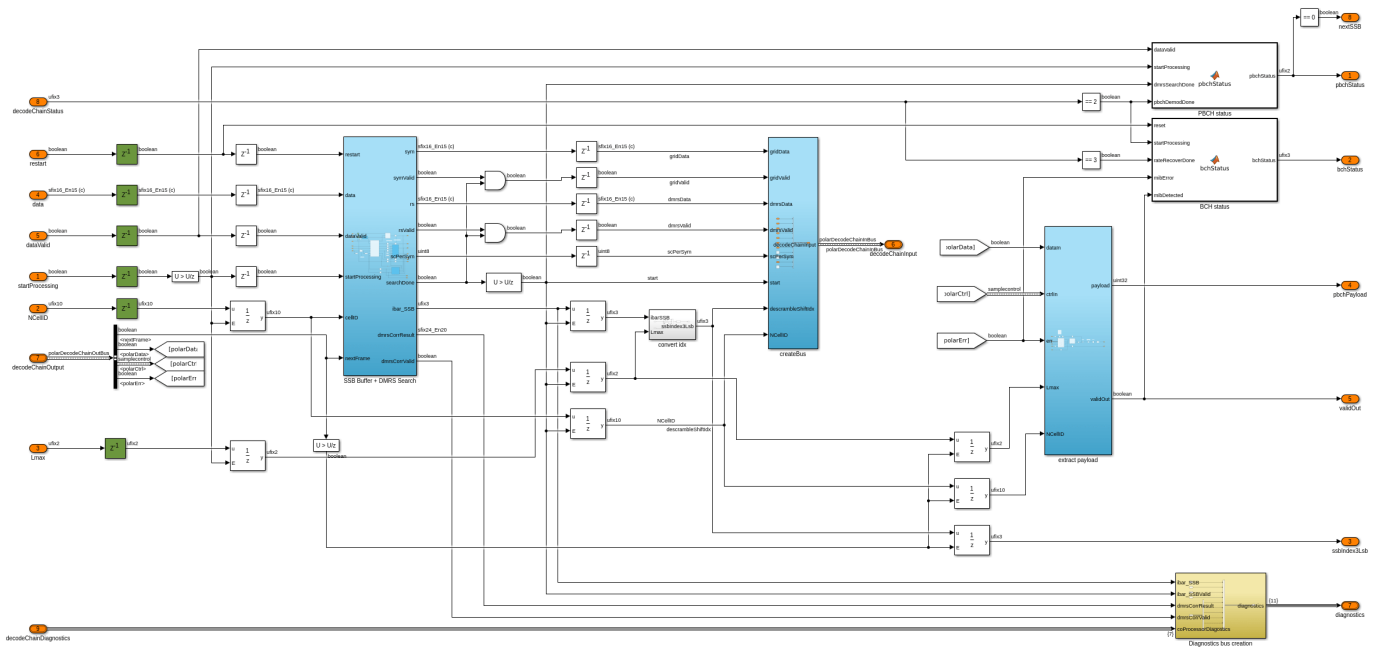
The `SSB Buffer + DMRS Search` subsystem performs the DMRS search. Incoming data is stored in a RAM buffer where it is held until startProcessing is asserted, indicating that all required

information is available to start the DMRS search process. The DMRS search reads the DMRS symbols from the RAM and correlates with the 8 possible DMRS sequences, selecting the strongest correlation value to determine ibar_SSB. Once the DMRS search has been completed ibar_SSB is used to generate the reference DMRS required for channel estimation. The reference DMRS is output from the model along with the received PBCH symbols and associated DMRS. This is used to drive the `nrhdlPolarDecodingChainCore` model.
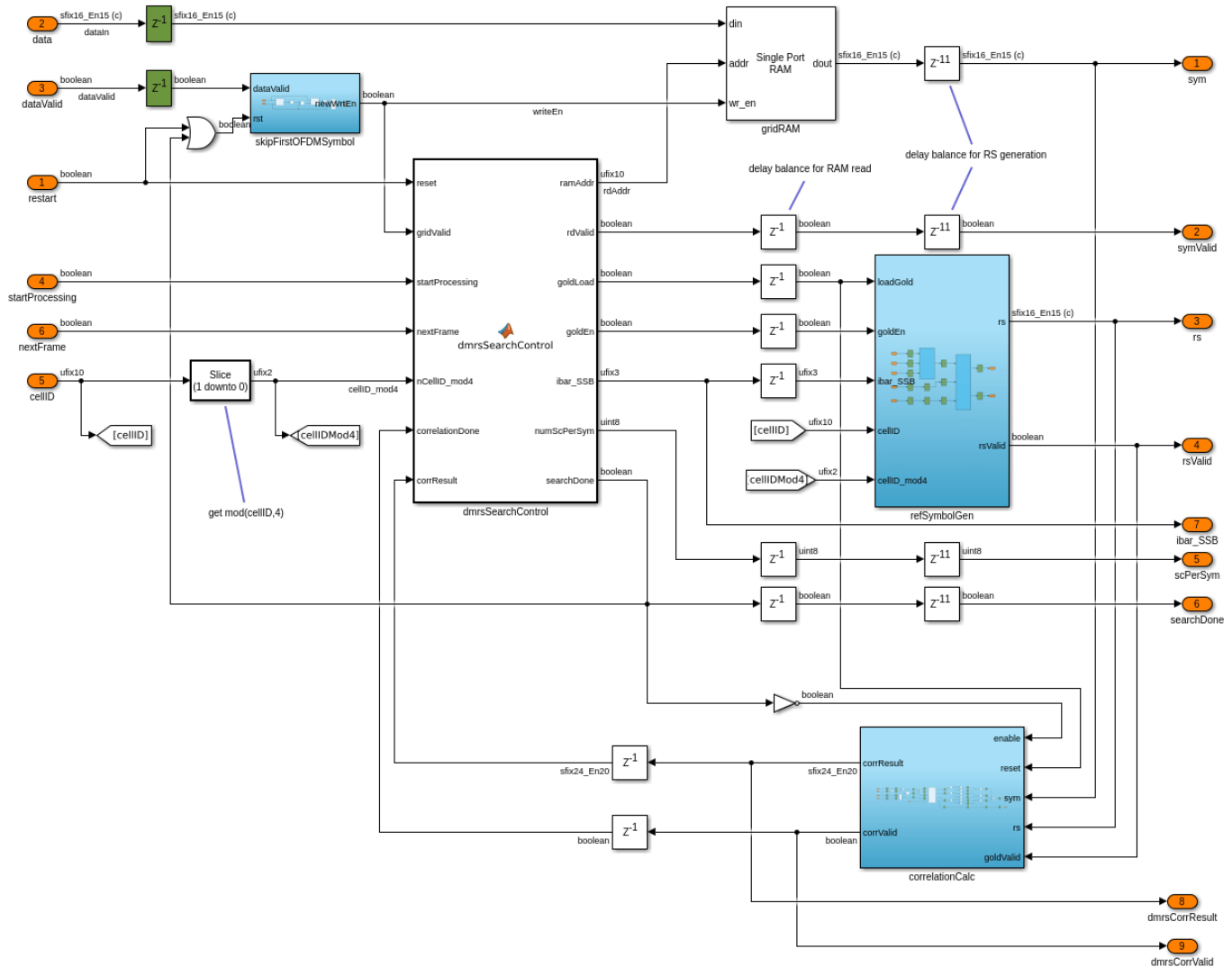
The extract payload subsystem performs descrambling and deinterleaving of the payload bits returned from the `nrhdlPolarDecodingChainCore` model.
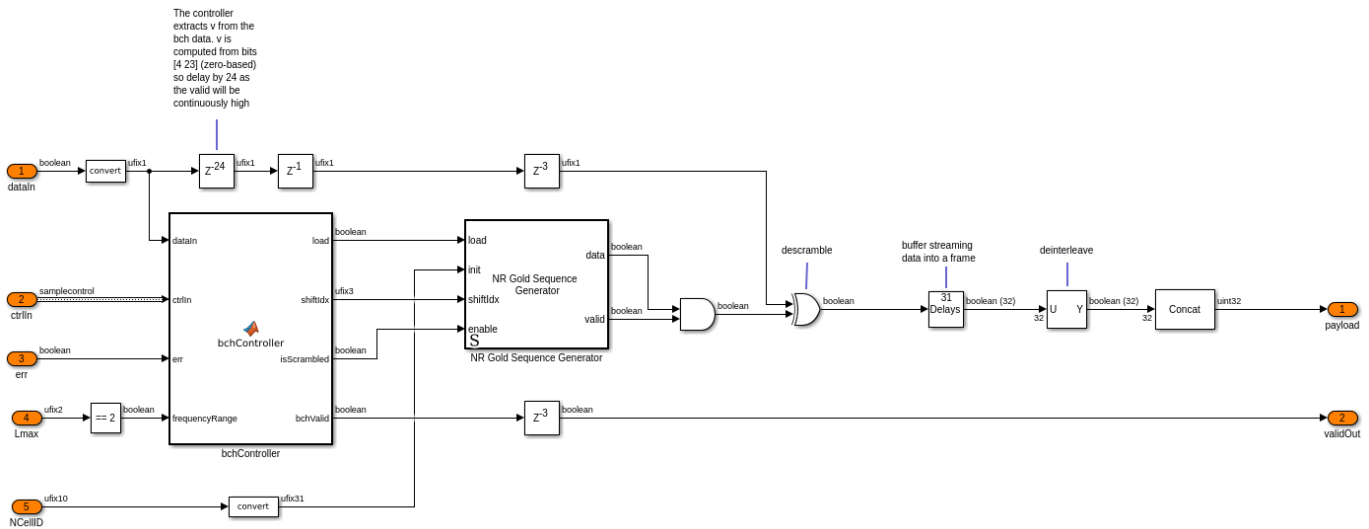
Polar Decoding Chain Model

The `nrhdlPolarDecodingChain` model performs channel estimation and equalization, QPSK symbol demodulation, descrambling, rate recovery, and polar and CRC decoding. This signal processing chain is common for the decoding of both the BCH and downlink control information (DCI).
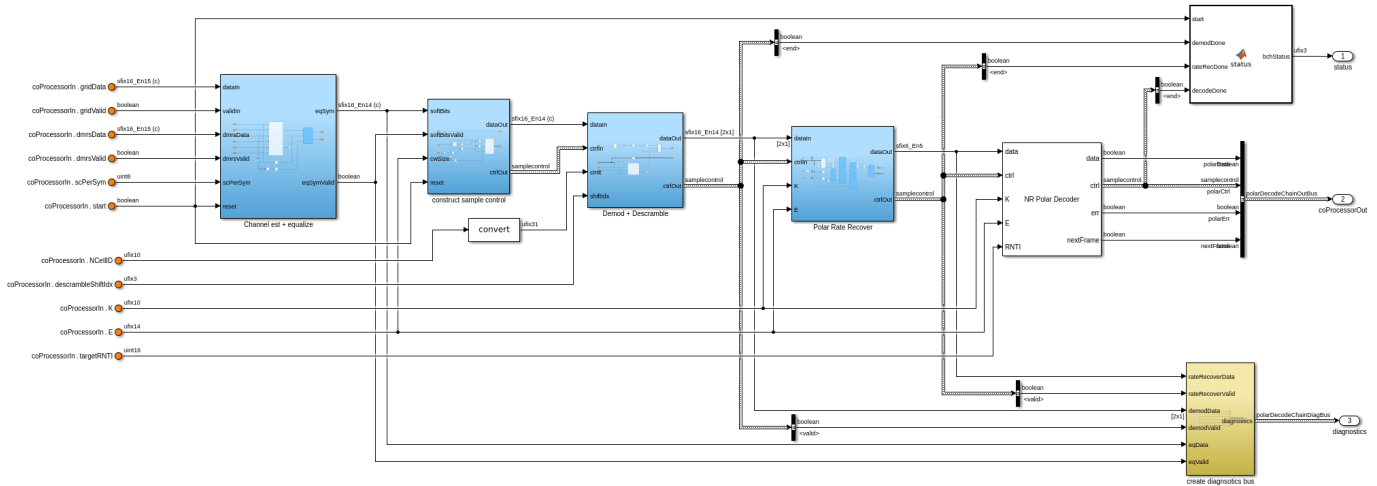
The `Channel est + equalize` subsystem performs channel estimation using the received data and the reference DMRS. The channel estimate applies linear interpolation between DMRS locations within an OFDM symbol, but does not average across time in case of any residual carrier frequency offset. Phase equalization of the QPSK symbols is then performed, followed by QPSK demodulation and descrambling, using the descrambling shift index and NCellID inputs to seed the scrambler. Subsequent processing performs rate recovery, polar decoding, and CRC decoding of the descrambled data. The `Polar Rate Recover` subsystem includes signal scaling and wordlength reduction to prepare the data for polar decoding. The scaled, rate-recovered soft bits are then passed to the NR Polar Decoder block, which also performs CRC decoding. The err output port from the NR Polar Decoder block indicates if decoding was successful or encountered any errors.

SSB Decoding Simulation Setup

The block diagram shows the simulation setup implemented by this example. 5G Toolbox™ functions are used to generate a test waveform. MATLAB reference code for the SSB detector is then used to search for and demodulate the strongest SSB within the waveform. This result provides test input for the SSB decoding stage. The test data is passed to both MATLAB and Simulink implementations, and the outputs are compared to verify the operation of the Simulink model.
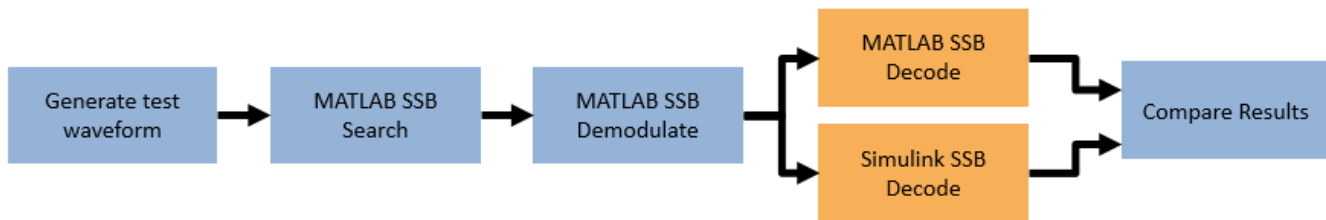
SSB Decoding Simulation

Use the `runSSBDecodingModel` script to run an SSB decoding simulation. The script displays its progress at the MATLAB command prompt. The final results of decoding the SSB in MATLAB and Simulink are displayed, showing that they match exactly. Plots of the DMRS search correlation strength and the equalized PBCH QPSK symbols show that the signals from MATLAB and Simulink match closely.

```
runSSBDecodingModel;

Generating test waveform.
Selected Simulation case:
```

Simulation Case	SSB Pattern	Subcarrier Spacing Common	PDCCH Config SIB1	SNR dB
"SimCase 1"	"Case C"	30	164	50

```
Searching for SSBs using the MATLAB reference.
Demodulating the strongest SSB using the MATLAB reference.
Decoding the SSB using the MATLAB reference.
MIB successfully decoded by MATLAB reference
Decoding the SSB using the Simulink model.
Running nrhdlSSBDecoding.slx
### Starting serial model reference simulation build
### Model reference simulation target for nrhdlPolarDecodingChainCore is up to date.
### Model reference simulation target for nrhdlSSBDecodingCore is up to date.

Build Summary

0 of 2 models built (2 models already up to date)
Build duration: 0h 0m 0.59046s
..........
MIB successfully decoded by Simulink model
 MATLAB decoded information
    pbchPayload: 17637376
        ssbIndex: 4
            hrf: 0
            err: 0
            mib: [1×1 struct]
```

```
Simulink decoded information
    pbchPayload: 17637376
      ssbIndex: 4
           hrf: 0
           err: 0
           mib: [1×1 struct]

MATLAB decoded MIB parameters
                    NFrame: 0
    SubcarrierSpacingCommon: 30
                     k_SSB: 0
          DMRSTypeAPosition: 3
            PDCCHConfigSIB1: 164
                 CellBarred: 0
        IntraFreqReselection: 0

Simulink decoded MIB parameters
                    NFrame: 0
    SubcarrierSpacingCommon: 30
                     k_SSB: 0
          DMRSTypeAPosition: 3
            PDCCHConfigSIB1: 164
                 CellBarred: 0
        IntraFreqReselection: 0
```
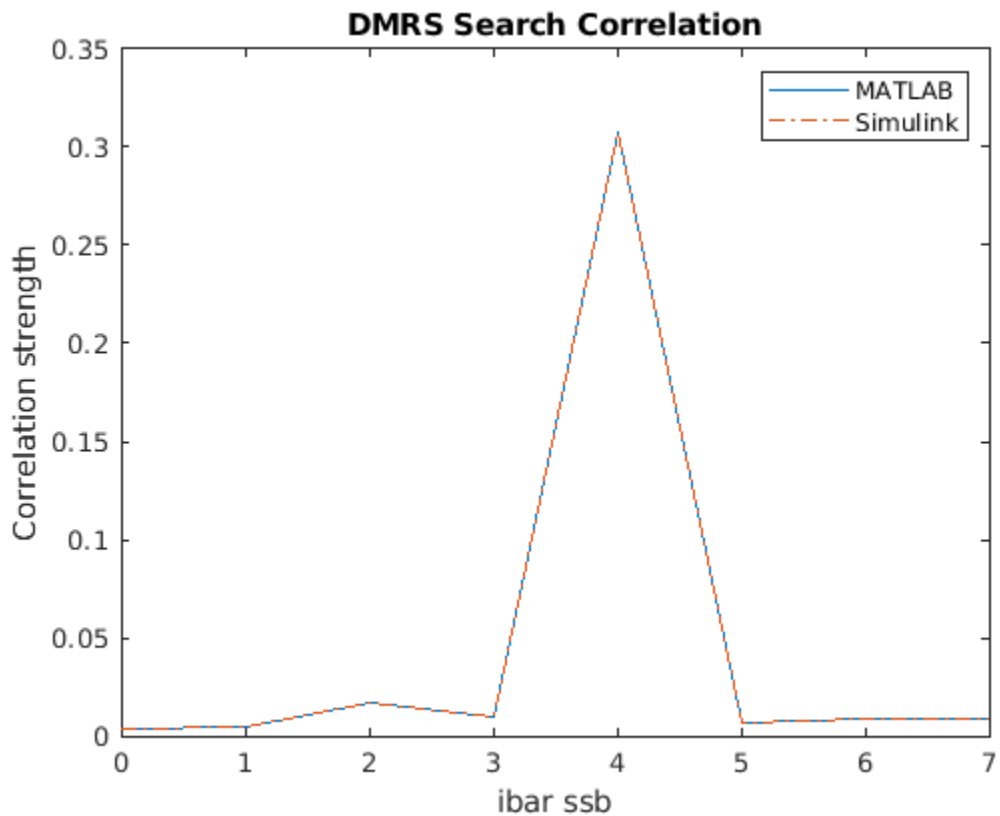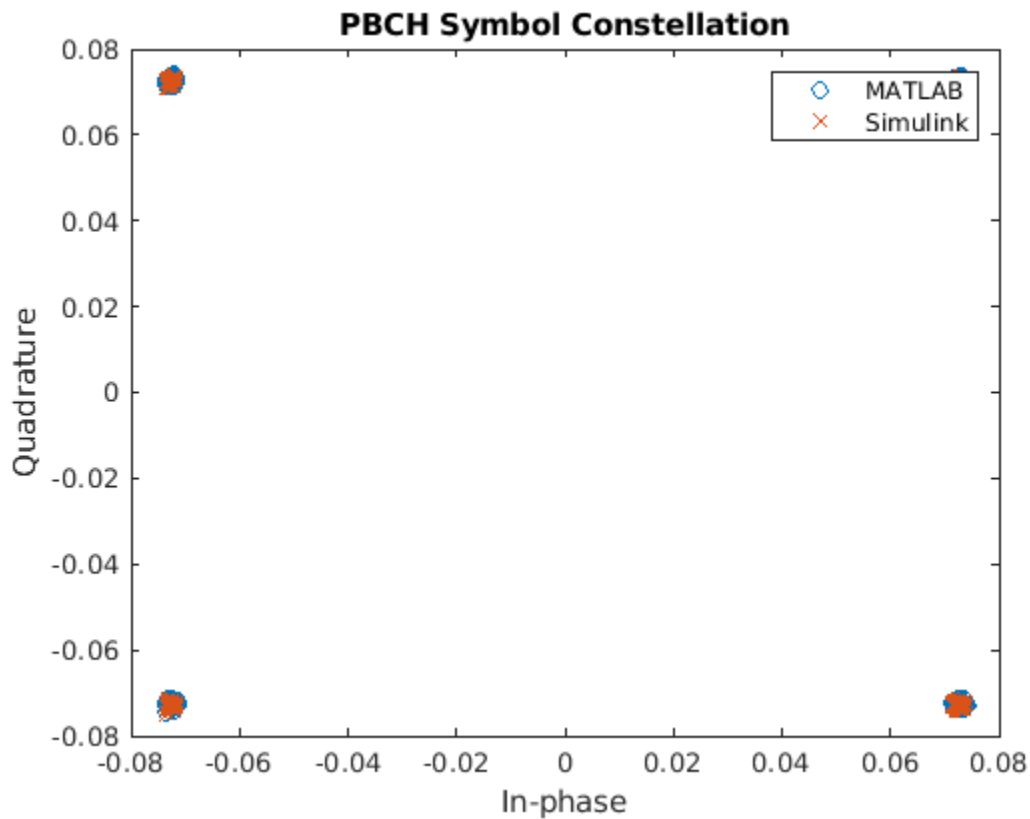
MIB Recovery Model

The `nrhdlMIBRecovery` model connects the for SSB decoding and SSB detection models to create a complete MIB recovery implementation. This model can be used to recover MIB from baseband 5G waveforms. The script `runMIBRecoveryModel` can be used to run this model and compare against the MATLAB reference. To reduce the processing time required the cell search part of the algorithm is performed in MATLAB then, once the strongest SSB has been determined, the Simulink model is used to re-acquire, demodulate, and decode the SSB.

The status signal from the detector is used to start the SSB decoder when it has reached state 8, indicating that demodulation is complete, SSS has been found, and the demodulated grid has been output. When the SSB decoder has the demodulated grid and received the startProcessing signal it will decode the SSB, outputting the PBCH payload which is then parsed to extract the MIB data.
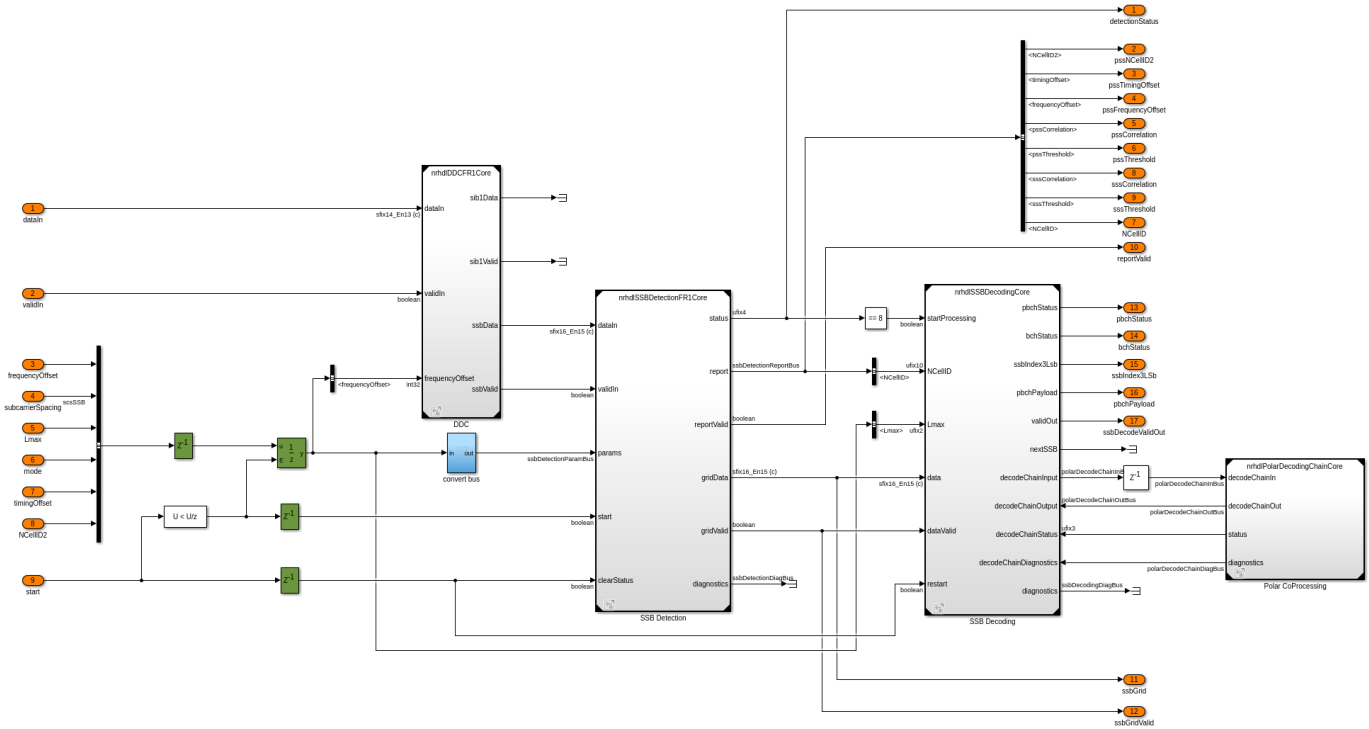
HDL Code Generation and Implementation Results

To generate the HDL code for this example, you must have an HDL Coder™ license. Use the `makehdl` and `makehdltb` commands to generate HDL code and an HDL test bench for `nrhdlSSBDecoding/SSB Decoding` or `nrhdlMIBRecovery/MIB Recovery` subsystems. The resulting HDL code was synthesized for a Xilinx® Zynq®-7000 ZC706 evaluation board. The table shows the post place and route resource utilization results. The design meets timing with a clock frequency of 150 MHz.

Resource utilization for nrhdlSSBDecoding model:

Resource	Usage
Slice Registers	9114
Slice LUTs	11635
RAMB18	8
RAMB36	5
DSP48	37

Resource utilization for nrhdlMIBRecovery model:

Resource	Usage
Slice Registers	45540
Slice LUTs	32186
RAMB18	21
RAMB36	5

DSP48   255

See Also

Related Examples
- "NR HDL Cell Search" on page 5-88

More About
- "NR HDL Receiver Performance" on page 5-5

NR HDL Downlink Receiver MATLAB Reference

This example shows how to model a 5G NR cell search, MIB and SIB1 recovery hardware algorithm in MATLAB® as a step towards developing a Simulink® HDL implementation of a downlink receiver.

This MATLAB code serves as a reference to verify the Simulink models of the hardware implementations in the "NR HDL Cell Search" on page 5-88, "NR HDL MIB Recovery" on page 5-56, "NR HDL SIB1 Recovery" on page 5-8, and "NR HDL SIB1 Recovery for FR2" on page 5-44 examples.

The NR HDL Downlink Receiver MATLAB Reference example bridges the gap between a mathematical algorithm and its hardware implementation by providing a MATLAB model of the algorithms that are implemented in hardware. The MATLAB reference is created to evaluate hardware-friendly algorithms and generate test vectors for verifying the Simulink fixed-point HDL optimized implementation. This example is one of a related set, for more information see "NR HDL Reference Applications Overview" on page 5-2.

Downlink Receiver Overview

A block diagram of the Downlink Receiver algorithm is shown. The algorithm detects, demodulates, and decodes 5G NR synchronization signal blocks (SSBs) and recovers SIB1. It is a hardware-friendly version of the corresponding steps in the "NR Cell Search and MIB and SIB1 Recovery" (5G Toolbox) example. At the top level, the algorithm consists of a Search Controller, an SSB Detector, an SSB Decoder, SIB1 grid demodulator and SIB1 decoder. This example explains each of these blocks in more detail and demonstrates the corresponding MATLAB reference functions, which are used to explore algorithms for hardware implementation and to verify the streaming fixed-point Simulink models. This example focuses on 5G NR frequency range 1 (FR1). See "NR HDL SIB1 Recovery for FR2" on page 5-44 for an example of how to use the MATLAB reference for FR2 SIB1 Recovery.
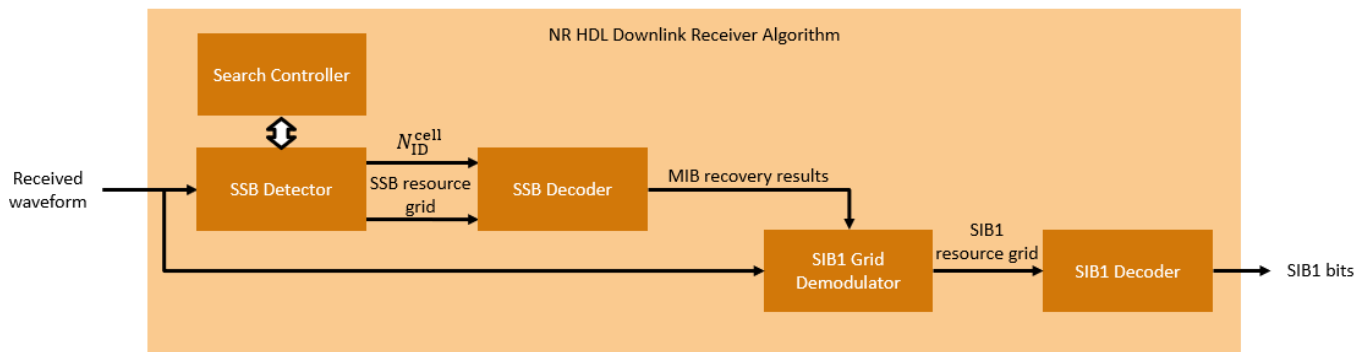
Cell Search

Cell search consists of carrier frequency recovery, primary synchronization signal (PSS) search, OFDM demodulation, and secondary synchronization signal (SSS) search. The Search Controller and the SSB Detector work together to perform these processing steps. The SSB Detector performs all of the high-speed signal processing tasks, making it well suited for FPGA or ASIC implementation. The Search Controller coordinates the search and operates at a low rate, making it well suited for software implementation on an embedded processor.

The algorithm starts by using the PSS to search for SSBs with subcarrier spacings of 15 kHz and 30 kHz across a range of coarse frequency offsets. The subcarrier spacing and coarse frequency offset

search ranges are configurable. If SSBs are detected, the receiver OFDM demodulates the resource grid of the SSB with the strongest PSS and determines its cell ID using the SSS. The residual fine frequency offset is corrected during the OFDM demodulation phase.
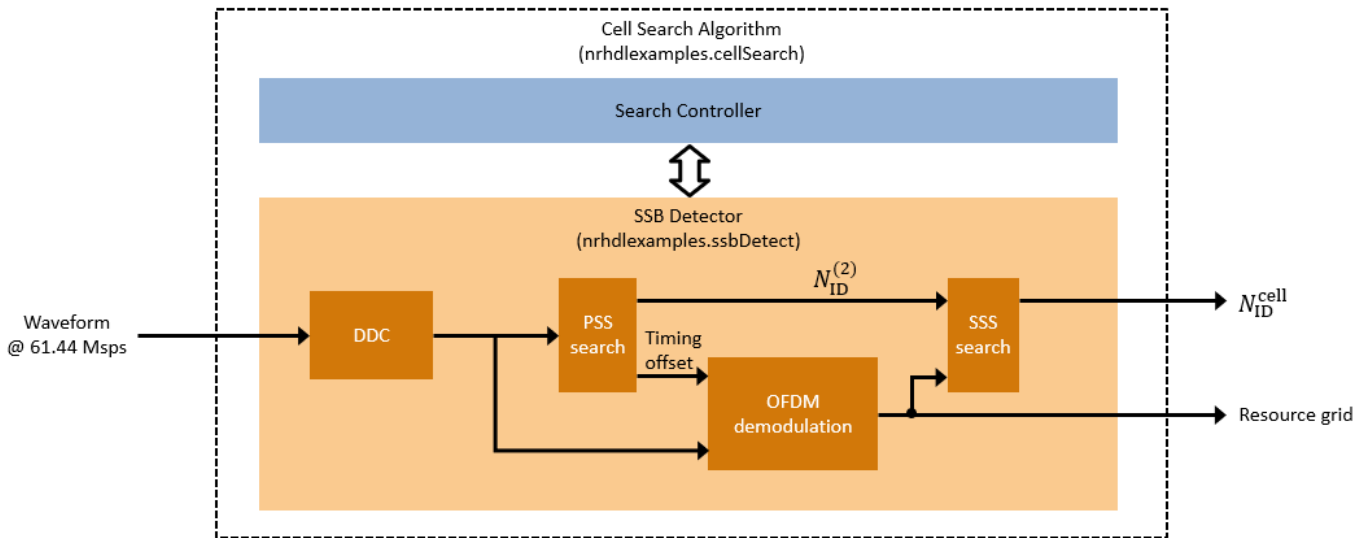
- *SSB Detector*: Searches for and OFDM-demodulates SSBs at a given carrier frequency offset and subcarrier spacing and measures the residual fine carrier frequency offset.
- *Digital Down Converter (DDC)*: Performs frequency translation to correct frequency offsets in the received waveform and then decimates the signal from 61.44 Msps to 7.68 Msps.
- *PSS search*: Searches for PSS symbols within the waveform.
- *OFDM demodulation*: OFDM-demodulates an SSB resource grid.
- *SSS search*: Searches for SSS and determines the overall cell ID.
- *Search Controller*: Coordinates the cell search by directing the SSB Detector to search for PSS symbols at different coarse frequency offsets and subcarrier spacings and to demodulate the SSB with the strongest PSS.

In the MATLAB reference, the `nrhdlexamples.cellSearch` function implements the cell search algorithm. This function implements the Search Controller shown in the diagram, and calls the `nrhdlexamples.ssbDetect` function, which implements the SSB Detector. The "NR HDL Cell Search" on page 5-88 example shows the streaming fixed-point Simulink HDL implementation of the SSB Detector. In the "5G NR MIB Recovery Using Analog Devices AD9361/AD9364" (Communications Toolbox Support Package for Xilinx Zynq-Based Radio) example, the SSB Detector is implemented in programmable logic while the Search Controller is implemented in software on the integrated processing system.

Search Controller

The Search Controller is responsible for coordinating the overall search. The algorithm follows these steps.

1   For each subcarrier spacing, step through each coarse frequency offset and use the SSB Detector to search for SSBs until one or more is detected. The coarse frequency offset step size is half the

subcarrier spacing. When SSBs are detected at a given frequency, record the residual fine carrier frequency offset of the strongest SSB that is returned.

2 Move to the next coarse frequency step and search for SSBs again. If the search detects SSBs, choose the coarse frequency offset that resulted in the smallest fine frequency offset measurement. Otherwise, pick the last coarse frequency offset.

3 Compute the total frequency offset by adding the coarse and fine frequency offsets together.

4 Use the SSB Detector to correct the frequency offset and perform one more search for SSBs.

5 Pick the SSB with the strongest PSS correlation. Use the SSB Detector in demodulation mode to find and demodulate the SSB and determine its cell ID.

SSB Detector

These diagrams show the SSB Detector structure for FR1, and the parameters and data passed to and from the Search Controller. The SSB Detector is subdivided into two functions: SSB Detector DDC (`nrhdlexamples.ssbDetectDDC`) and SSB Detection Search and Demod (`nrhdlexamples.ssbDetectSearchDemod`). The DDC accepts samples at 61.44 Msps and performs a frequency shift followed by decimation by a factor of 8 using halfband filters. The frequency offset, in Hz, is provided by the search controller and is used by the algorithm to compensate for both coarse and fine frequency offsets.
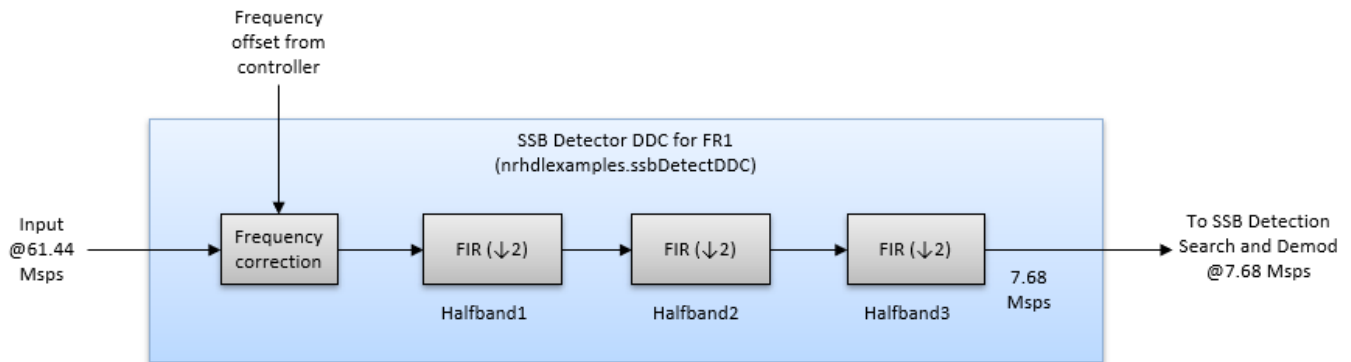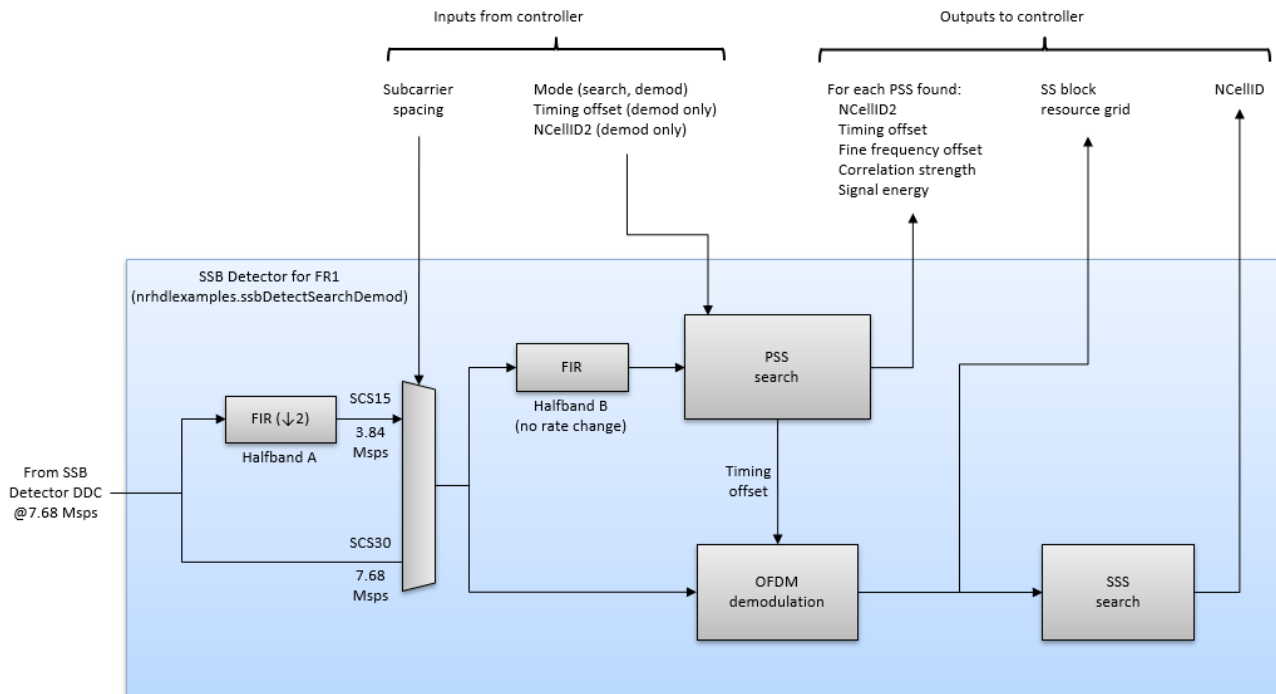
SSB Detection Search and Demod accepts samples at 7.68 Msps. For 30 kHz subcarrier spacing, it uses the samples at this rate. For 15 kHz subcarrier spacing, it decimates the input by a factor of two, operating at 3.84 Msps. SSB Detection Search and Demod has two modes of operation: search and demodulation.

In search mode, the function searches for SSBs at the specified subcarrier spacing using the PSS, and returns a list of those detected. For each SSB that is found, the function returns these parameters:

- *NCellID2*: Indicates which of the three possible PSS sequences (0,1, or 2) was detected.
- *timing offset*: The timing offset from the start of the waveform to the start of the SSB.
- *fine frequency offset*: The residual fine frequency offset in Hz measured by using the cyclic prefixes of all four OFDM symbols in the SSB.
- *correlation strength*: The measured PSS correlation level.
- *signal energy*: The total energy in the samples in which the PSS was detected.

In demodulation mode, the function attempts to find a specific SSB by using its timing offset and NCellID2. If the function finds the specified PSS, the receiver OFDM demodulates the SSB resource grid and attempts to detect its SSS. In demodulation mode, the function returns these results.

- Updated parameters for only the specified SSB if the PSS is found.
- The demodulated SSB resource grid if the PSS is found.
- The cell ID if the SSS is found.

The OFDM demodulator uses a 256-point FFT to demodulate the SSB resource grid, which contains 240 active subcarriers.

Timing Offsets

The cell search algorithm uses timing offsets to identify positions within the received waveform and intermediate signals. A timing offset is the number of samples from the start of the waveform to a given position, such as the start of an SSB. Timing offsets are given in samples at 61.44 Msps and wrap around every 20 ms, or 1228800 samples. In 5G NR, receivers can assume that the SS burst periodicity is 20 ms or less for cell search purposes, hence the reason for this choice of timing reference periodicity.

The figure shows two 5G waveforms with different SS burst periodicities (5 ms and 20 ms) and the receiver timing reference. The MATLAB reference can detect SSBs at any position within the received waveform. However, if the waveform is longer than 20 ms, ambiguity in the returned timing offsets exists because the timing reference wraps around every 20 ms. Additionally, the receiver can demodulate only SSBs that begin within the first 20 ms of the waveform.
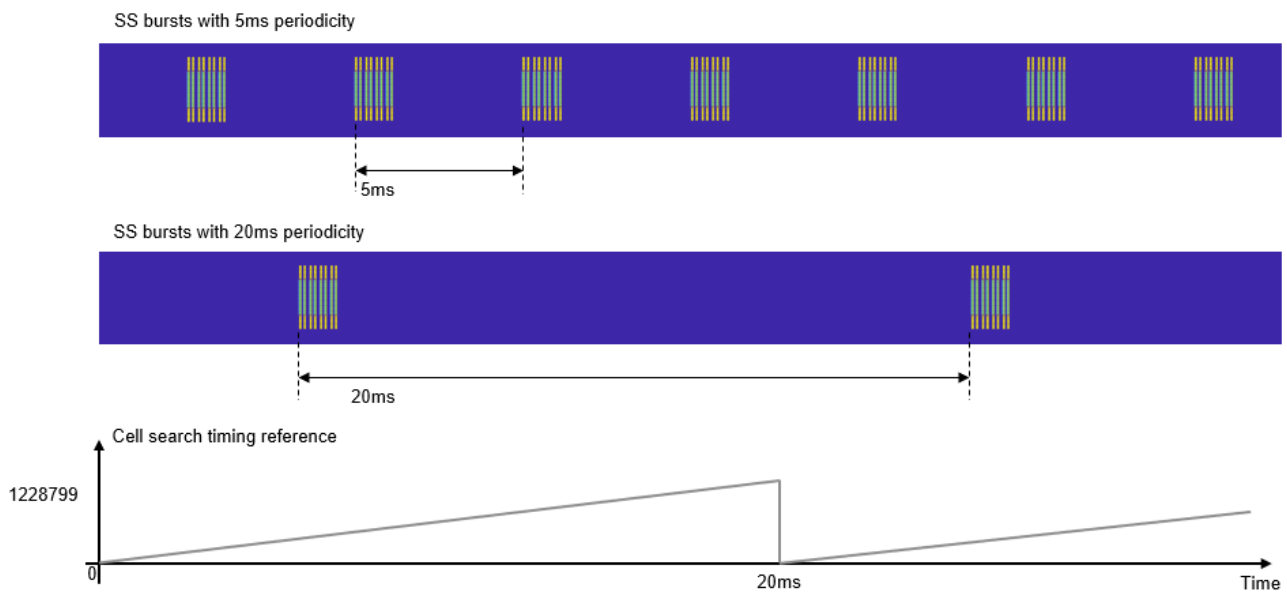
SSB Decoding

The diagram shows the structure of the SSB decoder, which is implemented by the `nrhdlexamples.ssbDecode` function. The algorithm takes the SSB resource grid from the OFDM demodulation phase of the SSB detector, processes it through PBCH and BCH decoding, and outputs MIB parameters and PBCH timing information.
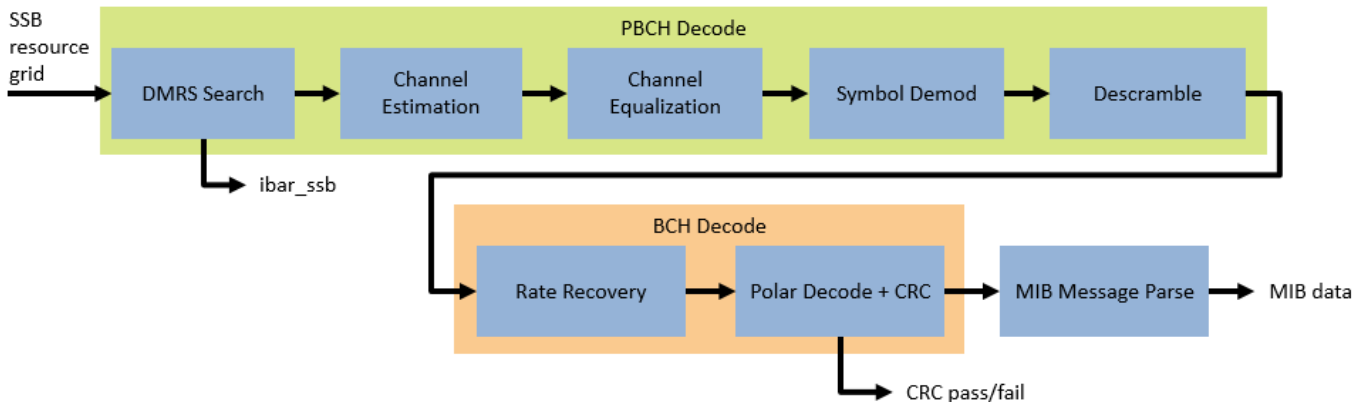
PBCH decoding takes the demodulated OFDM symbols of the resource grid and processes using these steps:

- *DMRS Search*: Searches for the index used for demodulation reference symbol (DMRS) generation.
- *Channel Estimation*: Calculates an estimate of the channel using the DMRS.
- *Channel Equalization*: Equalizes the received data using the channel estimate.
- *Symbol Demod*: Performs QPSK demodulation to get the PBCH soft bits.
- *Descramble*: Descrambles the soft bits.

BCH Decode then processes the descrambled soft bits to recover the MIB data using these steps:

- *Rate Recovery*: Combines repeated soft bits then performs scaling and quantization.
- *Polar Decode + CRC*: Performs polar decoding to get the message bits and CRC decoding to check for errors.
- *MIB Message Parse*: Interprets the decoded message bits to produce the MIB parameter outputs.

SIB1 Demodulation

The diagram shows the structure of the SIB1 Demodulator algorithm, which is implemented by the `nrhdlexamples.sib1Demodulate` function. The algorithm accepts samples at 61.44 MHz, and uses the results from the previous processing stages to locate and demodulate a grid containing CORESET0 and the scheduled SIB1 transmission. The MIB results are used to calculate the parameters of CORESET0, which includes the frequency offset, number of resource blocks, and the monitoring occasion. The frequency offset is relative to the location of the detected SSB. The first stage of data processing is a DDC which performs a frequency shift to center the SIB1 grid and then downsamples to 30.72 MHz - the maximum bandwidth for CORESET0 in FR1. The next stage is to wait for the CORESET0 monitoring occasion - the algorithm contains a timing reference that is synchronized with the SSB Detector timing references to identify the next occurance of the monitioring occasion. Once the monitoring occasion is reached the received samples are OFDM demodulated to produce a grid CORESET0 resource blocks wide and two slots in duration.
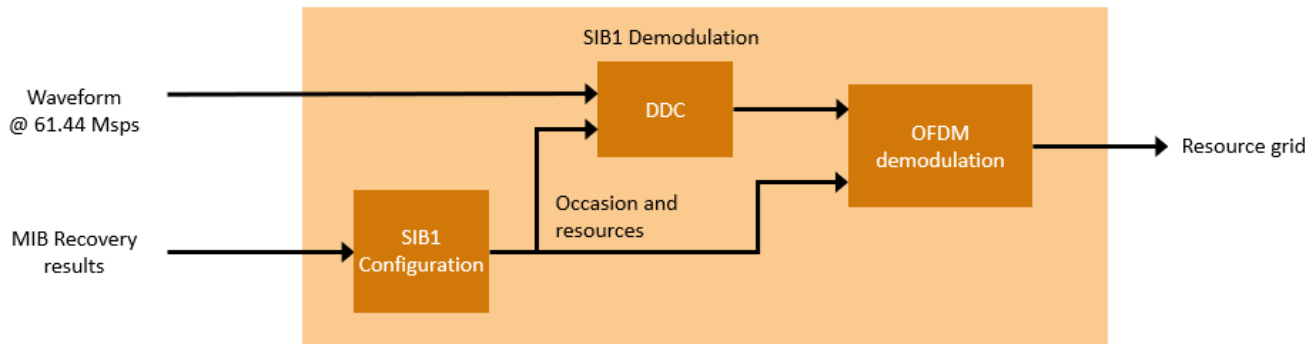
SIB1 Decoding

SIB1 decoding is performed on the SIB1 grid output by SIB1 demodulation. SIB1 decoding requires decoding of PDCCH to recover the SI-RNTI encoded DCI message, and decoding PDSCH to recover SIB1 message. The example shows two methods for decoding SIB1, either with or without hardware accelerators. The hardware accelerator version splits the each decoding stage into two steps. First, a setup step to create the input vectors which can be deployed to embedded software. Second, the hardware accelerated portion of the algorithm which can be deployed on an FPGA. Without hardware accelerators each decoding stage is performed by a single step which can be deployed to embedded software. By default, `nrhdlexamples.CORESET0Extract` and `nrhdlexamples.CORESET0Decode` are used. Alternatively, `nrhdlexamples.pdcchDecoding` is used. Both methods use the SIB1 grid and the parameters recovered from the previous decoding stages to locate and decode the PDCCH for CORESET0. They return the DCI message which signals the location of the PDSCH resources allocated to SIB1, and returns a flag indicating whether the DCI was found in the first or second slot of the SIB1 grid. The slot carrying PDSCH and PDSCH for SIB1 is selected from the SIB1 grid and `nrhdlexamples.coreset0PhaseAdjustment` corrects for the phase offset applied on an OFDM symbol basis by the transmitter, as detailed in TS 38.211 section 5.4. By default, `nrhdlexamples.SIB1Extract` and `nrhdlexamples.SIB1Decode` are used. Alternatively, `nrhdlexamples.pdschDecoding` is used. Both methods use the phase corrected slot grid, the DCI message, and other information from the previous decoding stages to locate and decode the PDSCH resources carrying the SIB1 message. They return the SIB1 message bits and the result of the SIB1 CRC check. A CRC value of 0 indicates successful recovery of the SIB1.
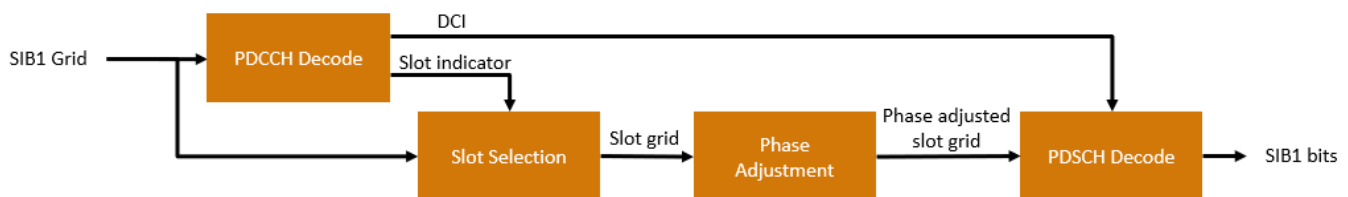
Generate a Test Waveform

This section shows how to use the MATLAB reference functions to search for SSBs in a waveform, demodulate and decode an SSB to recover the MIB, and recover the scheduled SIB1.

Use the `nrhdlexamples.generateFR1RxWaveform` function to generate a 5G FR1 waveform containing SSB bursts and the corresponding SIB1 transmissions. Change the `simulationCase` to explore different parameter sets. The full set of simulation cases is shown.

```
disp('Test waveform configurations:')
disp(nrhdlexamples.generateFR1RxWaveform('list'));

rng('default');

simulationCase = "SimCase 1";
[rxWaveform,ssbPattern,minChanBW,Lmax,txMIB,simCase] = nrhdlexamples.generateFR1RxWaveform(simula

disp("Selected Simulation case:" + newline);
disp(simCase);

FoCoarse = 0;

if ssbPattern == "Case A"
    scsSSB = 15;
else
    scsSSB = 30;
end
```

```
Test waveform configurations:
    Simulation Case     SSB Pattern     Subcarrier Spacing Common     PDCCH Config SIB1     SNR dB
    _____     _____     _____     _____     _____

      "SimCase 1"        "Case C"                  30                          164             50
      "SimCase 2"        "Case B"                  15                          100             6
      "SimCase 3"        "Case A"                  30                          4                20
      "SimCase 4"        "Case A"                  15                          84               7
Selected Simulation case:

    Simulation Case     SSB Pattern     Subcarrier Spacing Common     PDCCH Config SIB1     SNR dB
    _____     _____     _____     _____     _____

      "SimCase 1"        "Case C"                  30                          164             50
```

Plot the spectogram of the waveform.

The plot shows a spectogram of the SSBs, CORESET0s, and PDSCH regions carrying SIB1. These regions are generated with different power levels. The amplitude of each resource element is indicated by its color.

```
figure(1); clf;
rxSampleRate = 61.44e6;
nfft = rxSampleRate/(scsSSB*1e3);
spectrogram(rxWaveform(:,1),ones(nfft,1),0,nfft,'centered',rxSampleRate,'yaxis','MinThreshold',-1
title('Spectrogram of the Received Waveform')
```

Spectrogram of the Received Waveform
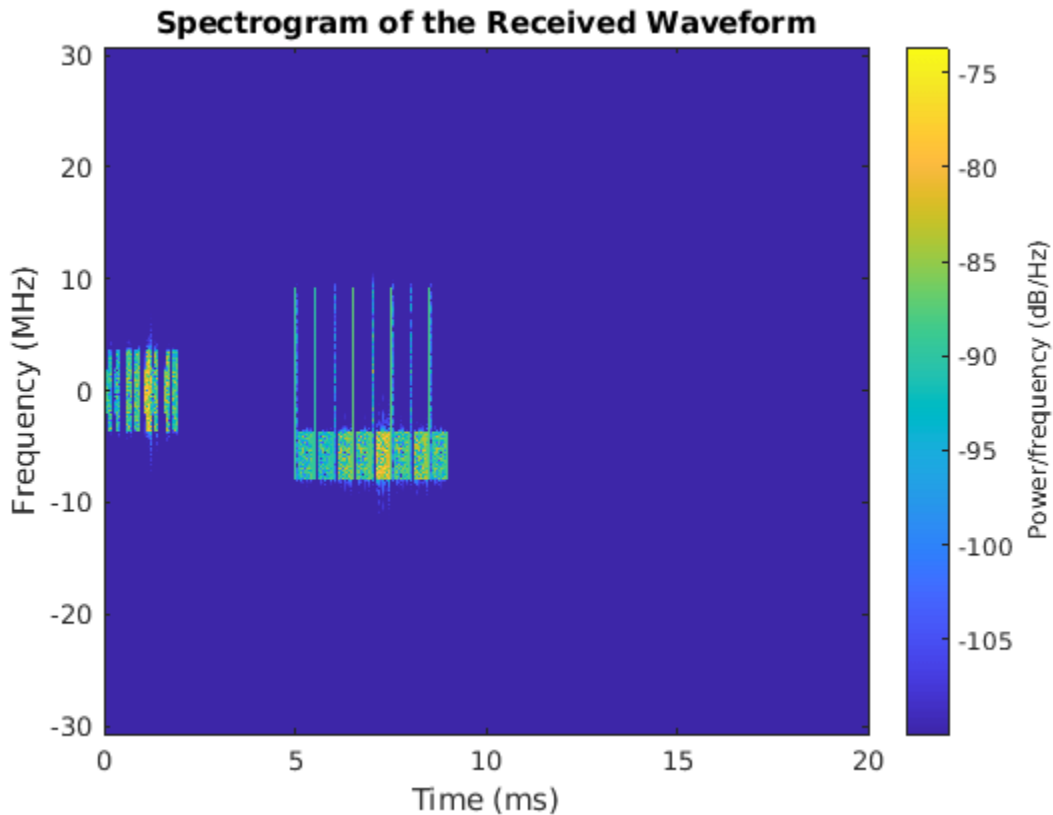
Detect SSBs

Use the `nrhdlexamples.ssbDetect` function to find SSBs in the waveform by searching for PSS symbols. This example calls the function with a coarse carrier frequency offset estimate of zero and a subcarrier spacing determined from the SSB pattern of the generated waveform. The function corrects the coarse frequency offset and measures the residual fine frequency offset of each SSB. Frequency offset input and output are given in Hz. The function returns a list of detected PSS symbols as a structure array. Display the structure array contents by converting it to a table.

```
[pssList,diagnostics] = nrhdlexamples.ssbDetect(rxWaveform,FoCoarse,scsSSB);

% Check if any PSS have been detected
if isempty(pssList)
    disp('No PSS found during SSB detection.');
    return;
end

disp('Detected PSS list:')
disp(struct2table(pssList));
```

```
Detected PSS list:
    NCellID2    timingOffset    pssCorrelation    pssEnergy    frequencyOffset
    _____    _____    _____    _____    _____

       0            4416           0.89699          0.90192          58
       0           17568           0.71337          0.71633          -2
       0           35136           1.7974           1.8062           18
       0           48288           1.4134           1.4254           35
```

0	65856	5.6595	5.6862	-59
0	79008	1.2633	1.2707	4
0	96576	2.2599	2.2702	14
0	1.0973e+05	1.1261	1.1315	-8

The `nrhdlexamples.ssbDetect` function also returns a structure containing diagnostic signals. Use this output to plot the PSS correlation results. Each peak in the correlator output shown corresponds to an entry in the PSS list.

```
figure(2); clf;
nrhdlexamples.plotUtils.PSSCorrelation(diagnostics,'PSS Correlation');
```
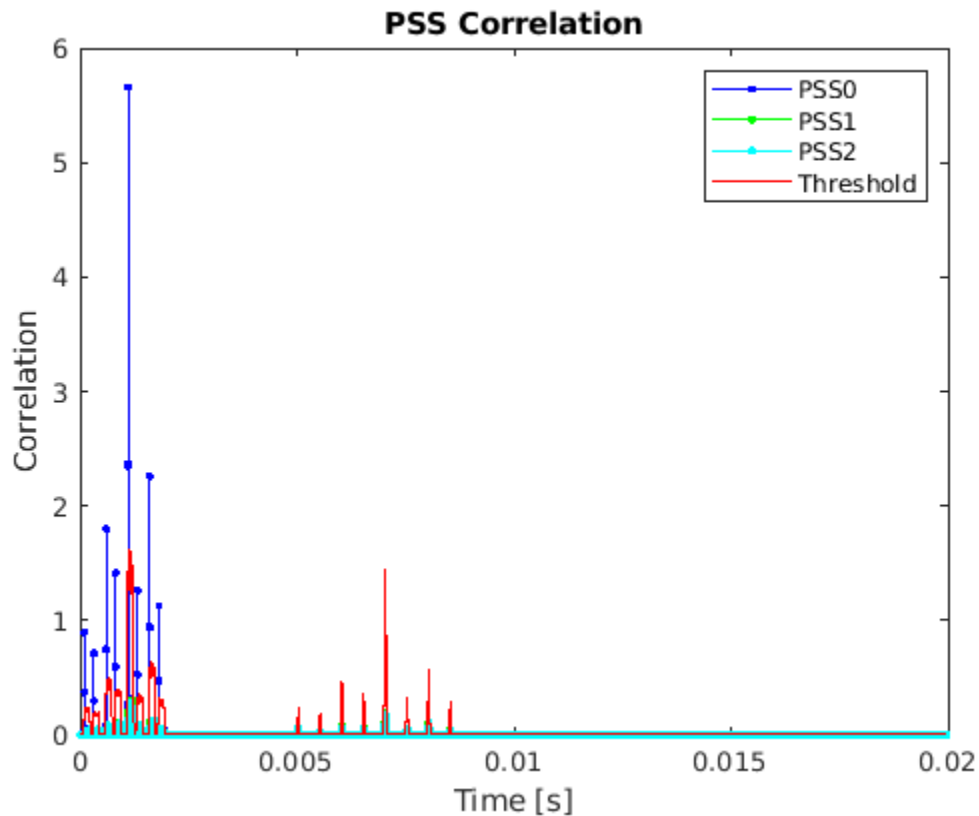

Use the `nrhdlexamples.ssbDetect` function to OFDM-demodulate one of the SSBs and attempt SSS detection. For this operation, call the function with an optional 4th argument that specifies the timing offset and NCellID2 of the desired SSB. This example chooses the PSS with the highest correlation metric, however you can choose any of the detected SSBs. Correct the frequency offset by passing in the sum of the coarse and fine frequency offset estimates.

```
[~,maxCorrIdx] = max(vertcat(pssList.pssCorrelation));
chosenPSS = pssList(maxCorrIdx);

disp('Selected PSS:')
disp(struct2table(chosenPSS));

FoFine = chosenPSS.frequencyOffset;
FoEst = FoCoarse + FoFine;
```

```
[ssBlockInfo,ssbGrid,diagnostics] = nrhdlexamples.ssbDetect(rxWaveform,FoEst,scsSSB,chosenPSS);

% Check SSB successfully demodulated
if isempty(ssBlockInfo)
    disp('Failed to demodulate selected SSB.');
    return;
end
```

```
Selected PSS:
    NCellID2      timingOffset      pssCorrelation      pssEnergy      frequencyOffset
    _____      _____      _____      _____      _____
       0             65856              5.6595            5.6862             -59
```

In demodulation mode, the function returns three outputs instead of two. The `ssBlockInfo` structure contains further details of the SSB, such as the SSS correlation strength and the overall cell ID. The `ssGrid` output is a matrix containing the demodulated OFDM symbols. Display the SSB info to confirm that the cell ID is correctly decoded.

```
disp('SSB info for demodulated SSB:')
disp(ssBlockInfo);
```

```
SSB info for demodulated SSB:
            NCellID2: 0
        timingOffset: 65856
      pssCorrelation: 5.6595
           pssEnergy: 5.6863
            NCellID1: 83
      sssCorrelation: 5.7356
           sssEnergy: 5.7357
             NCellID: 249
     frequencyOffset: 0
```

Display the resulting SSB resource grid.

```
figure(3); clf;
imagesc(abs(ssbGrid));
colorbar;
axis xy;
xlabel('OFDM symbol');
ylabel('Subcarrier');
title('SSB Resource Grid');
```

SSB Resource Grid
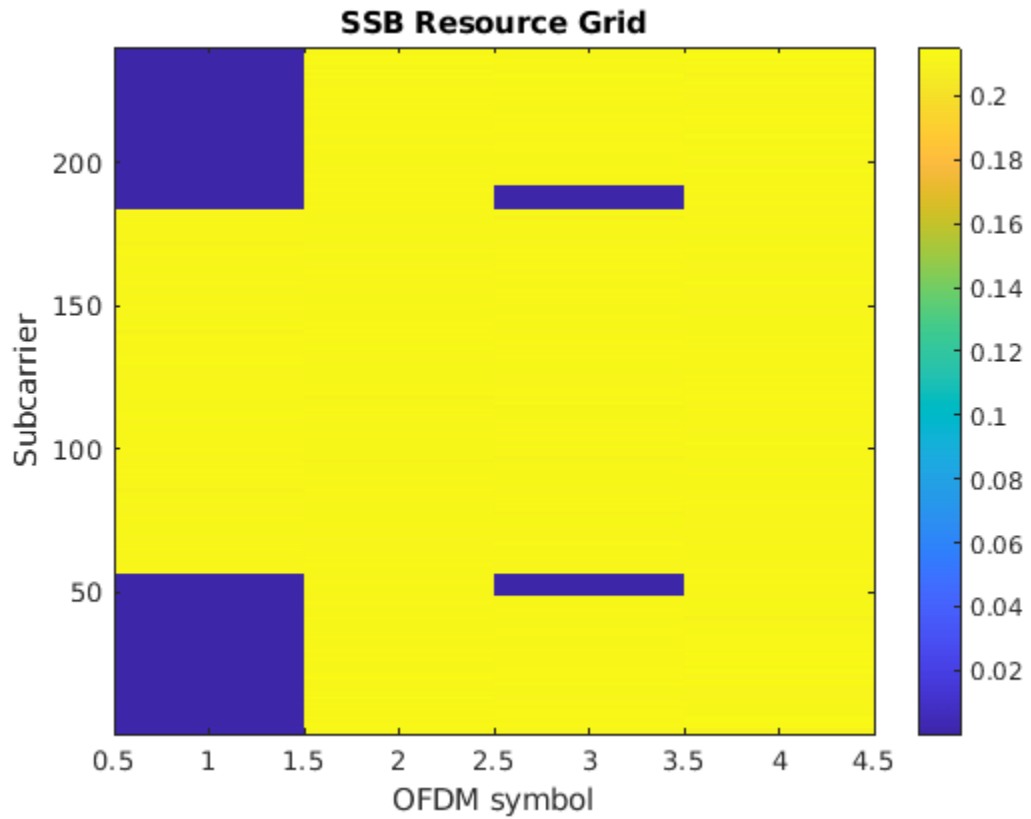
The `diagnostics` output includes SSS correlation results for all 336 possible sequences. Plot the SSS correlation results.

```
figure(4); clf;
nrhdlexamples.plotUtils.SSSCorrelation(diagnostics,'SSS Correlation')
```
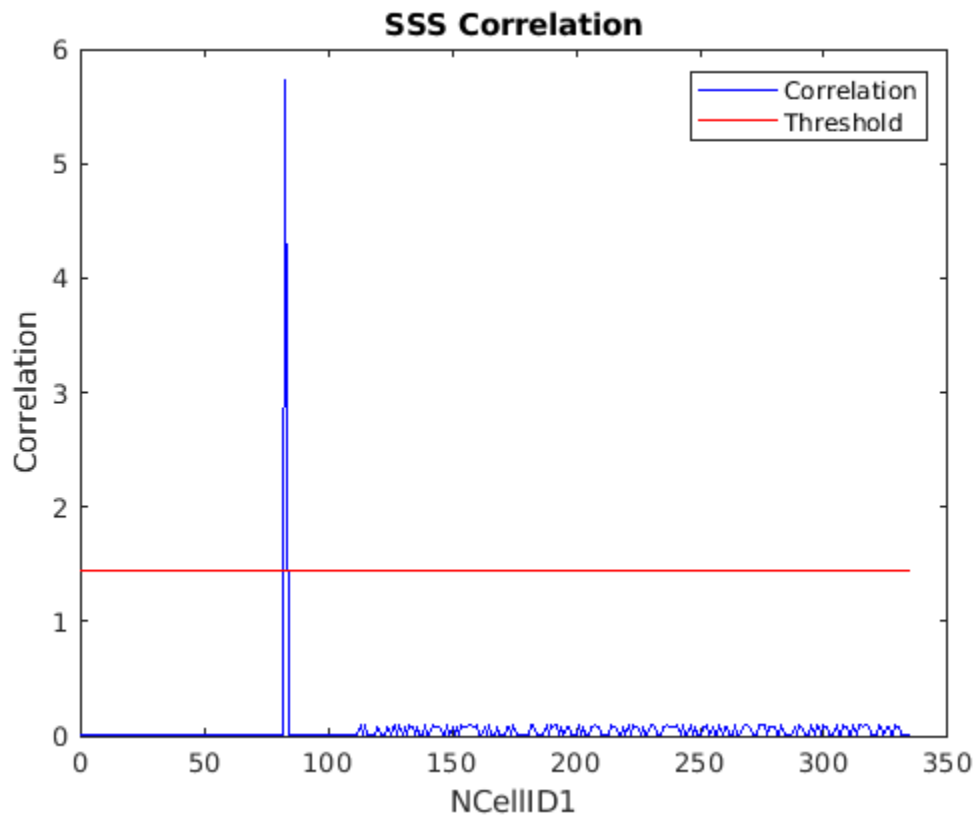
Search for Cells

This section shows how to use the `nrhdlexamples.cellSearch` function to search for and demodulate SSBs when the frequency offset and subcarrier spacing are not known. As described previously, the `nrhdlexamples.cellSearch` function builds on the `nrhdlexamples.ssbDetect` function by adding a search controller that looks for SSBs at different subcarrier spacings and frequency offsets.

Apply a frequency offset to test the coarse and fine frequency recovery functionality.

```
Fo          = 10000;
t           = (0:length(rxWaveform)-1).'/61.44e6;
rxWaveform = rxWaveform .* exp(1i*2*pi*Fo*t);
```

Define the frequency range endpoints and subcarrier spacing search space and call the `nrhdlexamples.cellSearch` function. The function displays information on the search progress as it runs. The frequency range endpoints must be multiples of half the maximum subcarrier spacing.

```
frequencyRange = [-30 30];
subcarrierSpacings = [15 30];

[ssBlockInfo,ssbGrid] = nrhdlexamples.cellSearch(rxWaveform,frequencyRange,subcarrierSpacings,st
    'DisplayPlots',false,...
    'DisplayCommandWindowOutput',true));

% Check cell search successfully found and demodulated SSB.
if isempty(ssBlockInfo)
```

```
        disp('Cell search failed to find or demodulate SSB.');
        return;
end
```

```
Searching for PSS (subcarrierSpacing: 15 kHz, frequencyOffset: -30 kHz)
Searching for PSS (subcarrierSpacing: 15 kHz, frequencyOffset: -22.5 kHz)
Searching for PSS (subcarrierSpacing: 15 kHz, frequencyOffset: -15 kHz)
Searching for PSS (subcarrierSpacing: 15 kHz, frequencyOffset: -7.5 kHz)
Searching for PSS (subcarrierSpacing: 15 kHz, frequencyOffset: 0 kHz)
Searching for PSS (subcarrierSpacing: 15 kHz, frequencyOffset: 7.5 kHz)
Searching for PSS (subcarrierSpacing: 15 kHz, frequencyOffset: 15 kHz)
Searching for PSS (subcarrierSpacing: 15 kHz, frequencyOffset: 22.5 kHz)
Searching for PSS (subcarrierSpacing: 15 kHz, frequencyOffset: 30 kHz)
Searching for PSS (subcarrierSpacing: 30 kHz, frequencyOffset: -30 kHz)
Searching for PSS (subcarrierSpacing: 30 kHz, frequencyOffset: -15 kHz)
Searching for PSS (subcarrierSpacing: 30 kHz, frequencyOffset: 0 kHz) ... PSS detected.
Searching for PSS (subcarrierSpacing: 30 kHz, frequencyOffset: 15 kHz) ... PSS detected.
Found PSS with (subcarrierSpacing: 30 kHz, frequencyOffsetEstimate: 9941 Hz)
Correcting frequency offset and searching for PSS again.
Found the following PSS symbols:

    NCellID2      timingOffset    pssCorrelation    pssEnergy    frequencyOffset
    _____      _____    _____    _____    _____
       0              4416           0.89698          0.90193         117
       0             17568           0.71336          0.71634          57
       0             35136           1.7974           1.8063           77
       0             48288           1.4135           1.4254           94
       0             65856           5.6595           5.6863            0
       0             79008           1.2633           1.2707           63
       0             96576           2.2599           2.2703           73
       0          1.0973e+05         1.1261           1.1315           51
Strongest PSS:
          NCellID2: 0
      timingOffset: 65856
    pssCorrelation: 5.6595
         pssEnergy: 5.6863
    frequencyOffset: 0
Attempting to reacquire strongest PSS and demodulate the corresponding SS block.
            NCellID2: 0
        timingOffset: 65856
      pssCorrelation: 5.6595
           pssEnergy: 5.6863
            NCellID1: 83
      sssCorrelation: 5.7356
           sssEnergy: 5.7357
             NCellID: 249
      frequencyOffset: 9941
    subcarrierSpacing: 30
Cell search summary:
  Subcarrier spacing: 30 kHz
    Frequency offset: 9941 Hz
       Timing offset: 65856
             NCellID: 249
```

As shown in the summary, the receiver returned the correct subcarrier spacing of 30 kHz, a cell ID of 249, and the measured frequency offset is close to the expected value of 10 kHz.

Decode SSB

Use the `nrhdlexamples.ssbDecode` function to decode the SSB resource grid and recover the MIB. The `nrhdlexamples.ssbDecode` function is based on the BCH decoding stages of the "NR Cell Search and MIB and SIB1 Recovery" (5G Toolbox) example.

```
[mibInfo,decodeDiags] = nrhdlexamples.ssbDecode(ssbGrid,ssBlockInfo.NCellID,Lmax);
```

```
% Check MIB successfully decoded from SSB.
if mibInfo.err
    disp('Failed to decode MIB from SSB.');
    return;
end
```

Plot the correlation peaks for the DMRS search. DMRS search is performed to determine ibar_ssb and the SSB index.

```
figure(5); clf;
plot(0:7,decodeDiags.dmrsCorr);
title('DMRS Search Correlation');
xlabel('ibar ssb');
ylabel('Correlation strength');
```
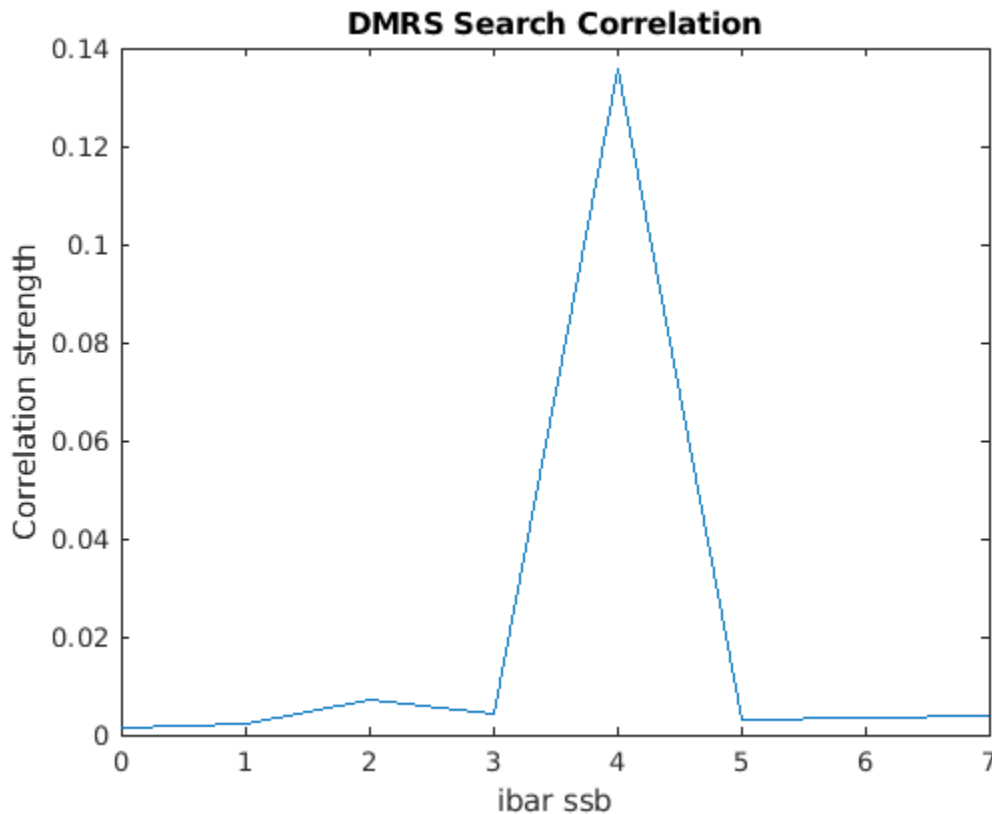

Plot the PBCH QPSK constellation after phase equalization.

```
figure(6); clf;
plot(decodeDiags.qpskSymb,'o');
xlim(max(abs(real(decodeDiags.qpskSymb))).*[-1.1 1.1]);
```

```
ylim(max(abs(imag(decodeDiags.qpskSymb))).*[-1.1 1.1]);
title('PBCH Symbol Constellation');
xlabel('In-phase');
ylabel('Quadrature');
```
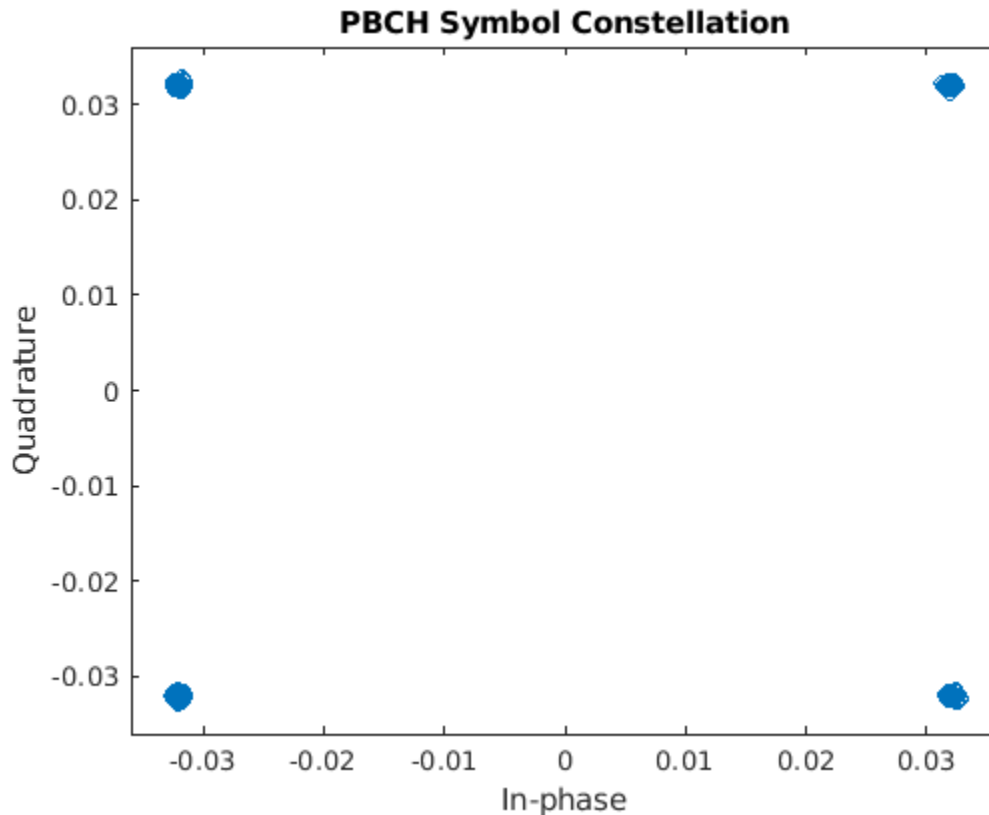

Display the decoded information and compare the transmitted and received MIB structures. These results show that the information was successfully decoded.

```
disp(['BCH CRC: ' num2str(mibInfo.err) newline]);

disp('Decoded information');
disp(mibInfo);

disp('Decoded MIB');
disp(mibInfo.mib);

disp('Expected MIB');
disp(txMIB);

BCH CRC: 0

Decoded information
    pbchPayload: 17637376
        ssbIndex: 4
             hrf: 0
             err: 0
             mib: [1×1 struct]
Decoded MIB
```

```
                    NFrame: 0
    SubcarrierSpacingCommon: 30
                     k_SSB: 0
           DMRSTypeAPosition: 3
             PDCCHConfigSIB1: 164
                 CellBarred: 0
        IntraFreqReselection: 0
Expected MIB
                    NFrame: 0
    SubcarrierSpacingCommon: 30
                     k_SSB: 0
           DMRSTypeAPosition: 3
             PDCCHConfigSIB1: 164
                 CellBarred: 0
        IntraFreqReselection: 0
```

Demodulate the SIB1 Grid

The `nrhdlexamples.sib1Demodulate` function determines the location of CORESET0, using information decoded from previous stages, and OFDM demodulates the SIB1 grid. The SIB1 grid contains CORESET0 and the PDSCH resources allocated to the SIB1 message.

```
ssbFrequencyOffset = ssBlockInfo.frequencyOffset;

ssbResults = struct(...
        'SubcarrierSpacing', scsSSB, ...
        'TimingOffset', ssBlockInfo.timingOffset, ...
        'FrequencyOffset', ssbFrequencyOffset);

bandCfg = struct( ...
        'ssbPattern', ssbPattern, ...
        'Lmax', Lmax, ...
        'MinChanBW', minChanBW ...
        );

sib1Grid = nrhdlexamples.sib1Demodulate(rxWaveform,ssbResults,mibInfo,bandCfg);
```

Plot the OFDM demodulated SIB1 grid.

```
figure_SIB1grid = figure(7); clf;
imagesc(abs(sib1Grid));
colorbar;
axis xy;
xlabel('OFDM symbol');
ylabel('Subcarrier');
title('SIB1 Grid');
```
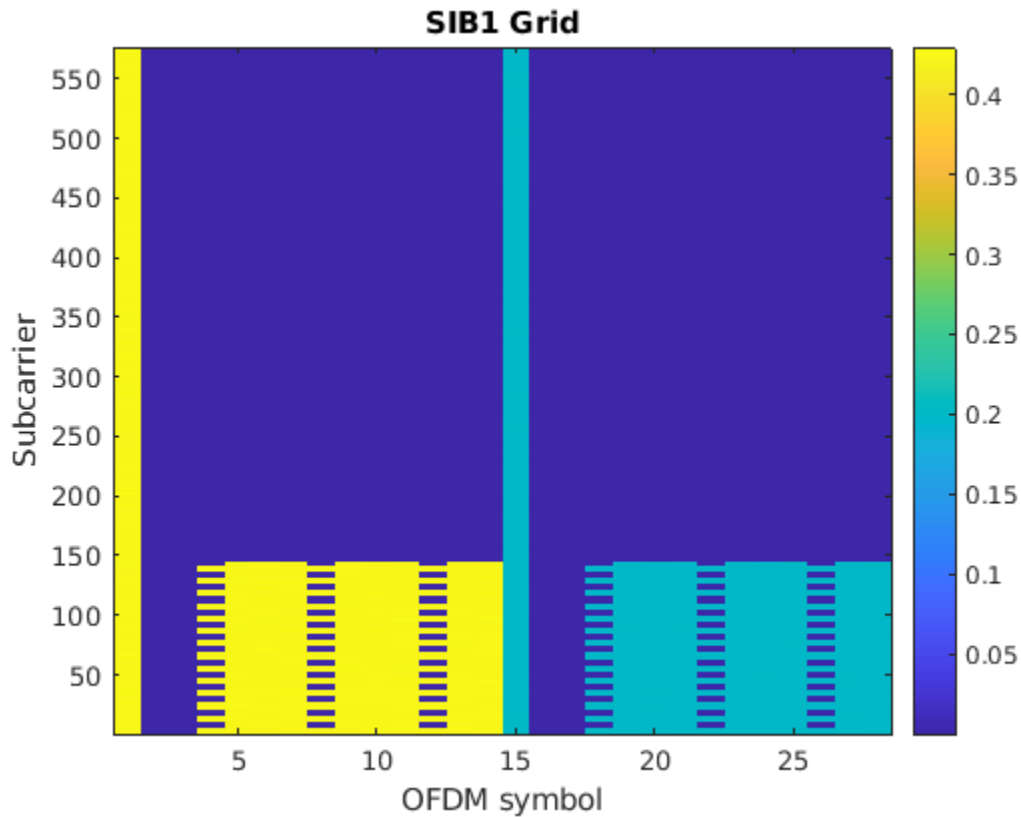
Decode the SIB1 Grid

The SIB1 grid consists of 2 slots. Only one of these slots carries CORESET0 and the PDSCH with SIB1. Depending on *useHardwareAccelerators* either `nrhdlexamples.CORESET0Extract` and `nrhdlexamples.CORESET0Decode` or `nrhdlexamples.pdcchDecoding` searches within each of the slots for DCI messages encoded with SI-RNTI. Once decoded, the SI-RNTI encoded DCI message provides information on the location of the SIB1 message within the PDSCH. Depending on *useHardwareAccelerators* either `nrhdlexamples.SIB1Extract` and `nrhdlexamples.SIB1Decode` or `nrhdlexamples.pdschDecoding` uses the DCI and information from the previous stages to locate and decode the SIB1 message within the PDSCH. If successfully decoded the sib1CRC will be 0, and the SIB1 message bits output.

```
useHardwareAccelerators = true;

if ~useHardwareAccelerators
    % Decode PDCCH and recover DCI message
    [dci,dciCRC,NSlot,secondSlotFlag,coresetNRB,muxPattern] = nrhdlexamples.pdcchDecoding(sib1Gr:

    % Check DCI successfully decoded from PDCCH.
    if dciCRC
        disp('Failed to decode DCI from PDCCH.');
        return;
    end

    % Select slot containing SIB1 message
    slotGrid = sib1Grid(:,(1:14)+(14*secondSlotFlag));
```

```
        % Adjust for phase offset applied by transmitter
        correctedSlotGrid = nrhdlexamples.coreset0PhaseAdjustment(slotGrid,mibInfo.mib,scsSSB,minChar

        % Decode PDSCH and recover SIB1 message bits
        [sib1bits,sib1CRC] = nrhdlexamples.pdschDecoding(correctedSlotGrid,ssBlockInfo.NCellID,mibIn
    else
        % Extract the CORESET0 search space candidates from the SIB1 grid to pass to
        % the CORESET0 decoding hardware accelerator
        [candData,candNSym,candBaseRBIdx,...
         searchSpaces,numSlots,coresetDuration,coresetNRB,coresetNSlot,muxPattern] = ...
            nrhdlexamples.coreset0Extract(sib1Grid,ssBlockInfo.NCellID,mibInfo.ssbIndex,scsSSB,mibIn

         % Run the CORESET0 decoding hardware accelerator
        [dci,dciCRC,secondSlotFlag] = ...
            nrhdlexamples.coreset0Decode(candData,candNSym,candBaseRBIdx,searchSpaces,numSlots,corese

        % Check DCI successfully decoded from PDCCH.
        if dciCRC
            disp('Failed to decode DCI from PDCCH.');
            return;
        end

        % Select slot containing SIB1 message
        slotGrid = sib1Grid(:,(1:14)+(14*secondSlotFlag));

        % Adjust for phase offset applied by transmitter
        correctedSlotGrid = nrhdlexamples.coreset0PhaseAdjustment(slotGrid,mibInfo.mib,scsSSB,minChan

        % Extract the SIB1 LDPC codeword from the SIB1 grid to pass to the SIB1
        % decoding hardware accelerator
        [ldpcCW,tbsLength] = nrhdlexamples.sib1Extract(correctedSlotGrid,ssBlockInfo.NCellID,mibInfo

        % Run the SIB1 decoding hardware accelerator
        [sib1Bits,sib1CRC] = nrhdlexamples.sib1Decode(ldpcCW,tbsLength);
    end

    % Update SIB1 grid plot to highlight PDCCH and PDSCH areas
    nrhdlexamples.plotUtils.labelSIB1Plot(figure_SIB1grid.Number,size(sib1Grid),ssBlockInfo.NCellID,n

    if sib1CRC == 0
        disp('SIB1 successfully decoded');
    else
        disp('SIB1 decoding failed');
    end
```

SIB1 successfully decoded
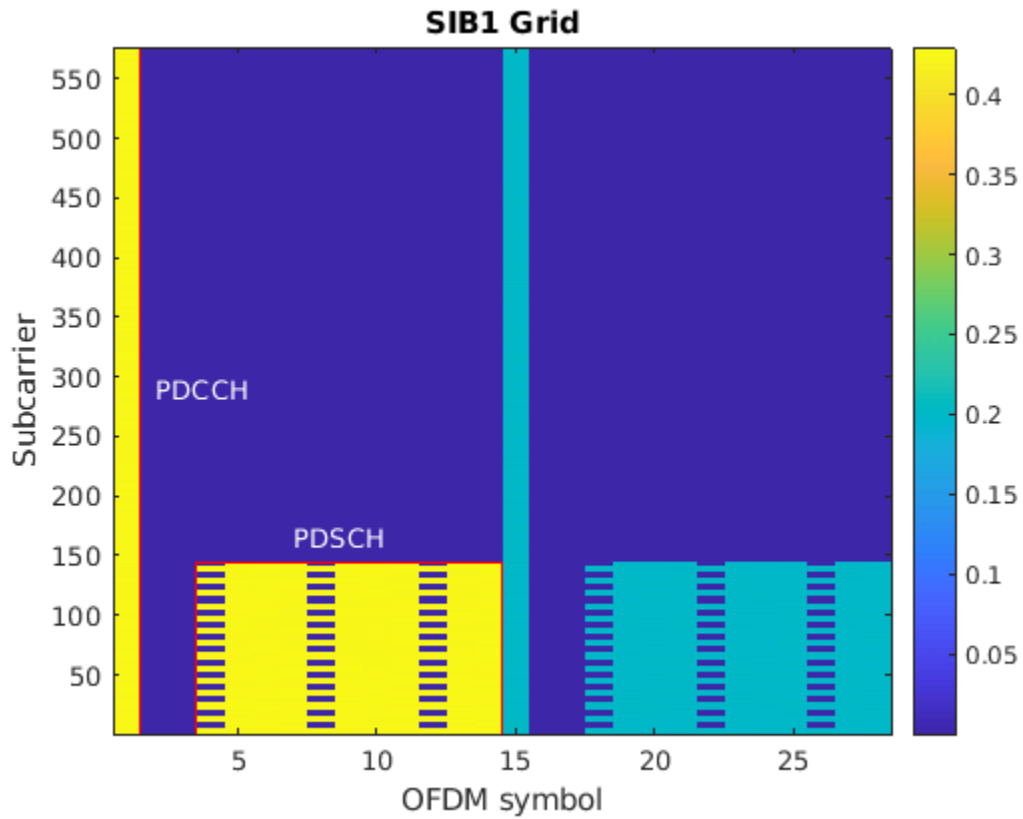
See Also

Related Examples

- "NR HDL Cell Search" on page 5-88
- "NR HDL MIB Recovery" on page 5-56

More About

- "NR HDL Receiver Performance" on page 5-5

NR HDL Cell Search

This example shows the design of a 5G NR cell search subsystem optimized for HDL code generation and hardware implementation.

Introduction

The Simulink® model described in this example is an HDL-optimized implementation of a synchronization signal block (SSB) detector for 5G NR frequency range 1 (FR1). This example is one of a related set, for more information see "NR HDL Reference Applications Overview" on page 5-2.

A block diagram of the SSB detector is shown in the figure. The detector performs all of the high-speed signal processing tasks associated with the cell search algorithm therefore is well suited for FPGA or ASIC implementation. The SSB detector searches for SSBs in time at a given frequency offset and subcarrier spacing. It is designed to be used as part of a larger system that implements carrier frequency offset recovery and subcarrier spacing detection. A controller must be used co-ordinate the overall cell search as shown in the "5G NR MIB Recovery Using Analog Devices AD9361/AD9364" (Communications Toolbox Support Package for Xilinx Zynq-Based Radio) example.

The SSB detector performs primary synchronization sequence (PSS) search, orthogonal frequency division multiplexing (OFDM) demodulation, and secondary synchronization sequence (SSS) search. It also includes a digital down converter (DDC) for correcting frequency offsets in the received signal. The SSB detector has two modes of operation, *search* and *demodulation*, which are demonstrated in this example. In search mode, the detector searches for SSBs and returns their parameters. In demodulation mode, the detector recovers a specified SSB OFDM-demodulates its resource grid and searches for SSS within the appropriate resource elements.
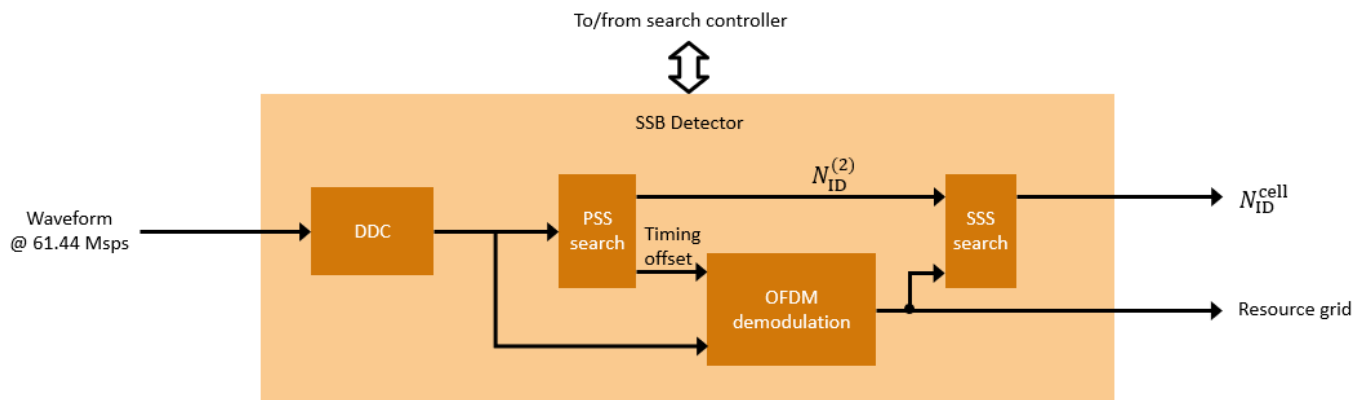
File Structure

The example uses these files.

Simulink models

- `nrhdlSSBDetection.slx`: This Simulink model uses the simulates the behavior of SSB detection.

- `nrhdlSSBDetectionFR1Core.slx`: This model implements the SSB detection algorithm.
- `nrhdlDDCFR1Core.slx`: This model implements a DDC to create sample streams for SIB1 and SSBs.

Simulink data dictionary

- `nrhdlReceiverData.sldd`: This Simulink data dictionary contains bus objects that define the buses contained in the example models.

MATLAB code

- `runSSBDetectionModelSearch.m`: Script for running and verifying the `nrhdlSSBDetection` model in search mode.
- `runSSBDetectionModelDemod.m`: Script for running and verifying the `nrhdlSSBDetection` model in demodulation mode.
- `nrhdlexamples`: Package containing the MATLAB reference code and utility functions for verifying the implementation models.

NR HDL Cell Search Model

This figure shows the `nrhdlSSBDetection` model. The top level of the model reads signals from the MATLAB base workspace, passes them to the SSB Detection subsystem, and writes the outputs back to the workspace. Use the `runSSBDetectionModelSearch` and `runSSBDetectionModelDemod` scripts to run the model and post-process the outputs.
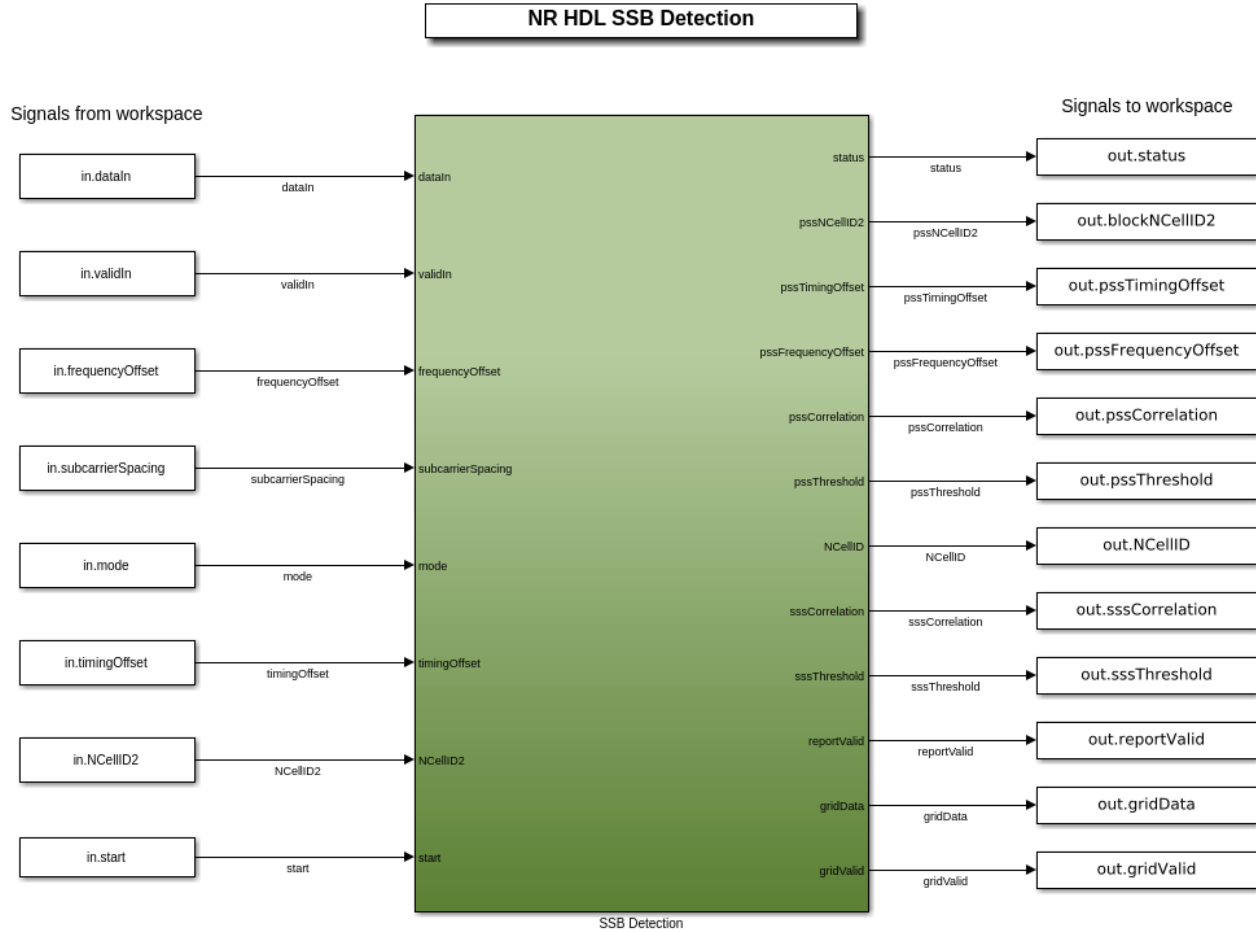
Copyright 2019-2021 The MathWorks, Inc.

SSB Detection Subsystem

The SSB Detection subsystem references the `nrhdlDDCFR1Core` and `nrhdlSSBDetectionFR1Core` models. The DDC performs frequency offset correction and decimation, and the SSB Detector searches for and demodulates SSBs. The algorithms of the model references are described in the next sections. The output of the DDC is the input to the SSB Detection algorithm.
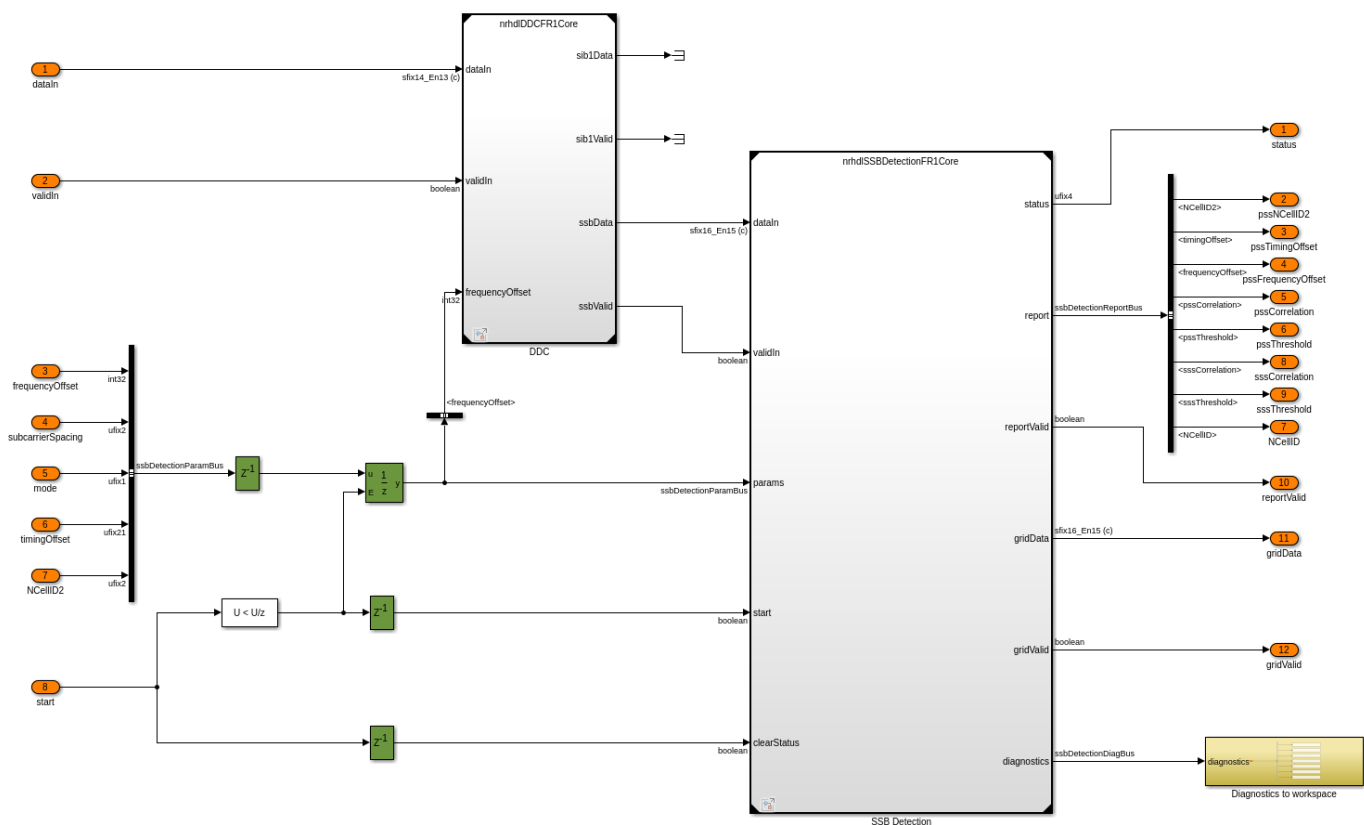
Inputs

- *dataIn*: 14-bit signed complex-valued signal, sampled at 61.44 Msps.

- *validIn*: 1-bit control signal to validate *dataIn*.

- *frequencyOffset*: 32-bit signed value specifying the frequency offset to be corrected. This signal is connected to an NCO with a 32-bit accumulator. Use this equation to convert the value to Hz: *frequencyOffset_Hz = frequencyOffset* * 61.44e6 / 2^32.

- *subcarrierSpacing*: 2-bit unsigned value specifying the subcarrier spacing. Set this signal to 0 to select 15kHz, or 1 to select 30kHz.

- *mode*: 1-bit unsigned value specifying the operation mode. Set this signal to 0 for search mode, or 1 for demod mode.

- *timingOffset*: 21-bit unsigned value specifying the timing offset of the start of the SSB to be demodulated. Specify the timing offset in samples at 61.44 Msps, from 0 to 1228799. This parameter applies only for demod mode.

- *NCellID2*: 2-bit unsigned value specifying the PSS (0, 1, or 2) of the SSB to be demodulated. This parameter applies only for demod mode.

- *start*: 1-bit control signal used to start a search or demodulation operation. To start an operation, set *frequencyOffset*, *subcarrierSpacing*, *mode*, *timingOffset*, and *NCellID2* to the desired values and set *start* to *1* (*true*) for one or more cycles. If an operation is already in progress, that operation is canceled when *start* is set to *1* (*true*). The new operation begins when *start* is returned to *0* (*false*).

Outputs

- *status*: 4-bit unsigned value that indicates the progress of the current operation. See the next section for the possible values of this signal.

- *pssNCellID2*: 2-bit unsigned value that is the PSS (0, 1 or 2) of the detected SSB.

- *pssTimingOffset*: 21-bit unsigned value that is the timing offset of the detected SSB. The timing offset is in samples at 61.44 Msp from 0 to 1228799.

- *pssFrequencyOffset*: 32-bit signed value that is the frequency offset of the detected SSB. This signal has the same units as the *frequencyOffset* input.

- *pssCorrelation*: 32-bit unsigned value that is the strength of the PSS correlation.

- *pssThreshold*: 32-bit unsigned value that is the threshold value when PSS was detected.

- *NCellID*: 10-bit unsigned value that is the cell ID of the demodulated SSB. This value is returned only in demod mode.

- *sssCorrelation*: 32-bit unsigned value that is the SSS correlation strength. This signal is returned only in demod mode.

- *sssThreshold*: 32-bit unsigned value that is the SSS threshold. This value is returned only in demod mode.

- *reportValid*: 1-bit control signal. In search mode, this signal validates *pssNCellID2*, *pssTimingOffset*, *pssFrequencyOffset*, *pssCorrelation*, and *pssThreshold* for each PSS that is detected. In demod mode, this signal also validates *NCellID*, *sssCorrelation*, and *sssThreshold*. In demod mode, *sssCorrelation* and *sssThreshold* are only valid if the specified SSB was found using its PSS, and *NCellID* is only valid if the SSS was detected.

- *gridData*: 16-bit signed complex-values that are the resource grid data. The receiver returns all four symbols of the SSB resource grid. Values are returned one resource element at a time. The resource grid is only returned in demod mode.

- *gridValid*: 1-bit control signal that validates the *gridData* output. Data is only returned if the specified SSB was found using its PSS. This signal is returned only in demod mode.

- *diagnostics*: Bus containing diagnostic signals.

Status Signal States

- `0`: Idle -- Initial state. Waiting for first start pulse.
- `1`: Search mode -- Searching for PSS.
- `2`: Search mode -- Operation complete, no PSS found.
- `3`: Search mode -- Operation complete, found one or more PSSs.
- `4`: Demod mode -- Waiting for specified PSS timing offset.
- `5`: Demod mode -- Operation complete, PSS not found.
- `6`: Demod mode -- Found specified PSS. Demodulating the resource grid and looking for SSS.
- `7`: Demod mode -- Operation complete, no SSS found. Returned demodulated resource grid.
- `8`: Demod mode -- Operation complete, found SSS. Returned demodulated resource grid.

DDC Model

This diagram shows the top level of the `nrhdlDDCFR1Core` model. The input signal (*dataIn*) is 16-bit signed complex-valued data sampled at 61.44 Msps. The DDC performs three operations. First, the `Input Scaling` subsystem scales the input by a factor of 0.875, providing headroom for the subsequent processing stages. Second, the `Frequency Correction` subsystem applies the given frequency offset to the data stream. Last, the `Filter and Downsample` subsystem filters and

decimates the samples by eight (to 7.68 Msps) using a chain of halfband filters. The `Filter and Downsample` subsystem produces two data streams. The *ssbData* output is sampled at 7.68 Msps and is used for SSB detection. The *sib1Data* output is sampled at 30.72 Msps and is used for SIB1 demodulation. These sample rates were selected as they are the minimum bandwidth required to compute a power of 2 FFT for each stream. The *sib1Data* output is used in the "NR HDL SIB1 Recovery" on page 5-8 example, where a single DDC is shared to drive both the SSB and SIB1 processing steps.
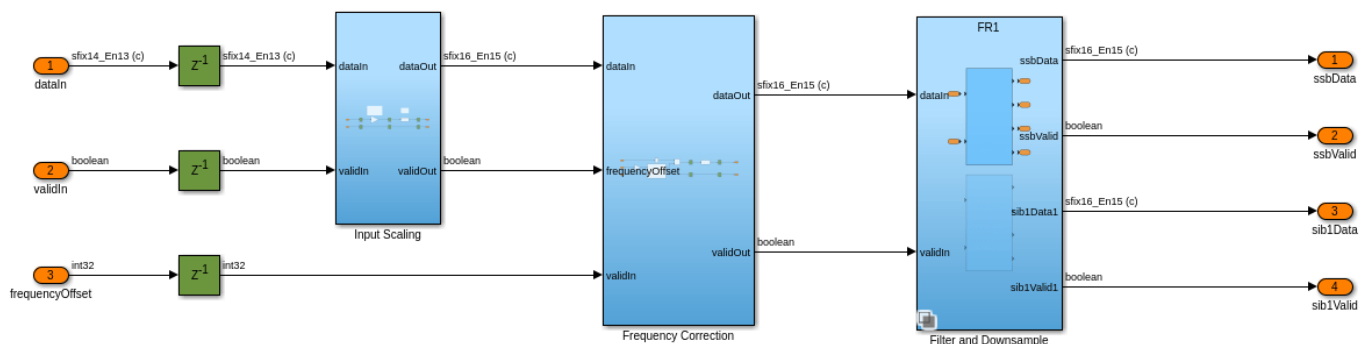
SSB Detection Model

This diagram shows the top level of the `nrhdlSSBDetectionFR1Core` model. The model performs SSB detection and demodulation. Its internal sampling rate varies depending on the subcarrier spacing (SCS). The model uses 7.68 Msps for 30kHz SCS and 3.84 Msps for 15kHz SCS. The subcarrier spacing selection logic on the left is responsible for changing the sampling rate. The rate can change only when a new operation is triggered by the *startProcessing* input.

The receiver has an internal timing reference system that keeps track of time by using counters at key points in the datapath. The timing reference counts 20ms periods - the assumed SSB periodicity for cell search as defined by the 5G NR standard. Time is measured in samples at 61.44 Msps modulo 1228800 to create the 20ms period. Since the actual sampling rate is either 7.84 Msps or 3.84 Msps, the timing reference counters increment by either 8 or 16, respectively, for each sample. When a new operation is triggered by the *start* input, the Start Controller records the start time and passes the time to the other timing references in the model. This signal tells the other timing references when a new subcarrier spacing and corresponding sampling rate applies. The other timing references wait until the start time before changing their increment. This design is possible only because hardware latency means the other timing references lag behind the Start Controller. This architecture enables the receiver to keep track of time consistently, even when a sampling rate change occurs.

The `nrhdlSSBDetectionFR1Core` model contains these main subsystems.

- *Subcarrier Spacing Selection*: Converts the input to two synchronized sample streams, one at 7.68 Msps and one at 3.84 Msps, and selects which stream to pass to subsequent processing stages according to the subcarrier spacing.

- *SSB Search*: Performs PSS correlation to search for SSBs.

- *SSB Demod*: Performs OFDM demodulation and SSS correlation.

- *Report Creation*: Aligns all of the parameters corresponding to one SSB detection, so that they are all valid at the same time.
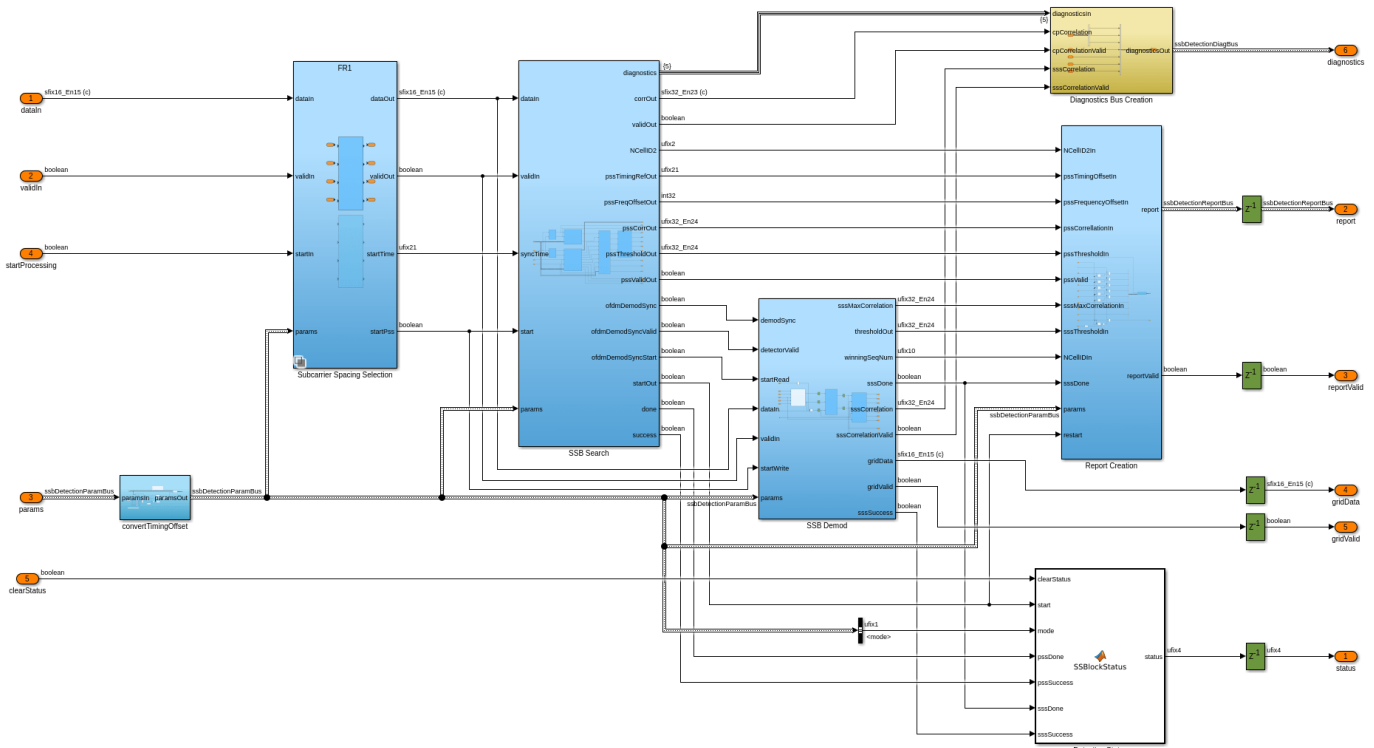
SSB Search subsystem

- *PSS Detection*: Searches for PSS symbols in the received signal. The next section describes this subsystem in more detail.

- *Cyclic Prefix Correlation*: Computes cyclic prefix (CP) correlation values. Each result is averaged across the last four OFDM symbols.

- *CP Correlation to Frequency*: Converts CP correlation values to fine frequency offset estimates.

- *PSS and CP Alignment*: Matches a CP-based frequency estimate with each PSS symbol detection instance. This alignment is necessary because the frequency estimate for a given PSS detection instance is available only at the end of the corresponding SSB.

- *PSS Info Serialization*: If PSS is detected on more than one PSS correlator output at the same timing offset, this block serializes the results so that they are returned from the detector one at a time.
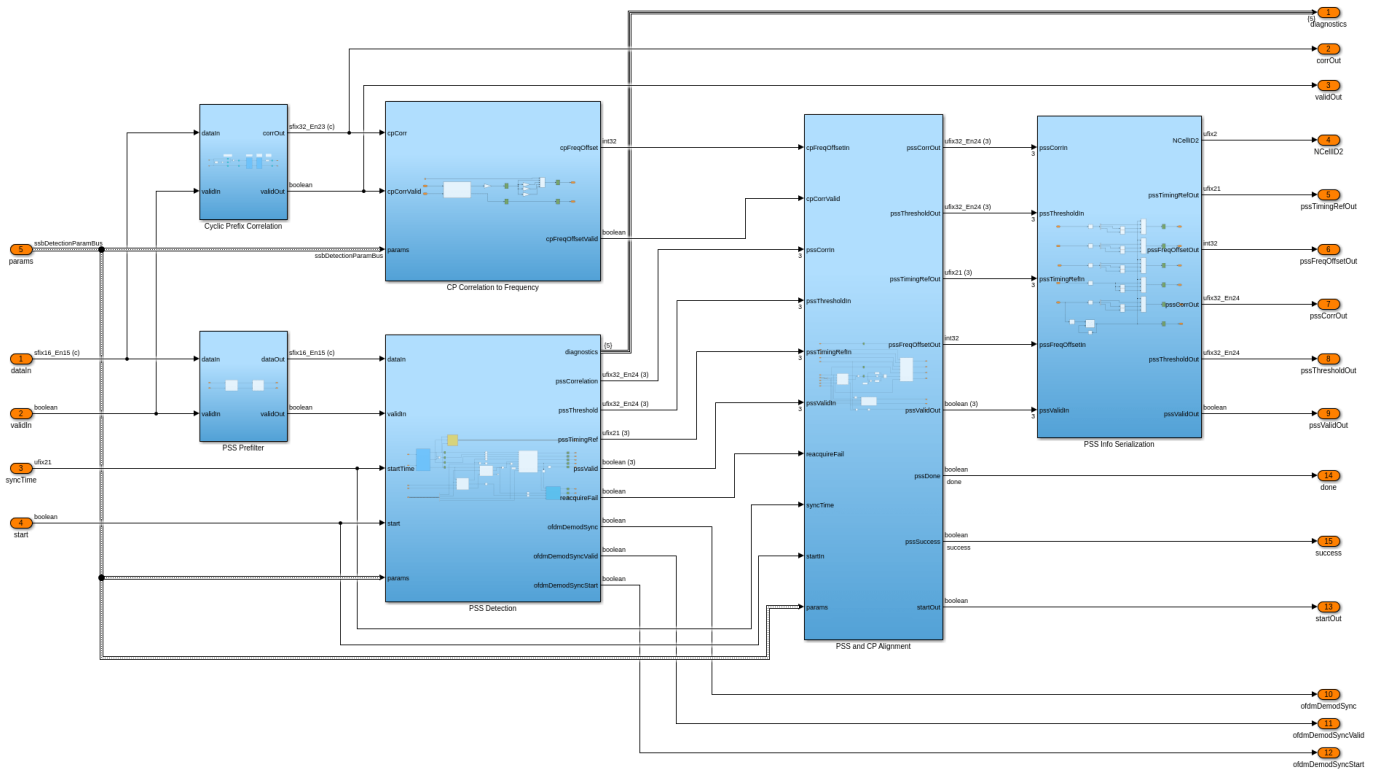
SSB Demod subsystem

- *OFDM Data Synchronization*: Synchronizes the OFDM demodulator input with the output of the PSS detector. This synchronization enables the PSS detector to trigger the OFDM demodulation process at the correct time. The synchronized data is one OFDM symbol behind the PSS correlator as the peak detection occurs at the end of the first OFDM symbol to be demodulated.

- *OFDM Demodulation*: OFDM-demodulates `the four symbols of the specified SSB.

- *SSS Detection*: Extracts the SSS resource elements from the OFDM demodulator output and correlates them with all 336 possible sequences to determine the cell ID.
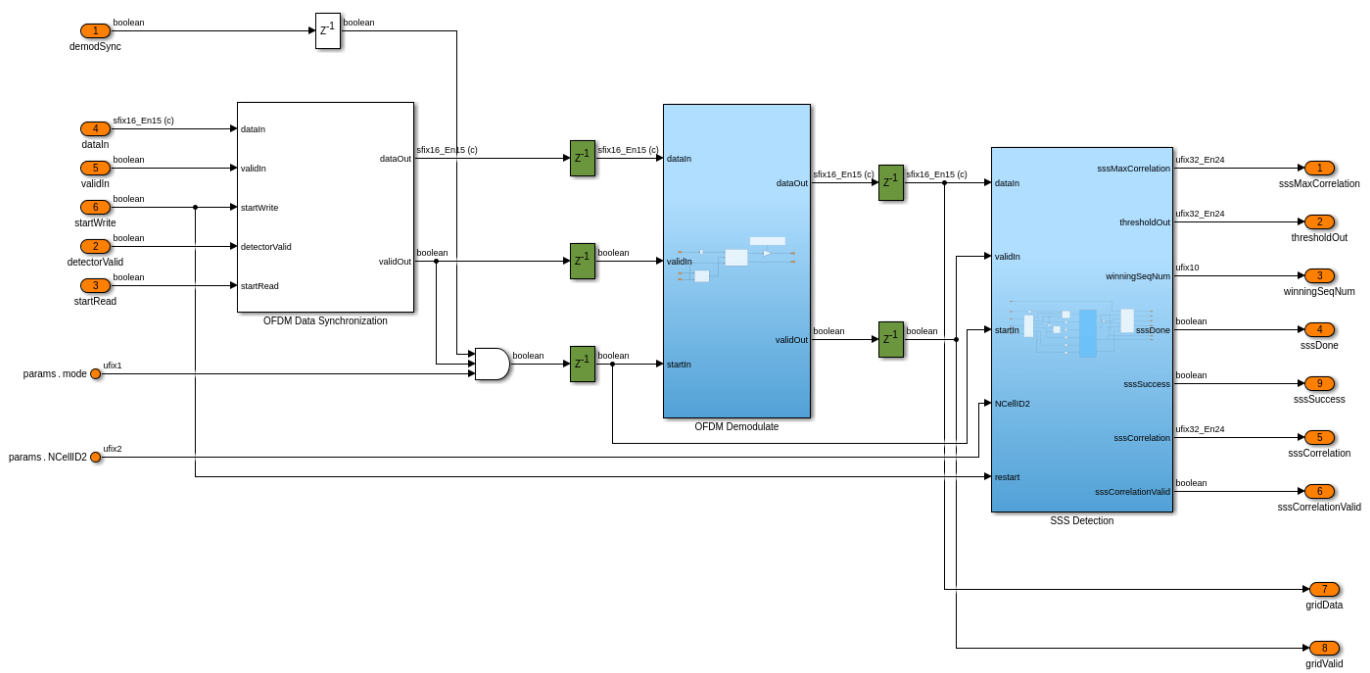
Simulation Setup

The block diagram shows the simulation setup of this example, which is implemented in the `runSSBDetectionModelSearch` and `runSSBDetectionModelDemod` scripts. 5G Toolbox™ functions are used to generate a test waveform which is applied to the MATLAB and Simulink implementations of the SSB detector in search mode and then in demodulation mode. Key diagnostic signals from each detector are compared in terms of their relative mean-squared error (MSE) and the final outputs are compared. Finally, the resource grid output of the Simulink model is decoded to show that the MIB contents are as expected.
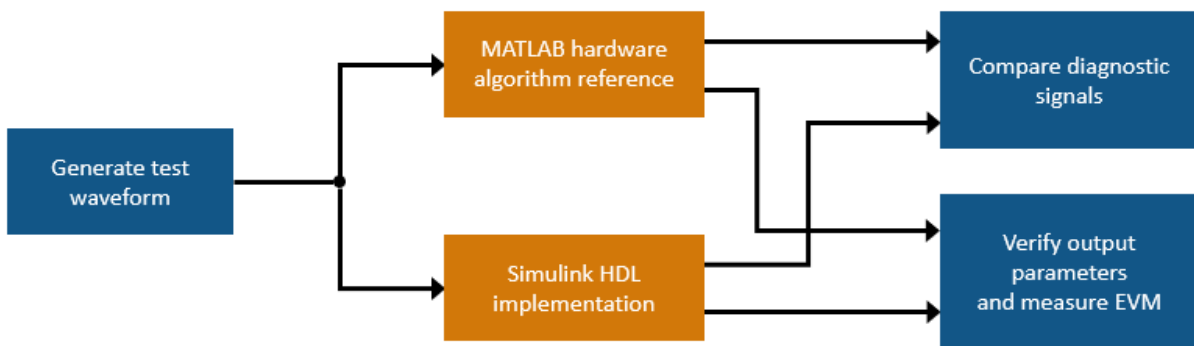
Search Mode Simulation

Use the `runSSBDetectionModelSearch` script to run a search mode simulation and verify the results. In search mode, the SSB detector searches for SSBs and returns their parameters. The script

displays its progress in the MATLAB command window. Tables show the parameters of each SSB detected by MATLAB and Simulink. The final table shows the relative MSE between MATLAB and Simulink for each correlator output and for the detection threshold. Plots are generated showing (i) the combined resource grid of all eight SSBs in the transmitted waveform and (ii) the PSS correlation outputs and threshold. The results show that the MATLAB and Simulink implementations match very closely. The small differences between the two implementations are due to quantization errors. These errors occur because the MATLAB reference uses floating-point data types, and the Simulink model uses fixed-point data types.

```
runSSBDetectionModelSearch;
```

```
Generating test waveform.
Selected Simulation case:
```

Simulation Case	SSB Pattern	Subcarrier Spacing Common	PDCCH Config SIB1	SNR dB
"SimCase 1"	"Case C"	30	164	50

```
Searching for SSBs using the MATLAB reference.
Searching for SSBs using the Simulink model.
Running nrhdlSSBDetection.slx
### Starting serial model reference simulation build
### Model reference simulation target for nrhdlDDCFR1Core is up to date.
### Model reference simulation target for nrhdlSSBDetectionFR1Core is up to date.
```

```
Build Summary
```

```
0 of 2 models built (2 models already up to date)
Build duration: 0h 0m 3.8694s
```

```
..........
```

SSBs found by MATLAB reference:

NCellID2	timingOffset	pssCorrelation	pssEnergy	frequencyOffset
0	4416	1.8564	2.0487	5057
0	17568	1.4776	1.6272	4997
0	35136	3.7246	4.1033	5016
0	48288	2.9372	3.243	5031
0	65856	11.729	12.921	4940
0	79008	2.6242	2.8901	5003
0	96576	4.6843	5.1559	5017
0	1.0973e+05	2.3364	2.5728	4997

SSBs found by Simulink model:

NCellID2	timingOffset	pssCorrelation	pssEnergy	frequencyOffset
0	4416	1.857	2.0492	5057
0	17568	1.4781	1.6277	4997
0	35136	3.7249	4.1042	5016
0	48288	2.9375	3.2439	5031
0	65856	11.732	12.923	4940
0	79008	2.6249	2.8908	5003
0	96576	4.6849	5.1569	5017
0	1.0973e+05	2.3367	2.5735	4997

Relative mean-squared error between MATLAB and Simulink in search mode:

```
         name                relativeMSEdB
    _____        _____

    {'PSS correlation 0'}        -71.51
    {'PSS correlation 1'}        -63.414
    {'PSS correlation 2'}        -63.272
    {'PSS threshold'    }        -76.134
```


Spectrogram of the Received Waveform
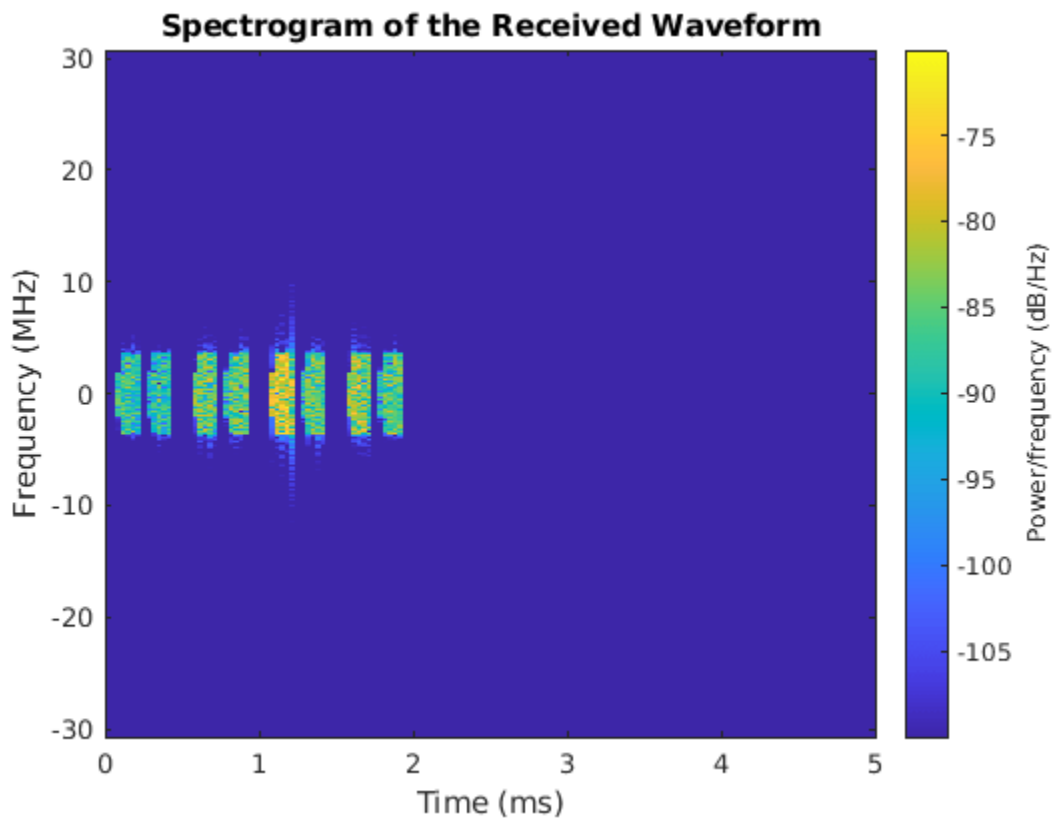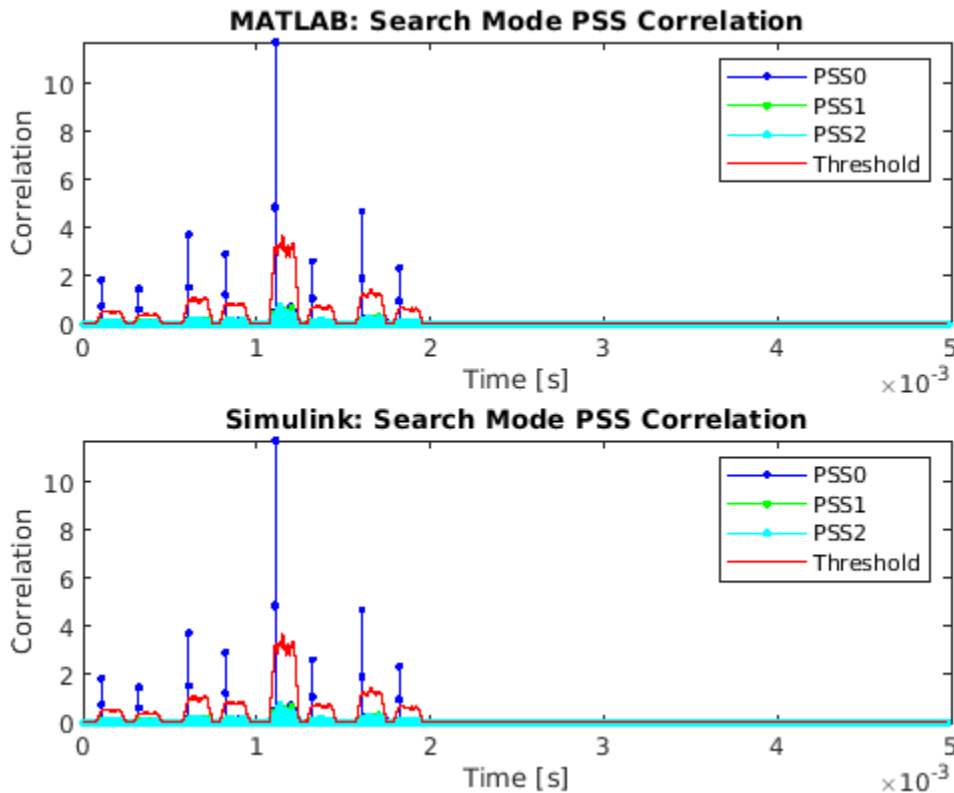
Use the Simulink Logic Analyzer to view the inputs and outputs to the SSB Detection subsystem. The detector looks for PSS symbols within a 20 ms time window, which begins after a pulse on the *start* input triggers the search operation. If no PSS symbols are found after 20 ms, the detector sets the *status* output to 2 - indicating that the search has failed. In this example, the detector finds all eight SSBs. The *status* output is set to 1 during the search, and a report is returned for each SSB by asserting the *reportValid* signal. The simulation only runs for 5 ms however if it is extended to run for more than 20 ms, then the *status* output is eventually set to 3 - indicating that the search has succeeded.
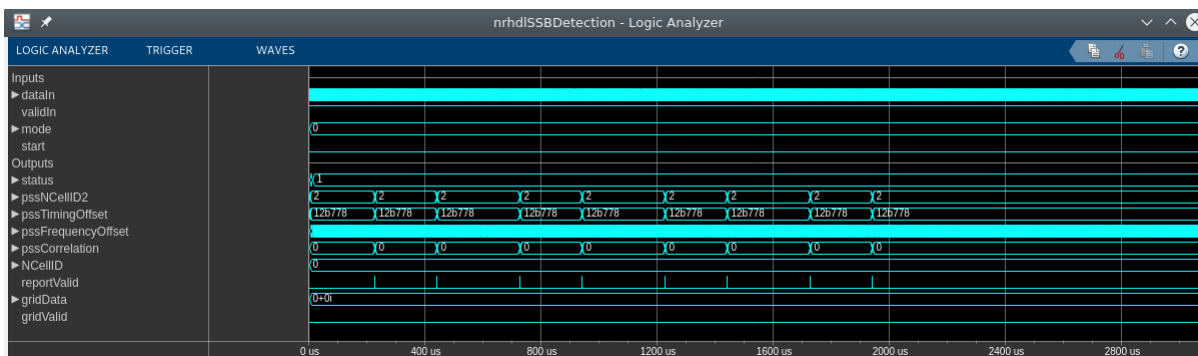
Demodulation Mode Simulation

After running `runSSBDetectionModelSearch`, use the `runSSBDetectionModelDemod` script to run a demodulation mode simulation and verify the results. In demodulation mode, the detector

recovers the specified SSB by searching for its PSS, OFDM-demodulating the resource grid, and searching for the SSS within the appropriate resource elements. The script displays its progress in the MATLAB command window. SS block reports from MATLAB and Simulink show that both detectors returned similar parameters and determined the cell ID correctly as 249. Relative MSE measurements indicate that the MATLAB and Simulink implementations match very closely. As a final verification step, the script decodes the broadcast channel (BCH) from the Simulink resource grid output. The CRC check passes and the master information block (MIB) contents match the transmission. Plots are generated which show the PSS and SSS correlation results, and the resource grid output. The PSS correlation levels are stronger in the demodulation mode simulation than in search mode simulation because the frequency offset is corrected.

```
runSSBDetectionModelDemod;

Choosing the strongest PSS from the previous search and computing its frequency offset.
        Strongest PSS index (1 based): 5
    Frequency offset (coarse + fine): 4.94 kHz
Demodulating the strongest SSBs using the MATLAB reference.
Demodulating the strongest SSBs using the Simulink model.
Running nrhdlSSBDetection.slx
### Starting serial model reference simulation build
### Model reference simulation target for nrhdlDDCFR1Core is up to date.
### Model reference simulation target for nrhdlSSBDetectionFR1Core is up to date.

Build Summary

0 of 2 models built (2 models already up to date)
Build duration: 0h 0m 1.6156s
...................

SS block report from MATLAB
          NCellID2: 0
      timingOffset: 65856
    pssCorrelation: 12.8410
         pssEnergy: 12.9020
          NCellID1: 83
    sssCorrelation: 13.0099
         sssEnergy: 13.0102
           NCellID: 249
   frequencyOffset: 0

SS block report from Simulink
          NCellID2: 0
      timingOffset: 65856
    pssCorrelation: 12.8441
         pssEnergy: 12.9064
          NCellID1: 83
    sssCorrelation: 13.0123
         sssEnergy: 13.0143
           NCellID: 249
   frequencyOffset: 0


Relative mean-squared error between MATLAB and Simulink in demod mode:

          name              relativeMSEdB
    _____    _____
```

```
{'PSS correlation 0'}        -69.651
{'PSS threshold'    }         -68.8
{'SSS correlation'  }        -69.961
{'Rx resource grid' }        -70.084
```

Decoding BCH from Simulink resource grid output:

 BCH CRC: 0

 Decoded (Rx) MIB
                      NFrame: 0
    SubcarrierSpacingCommon: 30
                      k_SSB: 0
           DMRSTypeAPosition: 3
            PDCCHConfigSIB1: 164
                  CellBarred: 0
        IntraFreqReselection: 0

 Expected (Tx) MIB
                      NFrame: 0
    SubcarrierSpacingCommon: 30
                      k_SSB: 0
           DMRSTypeAPosition: 3
            PDCCHConfigSIB1: 164
                  CellBarred: 0
        IntraFreqReselection: 0
```

**MATLAB: Rx Resource Grid**



**Simulink: Rx Resource Grid**



Use the Simulink Logic Analyzer to view the detector output as it progresses through these steps.

**1** The detector sets the *status* output to 4 while it waits for the specified timing offset and searches for the specified PSS.

**2** PSS is found. The detector sets the *status* output to 6 - the detector is now searching for the SSS within the resource grid. The four demodulated OFDM symbols are output, indicated by asserting *gridValid*.

**3** After the SSS is determined, the detector asserts *reportValid* to indicate that all of the PSS and SSS parameters, including *NCellID*, are valid. The *status* output changes to 8, to indicate that the operation is complete and SSS and cell ID are ready.

If the PSS is not found at the specified timing offset, the detector sets the *status* output to 5 and stops searching. If the detector is unable to determine the SSS, then it sets the *status* output to 7. In this example, the detector recovers the specified SSB - the SSB with the strongest PSS from the initial search.

**HDL Code Generation and Implementation Results**

To generate the HDL code for this example, you must have an HDL Coder™ license. Use the `makehdl` and `makehdltb` commands to generate HDL code and an HDL test bench for the `nrhdlSSBDetection/SSB Detection` subsystem. The resulting HDL code was synthesized for a Xilinx® Zynq®-7000 ZC706 evaluation board. The table shows the post place and route resource utilization results. The design meets timing with a clock frequency of 230 MHz.

```
    Resource          Usage
    _____   _____
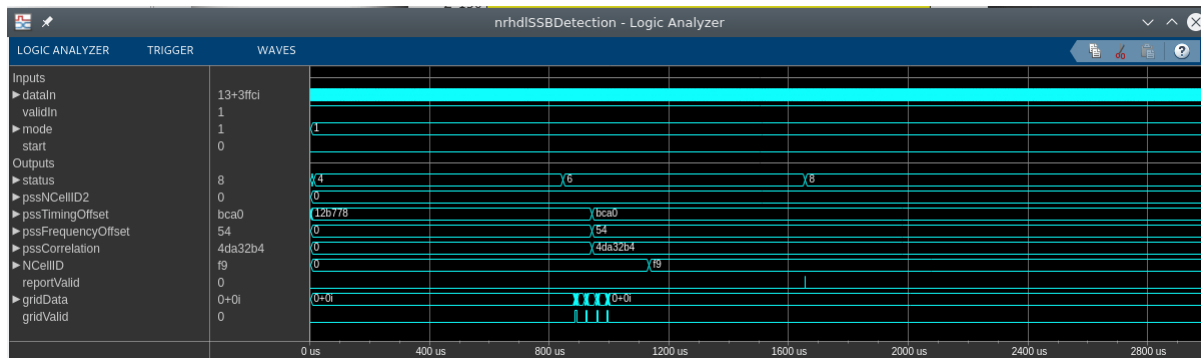
    Slice Registers    36531
    Slice LUTs         20857
    RAMB18                13
    RAMB36                 0
    DSP48                218
```

See Also

Related Examples

- "NR HDL Downlink Receiver MATLAB Reference" on page 5-68

More About

- "NR HDL Receiver Performance" on page 5-5

Deploy NR HDL Reference Applications on FPGAs and SoCs

This section contains the list of examples that show how to deploy 5G Wireless HDL Toolbox™ reference applications on FPGAs and SoCs.

- "5G NR MIB Recovery Using Analog Devices AD9361/AD9364" (Communications Toolbox Support Package for Xilinx Zynq-Based Radio): Deploy the MIB recovery algorithm.
- "5G NR MIB Recovery Using Xilinx RFSoC Device" (SoC Blockset Support Package for Xilinx Devices): Simulate and deploy a 5G NR MIB Recovery algorithm.
- "5G NR SIB1 Recovery Using Analog Devices AD9361/AD9364" (Communications Toolbox Support Package for Xilinx Zynq-Based Radio): Deploy the system information block 1 (SIB1) recovery algorithm.
- "5G NR SIB1 Recovery for FR1 and FR2 Using Xilinx RFSoC Device" (SoC Blockset Support Package for Xilinx Devices): Deploy the 5G NR SIB1 recovery algorithm for FR1 and FR2.

For a more detailed description of the algorithm, see the "NR HDL Downlink Receiver MATLAB Reference" on page 5-68 example.

These examples reuse the 5G Simulink® models to generate HDL for the FPGA logic. They use hardware-software co-design modeling techniques and hardware support packages to add all the software modeling and interfacing required to implement the algorithm in real-time on hardware.

See Also

Related Examples
- "NR HDL Cell Search" on page 5-88
- "NR HDL MIB Recovery" on page 5-56
- "Hardware Accelerators for NR SIB1 Recovery" on page 5-23
- "NR HDL SIB1 Recovery" on page 5-8
- "NR HDL SIB1 Recovery for FR2" on page 5-44

More About
- "NR HDL Receiver Performance" on page 5-5

LTE HDL Cell Search

This example shows how to design an LTE cell search and selection system optimized for HDL code generation and hardware implementation.

Introduction

Cell search and selection is the first step taken by User Equipment (UE) in attempting to gain access to an LTE network. The cell search and selection procedure involves detecting candidate eNodeB signals and then selecting one to synchronize to. This includes determining the chosen eNodeB's physical layer cell identity (cell ID) and duplex mode. Additionally, the UE acquires frequency and timing synchronization during this process. Once this procedure has been completed, the UE can demodulate the OFDM signal transmitted by the cell and recover its Master Information Block (MIB). A MIB Recovery model with HDL code generation capability, which reuses the cell search and selection functionality shown here, is presented in the "LTE HDL MIB Recovery" on page 5-141.

The functionality in the present example is based on the cell search functionality of the LTE Toolbox "Cell Search, MIB and SIB1 Recovery" (LTE Toolbox). However, the algorithms have been optimized for HDL code generation. LTE Toolbox was used extensively in the development of the present example. The HDL model described here performs the following functions:

- Frequency recovery
- Primary and secondary synchronization signal detection
- OFDM demodulation

The frequency recovery algorithm within the HDL model can only correct offsets fewer than +-7.5kHz. Large frequency offset recovery greater than +-7.5kHz is possible by driving the input and monitoring the outputs with an external controller. A demonstration of large frequency offset correction can be found in the "LTE MIB Recovery and Cell Scanner Using Analog Devices AD9361/ AD9364" (Communications Toolbox Support Package for Xilinx Zynq-Based Radio) example.

Once the model has completed the cell search and selection procedure, it outputs the cell ID, duplex mode and unequalized resource grid of the cell. This functionality is shown below. The model supports downlink signals with 15 kHz subcarrier spacing and normal cyclic prefix length. Frequency Division Duplex (FDD) and Time Division Duplex (TDD) modes are both supported. The duplex mode is automatically detected.

The LTE standard provides two *physical signals* to aid the cell search process. These are the Primary Synchronization Signal (PSS) and the Secondary Synchronization Signal (SSS). Refer to Appendix A for more information on LTE downlink synchronization signals.

Example Structure

The model consists of 5 files:

- `ltehdlCellSearch.slx`: This is the top level of the model, and acts as a test bench for `ltehdlDownlinkSyncDemod.slx`.
- `ltehdlDownlinkSyncDemod.slx`: Model reference which implements the cell search, synchronization, and OFDM demodulation functionality.
- `ltehdlCellSearch_init.m`: MATLAB® script for generating stimulus.
- `ltehdlCellSearch_analyze.m`: MATLAB script for analyzing output and displaying plots at the end of the simulation.
- `ltehdlCellSearchTools.m`: MATLAB class containing helper methods for analyzing and plotting results.

Note: `ltehdlDownlinkSyncDemod.slx` does not appear in the example working folder as it is shared with other examples. The file is on the MATLAB path and can be opened by entering `ltehdlDownlinkSyncDemod` at the MATLAB command line.

Model Architecture

The structure of the cell search and selection subsystem is shown below. The input is complex 16-bit data sampled at 30.72 Msps. The signal is passed to two signal processing data paths; one at 1.92 Msps and one at 30.72 Msps. Frequency recovery and PSS detection are performed on the 1.92 Msps data path. This sampling rate is used for two reasons. First, the cell bandwidth is not known at this stage therefore the smallest LTE bandwidth of 1.4 MHz is assumed for frequency recovery. This approach works irrespective of the actual cell bandwidth. Second, the PSS and SSS only occupy the six central resource blocks (1.4 MHz). Therefore, detection can be performed effectively at 1.92 Msps and resource sharing techniques can be used to optimize the hardware implementation.

The following steps describe the receiver operation.

1   The frequency estimation block estimates the frequency offset over a 10 ms period.

2   The frequency correction blocks are then activated on both the 1.92 Msps and 30.72 Msps sample streams.

3   PSS detection begins immediately after the frequency estimation stage has been completed.

4   SSS detection begins when PSS detection detects a valid PSS signal. If a valid SSS is found, this means that a valid cell has been detected and the duplex mode is now known.

5   The cell ID and frame start position are computed.

6   On the next frame boundary, the receiver starts to extract OFDM symbols from the 30.72 Msps sample stream. Each symbol is passed through a 2048-point FFT to perform OFDM demodulation.

Appendix B provides more details of the cell search and selection algorithm used in this example.

Cell Search Simulink Model

The top level of `ltehdlCellSearch.slx` is shown below. This model references `ltehdlDownlinkSyncDemod.slx`. `ltehdlCellSearch_init.m` is called by the `InitFcn` callback and `ltehdlCellSearch_analyze.m` is called by the `StopFcn` callback. The model uses a **Stop** sink to terminate the simulation when either (i) the *subframeNum* output is 5 or (ii) *cellSearchDone* is asserted `true` and no cell is detected. HDL code can be generated for the **Cell Search HDL** subsystem.

The **Cell Search HDL** subsystem is primarily a wrapper for the **ltehdlDownlinkSyncDemod** model. It contains a **Model** block (**Downlink Sync Demod**) which references `ltehdlDownlinkSyncDemod.slx`, and a **Diagnostics To Workspace** subsystem, which logs all of

the diagnostic outputs. The diagnostic outputs are used by `ltehdlCellSearch_analyze.m` to generate plots showing the internal operation.

Downlink Synchronization and Demodulation Model Reference

The **ltehdlDownlinkSyncDemod** model reference implements all of the cell search, synchronization and OFDM demodulation functionality. Appendix B details the cell search and selection algorithm implemented by this model. The top level of **ltehdlDownlinkSyncDemod** closely matches the architecture which was presented earlier.

Model Inputs:

- *dataIn*: Complex signed 16-bit data carrying the baseband input signal.
- *validIn*: Boolean, indicating whether *dataIn* is valid.
- *start*: Boolean. Assert this input `true` for one cycle at any time to initiate a cell search. This is referred to as a start command.

Model Outputs:

- *NCellID*: 9-bit cell ID of the detected eNodeB.
- *TDDMode*: Boolean, indicating the duplex mode of the detected cell: `false` for FDD, `true` for TDD.
- *timingOffset*: 19-bit timing offset. Indicates the number of samples from the first sample to enter the receiver to the first sample of the first full frame, from 0 to 307199.
- *freqEst*: 14-bit signed frequency offset estimate. Multiply this output by 15e3 / 2^14 in order to convert to Hz as shown in the **LTEHDLCellSearch** model.

- *cellDetected*: Boolean, indicating that a cell has been found.

- *cellSearchDone*: Boolean, indicating that the cell search has completed. If a cell is found, *cellDetected* and *cellSearchDone* will be asserted true at the same time. If no cell is found, *cellDetected* will remain `false` and *cellSearchDone* will be asserted `true` within 100 ms of the start command being issued. The time taken for *cellSearchDone* to be asserted depends on how many attempts are taken to detect PSS and SSS. See Appendix B for more details.

- *subframeNum*: 4-bit unsigned integer. Indicates which subframe is currently being passed out of the *gridData* port, from 0 to 9.

- *gridData*: 16-bit data carrying the demodulated resource grid.

- *gridValid*: Boolean, indicating whether *gridData* is valid.

- *diagnostics*: Bus signal, carrying various diagnostic outputs.

ltehdlDownlinkSyncDemod uses two Wireless HDL Toolbox™ example functions during initialization: `ltehdlDefineReceiverBuses` and `ltehdlDownlinkSyncDemodConstants`. `ltehdlDefineReceiverBuses` is shared with other Wireless HDL Toolbox examples, and defines a set of Simulink buses. This function is called in the `InitFcn` of **ltehdlDownlinkSyncDemod**. Only the `detectorDiagnosticsBus` output of the function is used here. The bus object is stored in the Base Workspace, making it available to both the **ltehdlDownlinkSyncDemod** and **ltehdlCellSearch** models.

```
[~,~,~,~,detectorDiagnosticsBus] = ltehdlDefineReceiverBuses();
```

The model relies on precomputed constants and lookup tables stored in a structure called `cellDetectorConfig`. This structure is generated by the `ltehdlDownlinkSyncDemodConstants` function and is only used inside the **ltehdlDownlinkSyncDemod** model reference. Therefore, it is defined in the Model Workspace rather than the Base Workspace. Use the Model Explorer to view the Model Workspace, which contains the following initialization code.

```
cellDetectorConfig = ltehdlDownlinkSyncDemodConstants(30.72e6);
```

The internal structure of **ltehdlDownlinkSyncDemod** is shown.

The **Decimation Filters** subsystem resamples the input data from 30.72 Msps to 1.92 Msps. It consists of CIC decimation, CIC gain compensation, CIC droop compensation, and transient removal. The filter chain is designed to have a group delay which is equal to an integer number of samples at 1.92 Msps. The **Transient Removal** block removes the initial transient due to this group delay from the sample stream. This is important because the frame timing offset is measured on the 1.92 Msps stream and then used to recover timing on the 30.72 Msps stream. Removing the initial transient from the decimation filter chain simplifies the logic which transfers the timing information.

The **FrequencyEstimation** subsystem uses the cyclic prefix to estimate the frequency offset of the incoming signal. The **CyclicPrefixCorrelator** continually measures the autocorrelation of the incoming signal with a lag of 128 samples across an 8 sample time window. It improves the measurement using a 7-tap OFDM-symbol spaced averaging filter. The result is converted to magnitude and phase by the **Rect2Polar** subsystem. Every 10ms, the **AngleAtMaximum** subsystem selects the strongest correlation peak, records its phase angle, and translates it into a frequency estimate. This algorithm works well for both FDD and TDD modes, including configurations where only a small portion of each frame is allocated to the downlink. Appendix B provides more information on how the cyclic prefix can be used to estimate the frequency offset.

The **Sync Signal Search** subsystem implements PSS and SSS detection. Timing is crucial in this part of the design, because the **SSS Searcher** uses the frame timing information from the **PSS Searcher** to identify SSS search locations. The **PSS Searcher** provides a **validOut** signal which is used by the **Stream Synchronizer** block to delay the input stream and compensate for the **PSS Searcher** pipeline latency. Synchronizing the input stream to the **PSS Searcher** outputs simplifies the design of the **SSS Searcher**.

The **PSS Searcher** is made up of two subsystems: the **Correlators** and the **Max Peak Searcher**. Together, these subsystems implement the PSS search algorithm described in Appendix B.

The **Correlators** subsystem contains a matched filter for each of the three PSS sequences, and a set of subsystems for determining the threshold. A lower limit is applied to the threshold to prevent small signals triggering false alarms. The PSS correlators and the threshold generation logic have different pipeline delays, therefore, a stream synchronizer is used to re-align their outputs.

Once a cell search is underway, the **SSS Searcher** continually stores samples in a circular buffer. Once PSS is detected, it continues to load samples into the buffer until the SSS search location has been reached and stored. The SSS search location is computed from the PSS timing information provided by the **PSSEndTimingOffset** signal. Next, the FDD location samples are read from the buffer, passed through a 128-point FFT, and the **Max Likelihood SSS** subsystem computes the correlation metrics and threshold. The same operation is then applied to the TDD location samples. The **Max Likelihood SSS** subsystem chooses the maximum correlation metric which exceeded the threshold and determines the duplex mode and frame timing. Finally, the frame timing offset is computed.

Initialization and Analysis Scripts

Initialization Script

ltehdlCellSearch_init.m is called in the InitFcn callback of ltehdlCellSearch.slx. Stimulus can either be loaded from a file containing a captured off-the-air waveform, or generated with LTE Toolbox.

```
% ltehdlCellSearch model initialization script
% Generates workspace variables needed by the ltehdlCellSearch model.
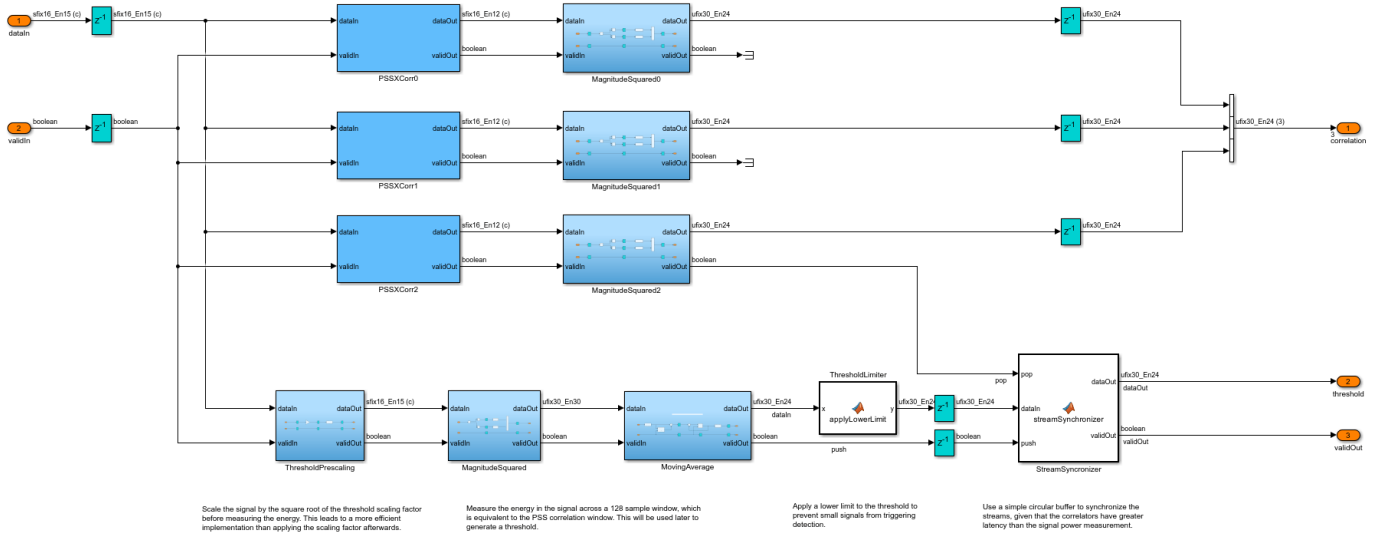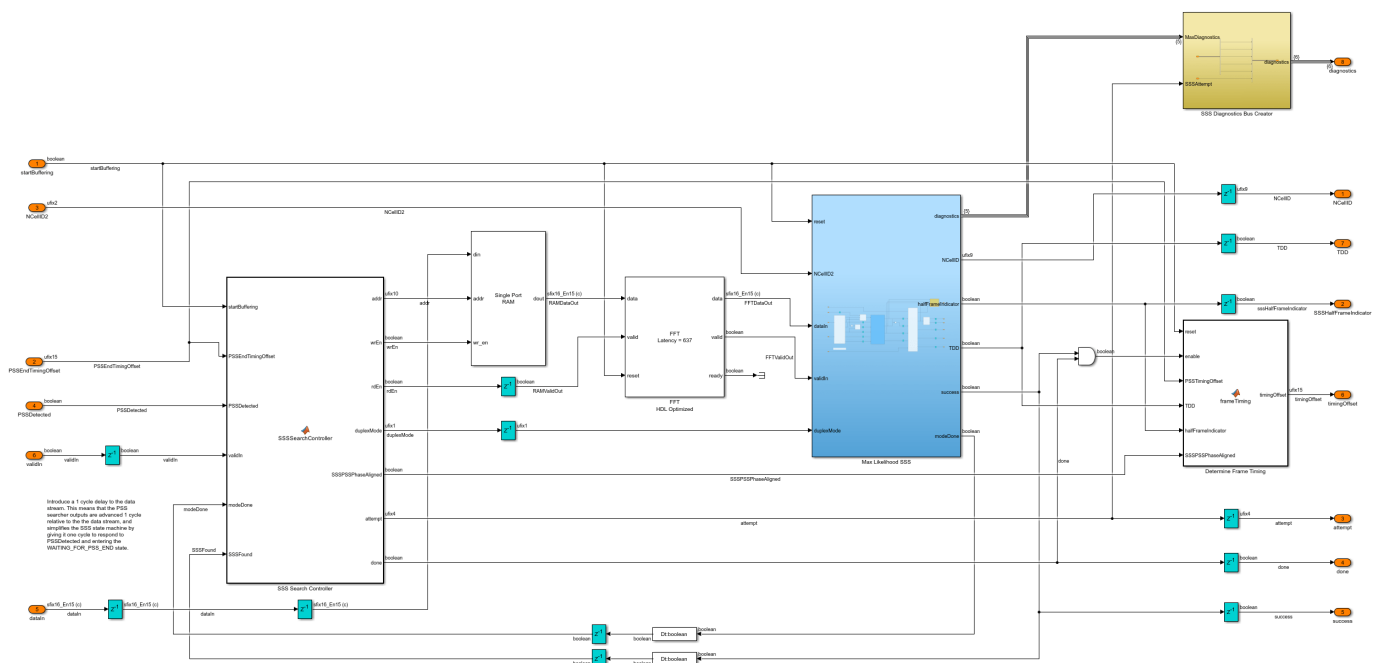
SamplingRate = 30.72e6;
simParams.Ts = 1/SamplingRate;

% Choose to load a captured off-the-air waveform from a file,
% or generate a test waveform with LTE Toolbox.
loadfromfile = true;

if loadfromfile
    % Load captured off-the-air waveform.
    load('eNodeBWaveform.mat');
    dataIn = resample(rxWaveform,SamplingRate,fs);
else
    % Generate a test waveform with LTE Toolbox.
    dataIn = hGenerateDLRXWaveform();
end

% Scale signal level to be in the range -1 to +1.
dataIn = 0.95 * dataIn / max(abs(dataIn));

% Start 1 subframe into the waveform (chosen arbitrarily).
startIn      = false(length(dataIn),1);
```

```
startIn(1e-3*SamplingRate)  = true;

% Configure PSS and SSS attempts
PSSAttempts = 2;
SSSAttempts = 4;

% Determine stop time.
simParams.stopTime = (length(dataIn)-1)/SamplingRate;
```

Analysis Script

`ltehdlCellSearch_analyze.m` is called in the `StopFcn` callback of `ltehdlCellSearch.slx`. This script relies heavily on `ltehdlCellSearchTools.m` to analyze the model output and display the plots.

```
% ltehdlCellSearch model analysis script
% Post-processes model outputs and generates plots.

% Check if any simulation output exists to analyze.
if exist('out','var') && ~isempty(out.PSSDetected)

    % Post-process the model output to extract key cell parameters,
    % diganostics and signals.

    [signals, report] = ltehdlCellSearchTools.processOutput(dataIn,startIn,out);

    % Plot results

    ltehdlCellSearchTools.figure('Input waveform and search stages'); clf;
    ltehdlCellSearchTools.plotSearchStates(signals,report);

    ltehdlCellSearchTools.figure('Frequency estimation'); clf;
    ltehdlCellSearchTools.plotFrequencyEstimate(signals,report);

    ltehdlCellSearchTools.figure('PSS search'); clf;
    ltehdlCellSearchTools.plotPSSCorrelation(signals,report);

    ltehdlCellSearchTools.figure('SSS search');
    ltehdlCellSearchTools.plotSSSCorrelation(signals,report);

end
```

Analysis Tools Class

This class contains helper functions for analyzing and plotting model output. Refer to `ltehdlCellSearchTools.m` for more information.

Simulation Output and Analysis

To execute the simulation, use the run button in the **ltehdlCellSearch** model. Simulink will automatically call **ltehdlCellSearch_init** and **ltehdlCellSearch_analyze** via the **InitFcn** and **StopFcn** callbacks respectively. Note that it will take a while to build the **ltehdlDownlinkSyncDemod** model reference on the first run. The simulation generates two main types of output: (i) **Display** blocks at the top level of the **ltehdlCellSearch** block diagram show key detection parameters, and (ii) four plots are generated at the end of the simulation.

The *NCellID*, *TDDMode*, *timingOffset*, *freqEst*, *cellDetected*, and *cellSearchDone* outputs all have associated **Display** blocks. Their values are shown below at the end of a simulation which used the captured off-the-air waveform (`eNodeBWaveform.mat`) as stimulus.

Cell Search HDL

The **Input waveform and search stages** plot shows:

- The magnitude of the input waveform vs time.
- The time window during which frequency estimation occurs.
- The PSS search window for each attempt (one in this case) and the location of the detected PSS.
- The SSS search windows for TDD and FDD for each attempt (one in this case), and the location of the detected SSS.

The **Frequency estimation** plot shows the output of the frequency estimator vs time. At the end of the 10 ms frequency estimation time window, the frequency estimate is loaded into a register and used to correct the frequency offset. This value is also shown on the plot.

The cell ID is made up of two components, `NCellID1` and `NCellID2`, where `NCellID1` is the SSS sequence number, and `NCellID2` is the PSS sequence number (See Appendix A). The **PSS search** plot shows all three PSS correlator outputs, and the PSS threshold. PSS was detected approximately 17 ms into the waveform on PSS #1, therefore `NCellID2 = 1`.

The **SSS search** plot shows the correlation metrics for the successful SSS detection attempt, and the SSS threshold. As previously discussed, the SSS detection algorithm determines the duplex mode and half frame position, as well as the cell ID. As a result, 4*168 = 672 correlation metrics are computed during each attempt. The correlation metrics are shown in the following order along the x-axis:

- FDD1: metrics at the FDD location for SSS sequences corresponding to 1st half frame
- FDD2: metrics at the FDD location for SSS sequences corresponding to 2st half frame
- TDD1: metrics at the TDD location for SSS sequences corresponding to 1st half frame
- TDD2: metrics at the TDD location for SSS sequences corresponding to 2st half frame

SSS was detected in the FDD location for SSS sequence corresponding to the 1st half frame. The SSS sequence number is 25 therefore `NCellID1 = 25`. The final cell ID is therefore:

`NCellID = 3*NCellID1 + NCellID2 = 76.`

HDL Code Generation and Verification

To generate the HDL code for this example you must have an HDL Coder™ license. Use the `makehdl` and `makehdltb` commands to generate HDL code and an HDL testbench for the **Cell Search HDL** subsystem. Note that testbench generation can take a while due to the length of the tests vectors that are generated.

The **Cell Search HDL** subsystem was synthesized on a Xilinx® Zynq®-7000 ZC706 evaluation board. The post place and route resource utilization results are shown in the table below. The design met timing with a clock frequency of 200 MHz.

```
     Resource        Usage
 _____      _____
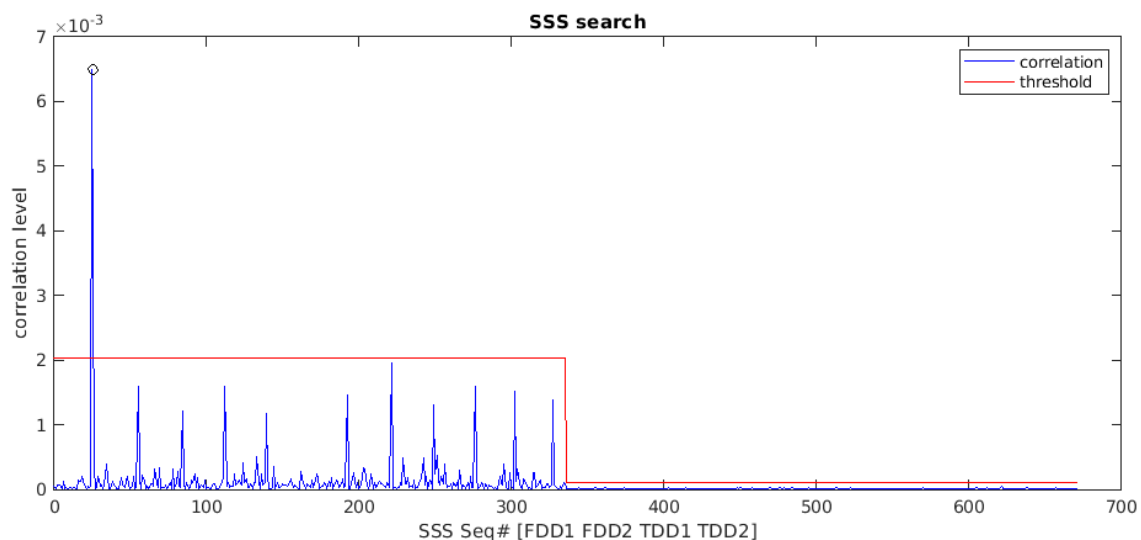
 Slice Registers     26274
 Slice LUTs          15740
 RAMB18                 25
 RAMB36                  9
 DSP48                  88
```

Appendix A - LTE Downlink Synchronization Signals

LTE provides two *physical signals* to aid the cell search and synchronization process. These are the Primary Synchronization Signal (PSS) and the Secondary Synchronization Signal (SSS).

The cell ID of the eNodeB is encoded in the PSS and SSS. The duplex mode, cyclic prefix length, and frame timing can be determined from their positions within the received signal. The PSS and SSS are transmitted twice every frame. There are 3 possible PSS sequences, and the eNodeB transmits the same PSS every half frame. For each PSS, there are 168 possible SSS sequences in the first half of the frame and 168 different possible SSS sequences in the second half of the frame. This means that once a SSS has been detected, the receiver knows if it is in the first or second half of a frame. The PSS and SSS sequences depend on the cell ID, therefore, there are 3 * 168 = 504 possible cell IDs. The cell ID is

`NCellID = 3*NCellID1 + NCellID2`

where NCellID2 is the PSS sequence number from 0 to 2, and NCellID1 is the SSS sequence number from 0 to 167. Each instance of the PSS occupies the central 62 subcarriers of one OFDM symbol, as does each instance of the SSS. For normal cyclic prefix mode the locations of the PSS and SSS signals are follows:

- FDD Mode: PSS is in symbol 6 of subframe 0, SSS is in symbol 5 of subframe 0
- TDD Mode: PSS is in symbol 2 of subframe 1, SSS is in symbol 13 of subframe 0

There are 14 symbols in each subframe, numbered from 0 to 13. Therefore, in FDD mode, the PSS is transmitted one OFDM symbol after the SSS, whereas in TDD mode the PSS is transmitted three OFDM symbols after the SSS. This difference in relative timing allows the receiver to discriminate between the two duplex modes. The positions of PSS and SSS within radio frames in FDD and TDD mode are illustrated below.

For more details see "Synchronization Signals (PSS and SSS)" (LTE Toolbox).

Appendix B - Cell Search and Selection Algorithm

This section describes the algorithm used by the model to detect eNodeB signals. The algorithm is designed to cope with real world conditions such as frequency offsets, noise and interference, and variation in the SNR of the PSS and SSS over time. To detect eNodeB in the presence of such conditions, the example uses three techniques:

1 Frequency recovery is applied prior to PSS and SSS detection.
2 Dynamic thresholds are used to validate the PSS and SSS correlation metrics and minimize the probability of false alarm.
3 Multiple attempts are made to detect the PSS and SSS; for example, if none of the correlation metrics for a specific instance of the SSS exceed the threshold, the detector will wait half a frame and try again, up to a predefined number of attempts.

Frequency Recovery

Frequency recovery is performed by utilizing the time domain structure of the received signal. In LTE (as with other OFDM based systems), each symbol consists of a *useful part* and a *Cyclic Prefix* (CP). The CP is generated by copying a small slice from the end of the symbol and prepending it to the start of the symbol. This can be exploited in a receiver by multiplying the received signal with the complex conjugate of a delayed version of itself, and then integrating across the CP duration, where the delay is the duration of the useful part. In effect, the received signal is cross-correlated with a delayed version of itself. The magnitude of the integrator output has peaks at symbol boundaries. The phase angle of the signal at these peaks is related to the frequency offset. This approach is used in the present example, and combined with additional averaging, to estimate the frequency offset. The algorithm can detect frequency offsets from -7.5 kHz to +7.5 kHz.

PSS Detection

PSS detection is performed by continuously cross-correlating the received signal with all three possible PSS sequences in the time domain. In addition, the energy of the signal within the span of the correlators is computed on each time step and then scaled to generate a threshold. The PSS detection algorithm aims to pick the strongest cell by picking the maximum PSS correlation metric within a 10 ms time window. The following pseudocode describes the search algorithm:

```
initialize position of first 10 ms search window

for k = 1 to 4 (number of PSS attempts)

    find correlation levels which exceed the threshold
    if any correlation levels exceed the threshold
        find the max correlation level of those which exceed the threshold
        PSS detected: break loop and start SSS search
    else
        PSS not detected: move search window to next 10ms period
    end
end
```

SSS Detection

Once PSS is located, the detector can narrow down the position of the SSS to two possible locations; one for FDD and one for TDD. The SSS correlation metrics are computed in the frequency domain, by evaluating the dot product of the sequence. The following algorithm is used to search for and select an SSS sequence.

```
initialize SSS search window

for k = 1 to 8 (number of SSS attempts)

    for each duplex mode in [FDD, TDD]
        extract 128 point search window for current duplex mode
        compute FFT and extract SSS subcarriers
        compute correlation metrics for SSS sequences corresponding to 1st half frame
        compute correlation metrics for SSS sequences corresponding to 2nd half frame
        compute signal energy-based threshold
    end

    discard correlation metrics which do not exceed the threshold
    if any metrics exceeded the threshold
        pick maximum correlation metric from surviving metrics
        SSS detected: break loop and proceed to next processing stage
    else
        SSS not detected: move SSS search window later by half a frame
    end

end
```

Cell Search Illustration

The cell search algorithm is shown below for a scenario where PSS and SSS each take 2 attempts to detect valid signals. The figure also shows the frequency recovery stage. Initially, the receiver has no knowledge of the received signal frame timing. In the Simulink model (and on hardware), a *start* input is used to trigger the detection process. The receiver begins by measuring the frequency offset, which takes 10 ms. Next, the first 10 ms PSS search takes place. In this case, no PSS is detected, therefore a second PSS search is initiated. This time PSS is detected. The first SSS search takes place just short of 10 ms after the location of the detected PSS, avoiding the need to buffer significant amounts of data, and making the algorithm hardware friendly. As shown, SSS also takes two attempts in this case. From the location of the detected SSS, the receiver knows the duplex model (FDD in this case) and the frame timing.

References

1. 3GPP TS 36.214 "Physical layer"

See Also

Related Examples

- "LTE HDL MIB Recovery" on page 5-141
- "LTE HDL SIB1 Recovery" on page 5-123
- "LTE HDL PBCH Transmitter" on page 5-152

LTE HDL SIB1 Recovery

This example shows how to design an HDL optimized receiver that can recover the first System Information Block (SIB1) from an LTE downlink signal.

Introduction

This design builds on the "LTE HDL MIB Recovery" on page 5-141, adding the processing required to decode SIB1. It is based on the LTE Toolbox™ "Cell Search, MIB and SIB1 Recovery" (LTE Toolbox).

In order to decode the SIB1 message, additional steps are required after the MIB (Master Information Block) has been decoded. This design adds functionality to locate and decode the PCFICH (Physical Control Format Indicator Channel), the PDCCH (Physical Downlink Control Channel), and the PDSCH (Physical Downlink Shared Channel). The extensible architecture used in the "LTE HDL MIB Recovery" on page 5-141 allows the design to be expanded, while reusing the core functionality of the MIB recovery implementation. This design can be implemented on SoC platforms using hardware-software co-design and hardware support packages. See "Deploy LTE HDL Reference Applications on FPGAs and SoCs" on page 5-168.

Summary of SIB1 Processing Stages

The initial stages of SIB1 recovery are the same as for the "LTE HDL MIB Recovery" on page 5-141, composed of the cell search, PSS/SSS detection, OFDM demodulation, and MIB decoding. LTE signal detection, timing and frequency synchronization, and OFDM demodulation are performed on the received data, providing information on the subframe number, duplex mode, and cell ID of the received waveform. The received data is buffered into the grid subframe memory buffer and, once a complete subframe has been stored in the memory, the channel estimate is calculated. The channel estimate can then be used to equalize the grid as data is read out from the buffer. When subframe 0 has been stored in the buffer, and the channel estimate calculated, the Physical Broadcast Channel (PBCH) can then be retrieved from the grid, equalized, and decoded, recovering the MIB message.

The MIB message contains a number of parameters which are required to decode the subsequent channels. One of these parameters is the System Frame Number (SFN). The SFN is required to determine the location of the SIB1 message, since the SIB1 message is only sent in even numbered frames (mod(SFN,2) = 0). Hence, if the MIB message was decoded within an odd frame, the receiver must wait until the next even frame before attempting to decode the SIB1. When the receiver has decoded the MIB message, and has received subframe 5 of an even frame, an attempt at decoding the SIB1 can be made.

The MIB message also provides the NDLRB system parameter, indicating the Number of Downlink Resource Blocks used by the transmitter. For different NDLRB values (different bandwidths) the number of active subcarriers is different. Hence the NDLRB affects the indexing of the resource grid memory for each of the channels processed after the PBCH.

NDLRB is first used to calculate the Resource Elements (REs) allocated to the Physical Control Format Indicator Channel (PCFICH), and the corresponding symbols can be retrieved from the resource grid. The PCFICH Decoder then attempts to decode the CFI data using the symbols retrieved from the resource grid.

The CFI indicates the number of OFDM symbols allocated to the Physical Downlink Control Channel (PDCCH). The CFI, in conjunction with the MIB parameters NDLRB, PHICH Duration, and Ng, is used to calculate which Resource Elements (REs) are allocated to the PDCCH. These REs are requested

from the grid, and passed to the PDCCH decoder. If the signal being decoded is using Time Division Duplexing (TDD) the PDCCH allocation varies based on the TDD configuration used. Because the TDD configuration is not know at this point, each of the TDD configurations that affect the PDCCH allocation are tried, until successfully decoding.

Once the PDCCH has been decoded, a blind search of the PDCCH common search space is conducted to find the DCI (Downlink Control Information) message for the SIB1. This DCI message has a CRC scrambled with the SI-RNTI (System Information Radio Network Temporary Identifier) and carries information about the allocation and encoding of the SIB1 message within the PDSCH. The search operation blindly attempts to decode DCI messages with a number of possible formats, from a number of candidates. If the signal being decoded is using TDD and a DCI message is not found during the search, then PDCCH decoding will be re-attempted for any untried TDD configurations.

Once located, the DCI message is parsed, giving the DCI allocation type, RIV, and Gap parameters required for the PDSCH resource allocation calculation. The Physical Resource Blocks (PRBs) allocated to the SIB1 message within the PDSCH can then be calculated. Parsing the DCI message also provides information on the transport block length and redundancy versions required to decode the PDSCH.

Using the PRB allocation information the REs allocated to the SIB1 message within the PDSCH can be calculated. The PDSCH decoding then processes the data retrieved from the resource grid. If decoding is error free the SIB1 message bits are returned.

Architecture and Configuration

The architecture is designed to be extensible, allowing channel processing subsystems to be added, removed, or exchanged for alternative implementations. This extensibility is illustrated by the additions made to the MIB design to produce the SIB1 design. The core functionality is the same, with additional processing and control added for the three extra channels required to decode the SIB1.

To allow reuse and sharing of the main subsystems of the model, the example uses "Model References". Model referencing allows for unit testing of each of the subsystems, and for the models to be instantiated in multiple different examples. The LTE HDL Cell Search, LTE HDL MIB Recovery and LTE HDL SIB1 recovery all share reference models.

- Cell search, synchronization and OFDM demodulation perform initial stages of detecting a downlink signal and synchronization. Unequalized grid data is streamed out to be buffered in the grid memory for further processing.

- The central resources of the grid memory, channel estimation, and channel equalization are grouped together, with an interface such that data can be requested by providing an address to the grid, and equalized symbols are output for processing by the decoding stages.

- The indexing subsystems request data from the grid by providing a subcarrier number, an OFDM symbol number, and a read enable flag. These signals are grouped together in a bus for easier routing in the Simulink model. Only one indexing subsystem can access the grid at a time. A controller is used to avoid contention and enable the indexing subsystems at the correct time. Each of the indexing subsystems has a corresponding decoding subsystem, which attempts to decode the data requested from the grid by the indexing subsystem.

- The decoding subsystems receive equalized complex symbols from the grid, with a signal indicating when the incoming data is valid. The decoding subsystems must be enabled before they will start to process valid samples at the input, and it is expected that only one of the decoding subsystems will be enabled at any point in time. A central controller for the SIB1 decoder enables the decoding subsystems at the appropriate time.

- The control subsystem tracks the state of the decoder and enables the decoding and indexing subsystems in the correct sequence using the done, valid, detected, and error signals (as appropriate) for the various processing stages.

- The DCI resource allocation function (ltehdlDCIResourceAllocation) was selected for implementation on software, as part of a hardware/software co-design implementation. This function was selected due to the low frequency of calculation, and the complex loop behavior making it inefficient to implement in hardware.

Structure of Example Model

The top-level of the **ltehdlSIB1Recovery** model is shown in the figure below. The **HDL LTE SIB1** subsystem supports HDL code generation. The **SW DCI Resource Allocation** subsystem represents the software portion of a design partitioned for hardware/software co-design implementation. The **stateViewer** MATLAB Function block generates text information messages based on the *decoderState* signal from the **HDL LTE SIB1**, and prints this information to both the Simulink Diagnostic Viewer and to a MATLAB figure window. The **stateViewer** also produces the *stopSimulation* signal, which stops the simulation when the decoder reaches a terminal state, as indicated by the text information messages.

LTE HDL SIB1 Recovery Example

SW DCI Resource Allocation

The **SW DCI Resource Allocation** subsystem contains an instance of the
ltehdlDCIResourceAllocation model. Buses are used here to facilitate signal routing to and from
this subsystem.

dciRecourceAllocation

The **ltehdlDCIResourceAllocation** model reference performs parsing of the DCI message bits, generates the DCI parameters, then uses the DCI parameters to perform the DCI Physical Resource Block (PRB) allocation calculation. These operations are equivalent to the LTE Toolbox functions `lteDCI` and `lteDCIResourceAllocation`. Due to the complexity of the PRB allocation calculation, this part of the design was selected for implementation in software, as an HDL implementation would require a large amount of hardware resources.

HDL LTE SIB1

The **HDL LTE SIB1** subsystem contains 2 subsystems. The **Downlink Sync Demod** subsystem is an instance of the **ltehdlDownlinkSyncDemod** model, which is described in the "LTE HDL Cell Search" on page 5-106 example. It performs the cell search, timing and frequency synchronization, and OFDM demodulation. The **HDL MIB + SIB1 Decoder** subsystem performs the channel decoding operations required to decode the MIB and SIB1 messages, as described below.

HDL MIB + SIB1 Decoder

The **HDL MIB + SIB1 Decoder** structure can be seen below. It receives OFDM demodulated grid data from the **Downlink Sync Demod** subsystem, and stores the data in a subframe buffer, **Resource Grid Memory**. It then calculates the channel estimate for the received data in the **Channel Estimation** subsystem and uses this to equalize data as it is read out of the **Resource Grid Memory**. A series of channel decoding steps are then performed in order to decode the SIB1 message. In total there are 10 referenced models at this level of hierarchy: 4 channel decoders, 4 channel index generation subsystems, and 2 subsystems performing resource grid buffering, channel estimation, and equalization.

The **PBCH Indexing**, **Resource Grid Memory**, **Channel Equalization** and **MIB Decoder** all instantiate the same referenced models used in the MIB example. For more detailed information about the operation of these referenced models, refer to "LTE HDL MIB Recovery" on page 5-141.

Indexing Subsystems

There are 4 indexing subsystems, corresponding to the 4 channels that need to be decoded in order to receive a SIB1 message: PBCH, PCFICH, PDCCH, and PDSCH. Each of the indexing subsystems has a corresponding decoding subsystem. The indexing subsystems use an address bus, consisting of a read address corresponding to the subcarrier number, a read bank corresponding to an OFDM symbol, and a read enable signal to control access to the grid. The **read_selector** MATLAB Function block selects between the outputs of the 4 indexing subsystems based on the read enable signal. It is assumed that only one indexing subsystem will attempt to read from the grid at any point in time, with the **CONTROL** subsystem in charge of enabling the indexing subsystems at the appropriate time.

PBCH Indexing

The **PBCH Indexing** block references the **ltehdlPBCHIndexing** model. It performs the index generation for the PBCH and is equivalent to the LTE Toolbox function `ltePBCHIndices`.

PCFICH Indexing

The **PCFICH Indexing** block references the **ltehdlPCFICHIndexing** model. It generates the indices required to read the PCFICH symbols from the grid memory and is equivalent to the LTE Toolbox function `ltePCFICHIndices`. The PCFICH is always in the first OFDM symbol (the first memory bank of the grid buffer) and is 16 symbols in length, in 4 groups of 4 symbols. The 4 groups of symbols are distributed at quarters of the occupied bandwidth, with an offset dependent on the Cell ID.

PDCCH Indexing

The **PDCCH Indexing** subsystem generates the indices required to read the PDCCH symbols from the grid memory. It references the **ltehdlPDCCHIndexing** model and is equivalent to the LTE Toolbox functions `ltePDCCHIndices` and `ltePDCCHDeinterleave`. The PDCCH spans between 1 and 4 OFDM symbols, as defined by the value decoded from the PCFICH. The number of subcarriers spanned by the PDCCH depends on NDLRB. As a result, the number of symbols read from the grid varies, which is indicated by the *nSymbols* output. The PDCCH occupies all of the OFDM symbols indicated by the CFI, but must exclude any locations which have already been allocated to other channels, such as the PCFICH and PHICH. The main indexing calculation is performed by the **PDCCH_Index_Gen** subsystem. It calculates the locations of the PCFICH and PHICH then excludes these locations from the range of indices occupied by the PDCCH. In TDD mode number of symbols occupied by the PHICH varies based on the TDD configuration. For different TDD configurations there are three possible values of mi (0, 1, and 2), as specified in section 6.9 of [1], which is a multiplier to the size of the region allocated to the PHICH. When in the duplexing mode is FDD, mi is

always 1. The size of the PDCCH in terms of both quadruplets (groups of 4 symbols) and symbols is given by the *Mquad* and *Msymb* outputs.

The **ramAddrCalc** and **lk_ram** subsystems are used to perform a cyclic shift on the quadruplets using the *cellID*. Because the DCI message for SIB1 is always transmitted in the common search space of the PDCCH, it is possible to reduce the number of symbols that are read from the grid memory by retrieving only the symbols from the common search space. In order to do this the PDCCH deinterleaving operation is performed, and the first 576 symbols are requested from the grid. If there are less than 576 symbols in the PDCCH then all of the symbols will be requested. In LTE Toolbox, the PDCCH deinterleaving operation is performed as part of the `ltePDCCHDecode` function. However, as this function simply re-orders the data and does not change the data content, it is possible to move this processing stage to an earlier point in the receiver. By moving the deinterleaver to act on the indices, rather than the data, and reducing to the common search space after deinterleaving, the memory requirements for the deinterleaver and the PDCCH decoder are reduced.

PDSCH Indexing

The **PDSCH Indexing** calculates the locations of the PDSCH in the grid memory based on the Physical Resource Block (PRB) set, which is passed to this block from the DCI resource allocation calculation in the **SW DCI Resource Allocation** subsystem. The **PDSCH Indexing** is an instance of the **ltehdlPDSCHIndexing** model and is equivalent to the LTE Toolbox function `ltePDSCHIndices`. The PDSCH occupies all of the symbols in the PRB set which have not previously been allocated to another channel. Hence the PDSCH indexing function must exclude any locations which are allocated to the PSS and SSS, and all of the control channel region (i.e. the OFDM symbols indicated by the PCFICH). As the SIB1 message always occurs in subframe 5 of an even frame, there is no need to exclude the PBCH locations, as these only occur in subframe 0.

Decoder Subsystems

There are 4 decoder subsystems, each of which has a corresponding indexing subsystem. When enabled, the decoder subsystems process equalized symbols from the **Channel Equalization** subsystem, performing the operations required to decode the channel. The **CONTROL** subsystem enables each of the decoder subsystems at the appropriate time. The outputs from each of the decoder subsystems are used to locate and decode subsequent channels in the chain. To ensure that this information is available when required, each of the decoder subsystems registers the decoded information at the output, for later access. The output registers are cleared using the *clearOutputReg* input on each of the decoders.

MIB Decoder

The **MIB Decoder** uses the same referenced model, **ltehdlPBCHDecoder**, that is used in the LTE HDL MIB example. It performs the PBCH and BCH decode operations, equivalent to the LTE Toolbox functions `ltePBCHDecode` and `lteMIB`. The outputs from this block provide the information required to locate and decode the channel information for the subsequent channels.

CFI Decoder

The **CFI Decoder** uses the **ltehdlPCFICHDecoder** referenced model. It performs the PCFICH and CFI decode operations equivalent to the `ltePCFICHDecode` and `lteCFIDecode` functions in LTE Toolbox. The input from the **Channel Equalization** is the 16 symbols requested by the **PCFICH Indexing**. The **PCFICH Decoder** subsystem performs descrambling and QPSK demodulation on the 16 PCFICH symbols to produce 32 soft bits. The **CFI Extraction** subsystem then correlates the soft bits with the three CFI codewords. The codeword with the strongest correlation gives the CFI value of 1, 2, or 3. The CFI value indicates the number of OFDM symbols occupied by the PCFICH. If NDLRB is greater than ten, the number of OFDM symbols is equal to the CFI value (1, 2, or 3). If NDLRB is less than or equal to ten, the number of OFDM symbols is one larger than the CFI value (2, 3, or 4). This information is used by the **PDCCH Indexing** and **PDSCH Indexing** subsystems.

PDCCHDecodeSearch

The **PDCCHDecodeSearch** subsystem uses the **ltehdlPDCCHDecode** referenced model. It performs the PDCCH decode, blind PDCCH search, and DCI decode operations required to locate and decode the SIB1 DCI message within the PDCCH. This is roughly equivalent to the LTE Toolbox functions `ltePDCCHDecode`, `ltePDCCHSearch`, and `lteDCI` (which is used within `ltePDCCHSearch`) with a few modifications. As the SIB1 DCI message is always within the common search space of the PDCCH, only these symbols are retrieved from the grid buffer, as described above for **PDCCH Indexing**. The SIB1 DCI message is always DCI format 1A or 1C. It is found in the PDCCH common search space using PDCCH aggregation levels 4 or 8, and the CRC for the DCI message is scrambled with the System Information Radio Network Temporary Identifier (SI-RNTI). Using this information the search can be simplified compared to the LTE Toolbox `ltePDCCHSearch` implementation. For more information on the LTE Toolbox PDCCH search process, see the "PDCCH Blind Search and DCI Decoding" (LTE Toolbox) example. The **PDCCHSearch** subsystem blindly attempts to decode DCI messages from all of the possible candidates and combinations within the common search space until a DCI message with the correct CRC mask is decoded, indicating that the SIB1 DCI message has been found, or all candidates have been attempted, and no SIB1 DCI message has been found. When a SIB1 DCI message has been found, the search stops, and the information from the decoded DCI message is returned from the block. This information is then passed to the **SW DCI Resource Allocation** subsystem to parse the DCI message, and determine which resources in the PDSCH have been allocated to the SIB1 message.

The **demod/descramble** subsystem performs descrambling and QPSK demodulation, while the **PDCCHSearch** subsystem performs the search process described in more detail below.

PDCCHSearch

Within the **PDCCHSearch** subsystem there are a number of processing stages which combine to perform the PDCCH search operation. The **pdcchSearchControl** MATLAB Function block writes the incoming data to the **PDCCH RAM**, then controls the search process, iterating through the different combinations of DCI format, PDCCH format, and PDCCH candidates. The **dciControl** MATLAB Function block generates the read addresses for the **PDCCH RAM** given the PDCCH candidate number and size. The **pdcchRateRecovery** MATLAB Function block is equivalent to the LTE Toolbox function `lteRateRecoverConvolutional`, performing the deinterleaving and rate recovery for the convolutional decoder. The **dciDecode** subsystem performs the convolutional decoding of the rate recovered bits, then checks the message CRC with the SI-RNTI to determine if a SIB1 DCI message has been found. If successfully decoded, the DCI message bits are buffered and output, and the search process is stopped. The PDCCH search process will also stop if all of the possible candidates have been checked, but no DCI message for SIB1 has been found, with the *error* output being asserted.

PDSCHDecode

The **PDSCHDecode** subsystem uses the **ltehdlPDSCHDecode** referenced model. It is equivalent to the `ltePDSCHDecode` and `lteDLSCHDecode` functions in LTE Toolbox. The **QPSKDemod** and **PDSCHPRBS** demodulate the incoming signals and generate the descrambling sequence. The descrambled bits are then passed to **TurboRateRecovery** which performs deinterleaving and rate recovery of the incoming bits. The **SampleControlBusGeneration** subsystem generates the control signals required to interface with the **LTE Turbo Decoder** and **LTE CRC Decoder**, which decode the signal. The **LTE CRC Decoder** indicates the status of the CRC decode, asserting the *err* signal, along with the *end* signal in the *ctrl* bus output, if errors have been detected. If the CRC does not detect any errors then the SIB1 message has been successfully decoded, and the *sib1_bits* are streamed out from the block, with *bitsValid* indicating when *sib1_bits* are valid. Once the SIB1 message has been detected, and the bits output from **PDSCHDecode**, the simulation stops. No attempt is made to combine the different Redundancy Versions (RVs) of the DLSCH.

CONTROL Subsystem

The **CONTROL** subsystem tracks the state of the decoder through the different channel processing stages, enabling each of the indexing and decoding subsystems in turn. The subframe number and frame number are taken as inputs, allowing the **frameCount** function to track the System Frame Number (SFN). The subframe and frame numbers are used to determine when channels will be

available for decode (e.g. SIB1 is only transmitted on subframe 5 of even numbered frames). The **decoderState** MATLAB Function block implements a simple state machine that keeps track of which processing stages have been completed, and which stage to enable next. The state of the decoder is output from the controller, and is parsed by the **stateViewer** MATLAB Function block at the top level of the model to produce human readable messages.

When the received signal is in TDD mode the **CONTROL** subsystem manages the blind search of each of the TDD configurations, running the **PDCCH Indexing** and **PDCCH Decoding** subsystems for each of the three possible *mi* values. The different *mi* values {0,1,2} result in different PHICH allocations, hence different PDCCH allocations. The PDCCH allocations are calculated, and the PDCCH decode attempted for each mi value, until a SIB1 DCI message is found, or all of the possibilities are exhausted.

Results and Display

The simulation model is configured to stop the simulation under a number of conditions:

- If the cell search does not find any cells.

- If the MIB detection has an error.

- If a SI-RNTI DCI message is not detected during the PDCCH search.

- At the end of the PDSCH decoding attempt.

If the SIB1 message is successfully decoded, it is output from the *sib1Bits* port, with the *sib1BitsValid* port indicating when the output is valid. The data is buffered and sent to the MATLAB workspace.

The LTE HDL SIB1 Receiver State Information figure window displays text messages indicating the current state of the decoder. The state of the system is tracked by the **CONTROL** subsystem, with the *decoderState* signal passed up to the top level of the model where the **statePrint** MATLAB Function block generates the text info messages.

The display blocks in the top level of the model show some of the key parameters decoded by each of the channel processing stages. A number of the key control signals, from within the **CONTROL** subsystem, are logged for viewing with the logic analyzer.

HDL Code Generation and Verification

To generate the HDL code for this example you must have an HDL Coder™ license. Note that test bench generation for this example takes a long time due to the length of the simulation required to create the test vectors.

HDL code for the **HDL LTE SIB1** subsystem was generated using the HDL Workflow Advisor IP Core Generation workflow for a Xilinx® Zynq®-7000 ZC706 evaluation board, and then synthesized. The post place and route resource utilization results are shown below. The design met timing with a target clock frequency of 150MHz. Using the workflow advisor IP core generation workflow allows the input and output ports to be mapped to AXI4-Lite registers, reducing the number of FPGA IO pins required, and allows the design to be split between hardware and software.

```
        Resource        Usage
    _____     _____
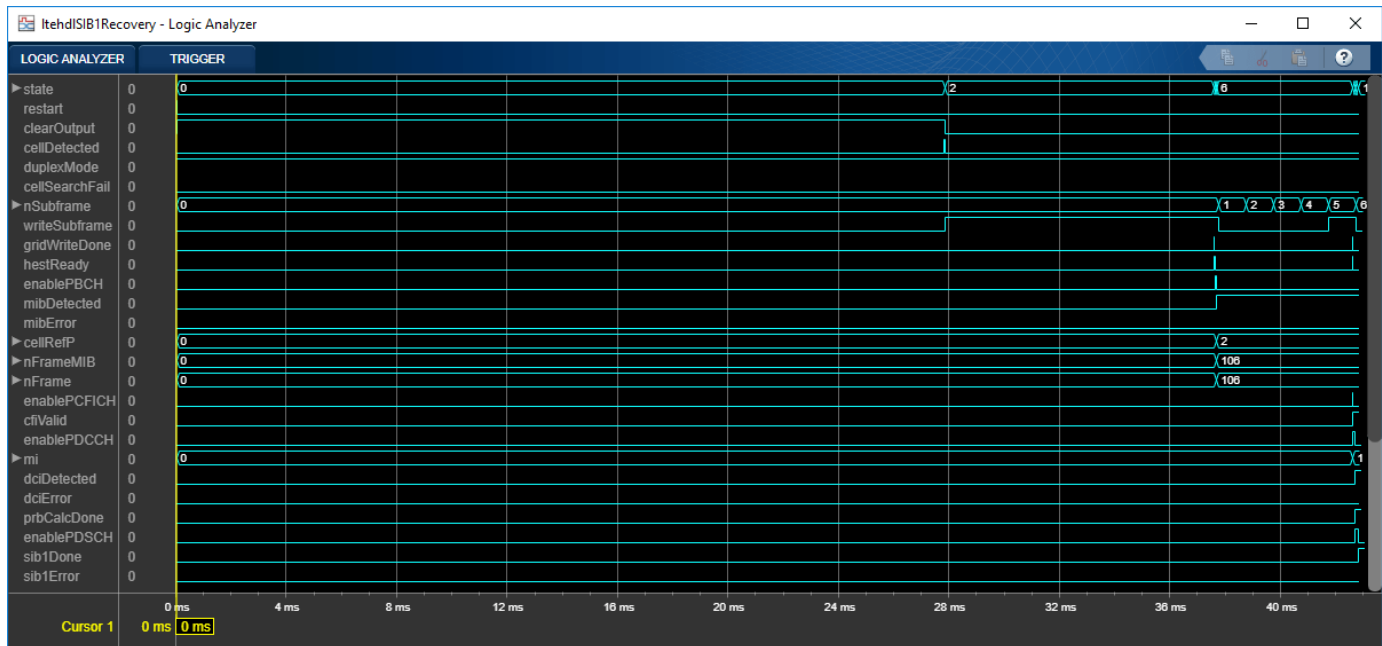
    Slice Registers     104299
    Slice LUTs           64678
    RAMB18                  55
    RAMB36                 191
    DSP48                  138
```

For more information see "Prototype Wireless Communications Algorithms on Hardware" on page 2-21

Limitations

The **stateViewer** MATLAB function block is not supported for simulation in rapid accelerator mode. This block can be removed or commented out if rapid accelerator simulation is required.

References

1. 3GPP TS 36.211, "Physical Channels and Modulation"

See Also

Related Examples

- "LTE HDL Cell Search" on page 5-106
- "LTE HDL MIB Recovery" on page 5-141

LTE HDL MIB Recovery

This example shows how to design an LTE MIB recovery system optimized for HDL code generation and hardware implementation.

Introduction

The model presented in this example can be used to locate and decode the MIB from LTE downlink signals. It builds upon the "LTE HDL Cell Search" on page 5-106 example, adding processing stages to decode the MIB. The Master Information Block (MIB) message is transmitted in the Physical Broadcast Channel (PBCH), and carries essential system information:

- Number of Downlink Resource Blocks (NDLRB), indicating the system bandwidth
- System Frame Number (SFN)
- PHICH (Physical HARQ Indicator Channel) Configuration

The design is optimized for HDL code generation and the architecture is extensible, allowing additional processing stages to be added, such as indexing and decoding for the PCFICH, PDCCH and PDSCH (see "LTE HDL SIB1 Recovery" on page 5-123). This design can be implemented on SoC platforms using hardware-software co-design and hardware support packages. See "Deploy LTE HDL Reference Applications on FPGAs and SoCs" on page 5-168.

MIB Processing Stages

In order to decode the MIB message this example performs these operations:

- Cell search and OFDM demodulation
- Buffering grid data
- Channel estimation and equalization
- PBCH Indexing - locating PBCH within the grid
- PBCH Decoding - decoding PBCH, BCH, and MIB

Cell Search and OFDM Demodulation

LTE signal detection, timing and frequency synchronization, and OFDM demodulation are performed on the received data. This produces the grid data and provides information on the subframe number and cell ID of the received waveform. The MIB message is always carried in subframe 0, and the cellID is used to determine the location of the cell-specific reference signals (CRS) for channel estimation, as well as being used to initialize the descrambling sequence for PBCH Decoder.

Buffering Grid Data

As the MIB message is always carried in subframe 0 of the downlink signal, subframe 0 is buffered in a memory bank. At the same time as the subframe is being written to the memory bank, the location of the CRS are calculated using the cellID, and CRS are sent to the channel estimator.

Channel Estimation

The CRS from the received grid are then compared to the expected values, and the phase offset calculated. The channel estimates for each CRS are averaged across time, and linear interpolation is used to estimate the channel for subcarriers which do not contain CRS. The channel estimate for the subframe is used to equalize data when it is read from the grid memory.

5-141

PBCH Indexing

The PBCH is always allocated to the central 6 Resource Blocks (RBs) of subframe 0, within the first 4 OFDM symbols of the 2nd slot. It occupies all of the Resource Elements (REs) within this region, excluding the locations allocated to CRS. The locations of the CRS are calculated using the cellID, then the addresses of the REs occupied by the PBCH can be calculated (240 locations in total), and the data retrieved from the grid memory bank.

PBCH Decoding

As the PBCH data is read from the grid memory bank it is equalized using the channel estimate. The 240 equalized PBCH symbols are buffered, and PBCH and BCH decoding are attempted for each of the 4 possible versions of the MIB within a PBCH transport block. Each of these versions requires a different descrambling sequence, so descrambling, demodulation, rate recovery, convolutional decoding, and CRC check must be attempted for each. If successfully decoded, the CRC value gives the cellRefP value - the number of transmit antennas, and the MIB bits can be parsed to give the system parameters.

Model Architecture

The architecture of the LTE HDL Cell Search and MIB Recovery implementation is shown in the diagram below.

The input to the receiver is baseband I/Q data, sampled at 30.72 Msps. A 2048-point FFT is used for OFDM demodulation, and is sufficient to decode all of the supported LTE bandwidths. The resource grid buffer is capable of storing one subframe of LTE data. Once the receiver has synchronized to a cell, data from the OFDM demodulator is written into the grid buffer. The PBCH indexing block then generates the indices of the resource elements which carry the PBCH. Those resource elements are read out of the grid buffer and equalized, before being passed through the PBCH decoder. This architecture is designed to be extensible and scalable so that additional channel indexing and decoding functions can be inserted as needed. For example it can be extended to perform SIB1 recovery as shown in the "LTE HDL SIB1 Recovery" on page 5-123 example.

The top level of the **ltehdlMIBRecovery** model is shown below. HDL code can be generated for the **HDL LTE MIB Recovery** subsystem.

The `ltehdlMIBRecovery_init.m` script is executed automatically by the model's `InitFcn` callback. This script generates the *dataIn* and *startIn* stimulus signals as well as any of the constants needed to initialize the model. Input data can be loaded from a file which, for this example, is an LTE signal captured off the air. For information about capturing LTE signals off the air see "LTE Receiver Using Analog Devices AD9361/AD9364" (Communications Toolbox Support Package for Xilinx Zynq-Based Radio). Alternatively, an LTE waveform can be synthesized using LTE Toolbox functions. To select an input source, change the `loadfromfile` parameter in `ltehdlMIBRecovery_init.m`.

```
SamplingRate = 30.72e6;
simParams.Ts = 1/SamplingRate;
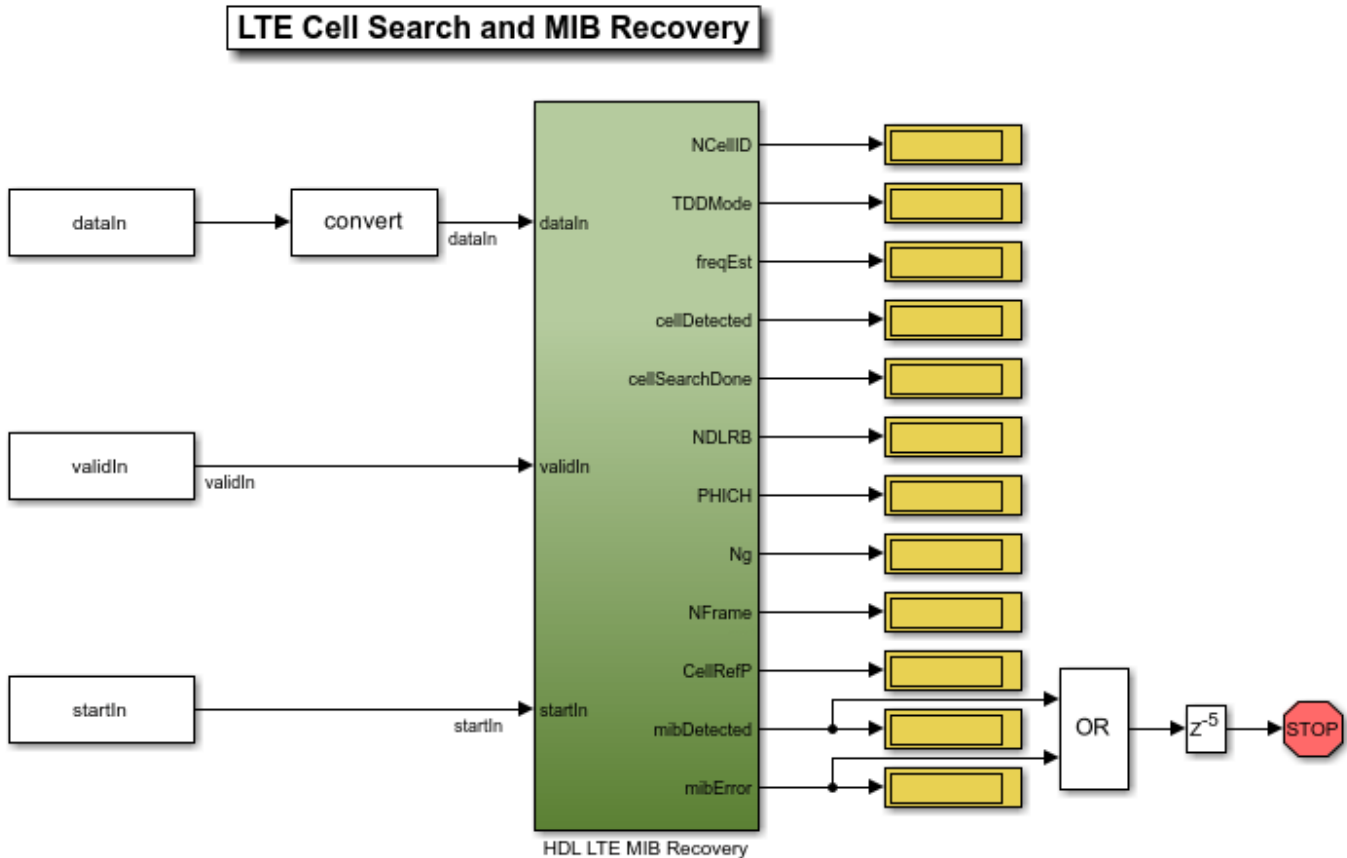
loadfromfile = true;

if loadfromfile
    load('eNodeBWaveform.mat');
    dataIn = resample(rxWaveform,SamplingRate,fs);
else
    dataIn = hGenerateDLRXWaveform();
end
```

HDL Optimized LTE MIB Recovery

The structure of the **HDL LTE MIB Recovery** subsystem is shown below. The **Downlink Sync Demod** block performs frequency and time synchronization, PSS/SSS signal detection, and OFDM demodulation. The **MIB Decoder** subsystem buffers subframe 0 of the incoming data, performs channel estimation, and attempts to decode the PBCH to recover the MIB information.

Downlink Synchronization and Demodulation

The **Downlink Sync Demod** subsystem takes in I/Q data at 30.72 Msps, and outputs the unequalized downlink resource grid data. It is an instance of the **ltehdlDownlinkSyncDemod** model reference, which implements the following functions:

- Frequency recovery
- Primary Synchronization Signal (PSS) detection
- Secondary Synchronization Signal (SSS) detection
- Timing recovery, based on the PSS and SSS signals
- OFDM demodulation (using a 2048 point FFT)
- Cell ID calculation, based on PSS and SSS detection results

The operation of the **ltehdlDownlinkSyncDemod** is described in more detail in the "LTE HDL Cell Search" on page 5-106 example.

MIB Decoder

The **MIB Decoder** subsystem is shown below. It consists of four subsystems: **PBCH Indexing**, **Resource Grid Memory**, **Channel Equalization**, and **PBCH Decoder**. The order of operations is as follows:

1. The *cellDetected* input is asserted, preparing the subsystem to receive and process data.
2. OFDM data is streamed into the **MIB Decoder** subsystem, and subframe 0 is stored in the **Resource Grid Memory**.
3. The **Channel Equalization** subsystem calculates a channel estimate for subframe 0
4. The **PBCH Indexing** block starts generating PBCH resource element indices.
5. Those resource elements are then read out of the **Resource Grid Memory** and equalized by the **Channel Equalization** block.
6. Finally the equalized PBCH data is passed through the **PBCH Decoder** block and the MIB is extracted.

Resource Grid Memory

The **Resource Grid Memory** block contains a memory bank, logic to control reading and writing of the grid memory bank, and logic to locate and output the CRS. The memory bank capacity is one subframe of demodulated OFDM data at the largest supported LTE bandwidth (20MHz).

The **MemoryBank Write Controller** is responsible for writing subframes of data to the memory bank. The *writeSubframe* input enables the write controller for the appropriate subframes; subframe 0 in the case of the present example. The **LTE Memory Bank** contains RAM of dimensions 14 x 2048 x 16 bit complex values; that is 14 ODFM symbols, each containing 2048 complex values. The **rsOutputGen** subsystem calculates the locations of the cell reference symbols, extracts these from the data as it is written to the grid memory, and outputs these via the *gridData* output signal.

The *gridData* output port carries the CRS signals, from **rsOutputGen**, when data is being written to the grid memory (*gridWriteDone* output port is low) and carries data from the **LTE Memory Bank** when the write to the grid memory is complete (*gridWriteDone* output port is high).

PBCH Indexing

The **PBCH Indexing** block computes the memory addresses required to retrieve the PBCH from the grid memory buffer. This is equivalent to the LTE Toolbox ltePBCHIndices function. The data retrieved from the grid memory is then equalized and passed to the **PBCH Decoder** for processing. The PBCH Indexing subsystem becomes active after the data for subframe 0 has been written to the grid memory, as indicated by the *gridWriteDone* output of the **Resource Grid Memory** subsystem. The PBCH is always 240 symbols in length, centered in the middle subcarriers, in the first 4 symbols within the 2nd slot of subframe 0.

Channel Estimation and Equalization

The **Channel Equalization** block contains three main subsystems. **cellRefGen** generates the cell-specific reference signal (CRS) symbols using a Gold Sequence generator. **chEst** performs channel estimation assuming two transmit antennas by using a simple, hardware-friendly channel estimation algorithm. **TxDivDecode** performs transmit diversity decoding to equalize the phase of the received data, using the channel estimates.

The channel estimator assumes the transmitter is using two antennas, generating a channel estimate for each antenna. For each antenna the channel estimator generates a single complex-valued channel estimate for each subcarrier of the subframe using the following algorithm:

1   Estimate the channel at each CRS resource element by comparing the received value to the expected symbol value (generated by **cellRefGen**).
2   Average these channel estimates across time (for the duration of the subframe) to generate a single complex-valued channel estimate for each subcarrier that contains CRS symbols.
3   Use linear interpolation to estimate the channel for subcarriers which do not contain CRS symbols.

The simple time average algorithm used for the channel estimation assumes low channel mobility. Therefore, the channel estimate may not be of sufficient quality to decode waveforms that were transmitted through fast fading channels. The algorithm also avoids using a division operation when calculating the channel estimate at each CRS. This means that the amplitude of the received signal will not be corrected, which is suitable for QPSK applications, but will not work for QAM, where accurate amplitude correction is required for reliable decoding.

Once the channel estimates are calculated for each of the transmit antennas they are used to equalize the *gridData* as it is read out from the **Resource Grid Memory**. **TxDivDecode** performs the inverse of the precoding for transmit diversity (as described in of TS 36.211 Section 6.3.4.3 [1]) and produces an equalized output signal, which is then passed to the **PBCH Decoder**.

PBCH Decoder

The **PBCH Decoder** performs QPSK demodulation, descrambling, rate recovery, and BCH decoding. It then extracts the MIB output parameters using the **MIB Interpretation** function block. These operations are equivalent to the `ltePBCHDecode` and `lteMIB` functions in the LTE Toolbox.

The `PBCH Controller` stores the equalized data in memory for iterative convolutional decoding attempts. The 4 attempts made at decoding the MIB correspond to the 4 repetitions of the MIB data per PBCH transport block.

BCH Decoder

The **BCH Decoder** quantizes the soft decisions and then decodes the data using the LTE Convolutional Decoder and LTE CRC Decoder blocks. The recommended wordlength of soft decisions at the input to the convolutional decoder is 4 bits. However, the **BCH Decoder** block receives 20-bit soft decisions as input. Therefore the **softBitScalingUnit** block dynamically scales the data so that it utilizes the full dynamic range of the 4 bit soft decisions. The CRC decoder block is configured to return the full checksum mismatch value. The CRC mask, once checked against the allowed values, provides *cellRefP*; the number of cell-specific reference signal antenna ports at the transmitter. If the CRC checksum does not match one of the accepted values then MIB has not been successfully decoded and the PBCH Controller decides whether or not to initiate another decoding attempt.

When a MIB has been successfully decoded, the **MIB Interpretation** subsystem extracts and outputs the fields of the message.

Performance Analysis

Quality of the input waveform is an important factor that impacts the decoding performance. Common factors that affect signal quality are multi-path propagation conditions, channel attenuation and interference from other cells. The quality of the input waveform can be measured using the `cellQualitySearch` function. This function detects LTE cells in the input waveform and returns a structure per LTE cell containing the following fields:

- FrequencyOffset: Frequency offset obtained by `lteFrequencyOffsets` function
- NCellID: Physical layer cell identity
- TimingOffset: Timing offset of the first frame in the input waveform
- RSRQdB: Reference Signal Received Quality (RSRQ) value in dB per TS 36.214 Section 5.1.3 [2]
- ReportedRSRQ: RSRQ measurement report (integer between 0 and 34) per TS 36.133 Section 9.1.7 [3]

Applying the `cellQualitySearch` function to the captured waveform `eNodeBWaveform.mat` used in `ltehdlMIBRecovery_init.m` returns the following report:

```
FrequencyOffset: 536.8614
NCellID: 76
TimingOffset: 12709
RSRQdB: -5.3654
ReportedRSRQ: 29
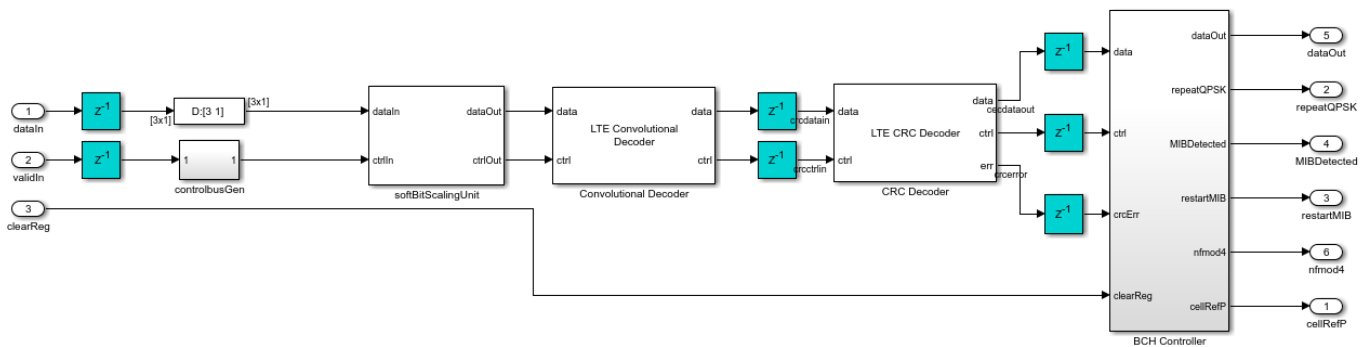
FrequencyOffset: 536.8614
NCellID: 160
TimingOffset: 3108
RSRQdB: -18.1206
ReportedRSRQ: 3
```

There are two cells in the captured waveform, one with cell ID 76 and one with cell ID 160. The cell with NCellID = 76 has a much higher ReportedRSRQ, indicating that it is a stronger signal. In this example the Simulink model decodes the MIB for NCellID = 76.

Results and Display

The scope below shows the key control signals for this example. After a pulse is asserted on the *start* signal the cell search process is started. Successful detection of a cell is indicated by the *cellDetected* signal. When the *cellDetected* signal is asserted the *NCellID* and *TDDMode* signal become active, indicating the cell ID number and whether the cell is using TDD (1) or FDD (0). After the cell has been detected the OFDM demodulator waits until subframe 0 of the next frame to start outputting the grid data, hence there is a gap between *cellDetected* going high, and grid data being output as indicated by the *gridDataValid* signal. When *gridDataValid* is first asserted *subFrameNum* will be zero, and will increment for subsequent subframes. The simulation stops on the *MIBDetected* or *mibError* signals being asserted.

Once MIB has been detected the *NDLRB*, *PHICH*, *Ng*, *nFrame*, and *CellRefP* signals all become active, indicating the key parameters of the cell. These parameters are displayed in the model, as they are static values when the simulation is stopped.

The following MIB information is decoded when decoding the captured waveform:

```
NCellID (Cell ID): 76
TDDMode (0 = FDD, 1 = TDD) : 0
NDLRB (Number of downlink resource blocks): 25
PHICH (PHICH duration) index: 0
Ng (HICH group multiplier): 2
NFrame (Frame number): 262
CellRefP (Cell-specific reference signals): 2
```

This indicates that the duplex mode used by the cell is FDD, the MIB was decoded in frame number 262, the PHICH duration is 'Normal' and the HICH group multiplier value is 'One'.

HDL Code Generation and Verification

To generate the HDL code for this example you must have an HDL Coder™ license. Use the `makehdl` and `makehdltb` commands to generate HDL code and HDL testbench for the **HDL LTE MIB Recovery** subsystem. Because the input waveform in this example contains at least 40 subframes to complete the cell search and MIB recovery, test bench generation takes a long time.

The **HDL LTE MIB Recovery** subsystem was synthesized on a Xilinx® Zynq®-7000 ZC706 evaluation board. The post place and route resource utilization results are shown in the table below. The design met timing with a clock frequency of 140 MHz.

```
      Resource        Usage
    _____   _____

    Slice Registers   33348
    Slice LUTs        25630
    RAMB18               41
    RAMB36               37
    DSP48               115
```

For more information see "Prototype Wireless Communications Algorithms on Hardware" on page 2-21

References

1   3GPP TS 26.211 "Physical Channels and Modulation"

2   3GPP TS 36.214 "Physical layer"

3   3GPP TS 36.133 "Requirements for support of radio resource management"

See Also

Related Examples

- "LTE HDL Cell Search" on page 5-106
- "LTE HDL SIB1 Recovery" on page 5-123
- "LTE HDL PBCH Transmitter" on page 5-152

LTE HDL PBCH Transmitter

This example shows how to implement an LTE transmitter multiple-input multiple-output (MIMO) design, including PSS, SSS, CRS, and MIB, optimized for HDL code generation.

Introduction

The model in this example generates a baseband waveform specified by 3GPP TS 36.211. The waveform includes the primary synchronization signal (PSS), secondary synchronization signal (SSS), cell-specific reference signals (CRS), and the master information block (MIB) for transmission through the physical broadcast channel (PBCH) for multiple antennas. The model supports dynamic change of NCellID and NDLRB. The MIMO transmitter design is optimized for HDL code generation and when implemented on an FPGA, it can be used to transmit MIMO signals in real time over the air. The MIMO design aids the decoding process in the presence of LTE fading channel. This example supports 1, 2, or 4 antennas and uses transmit diversity as specified in the [1].

The architecture presented in this example is extensible and allows for integration of additional physical transmission channels such as physical downlink control channel (PDCCH), physical downlink shared channel (PDSCH), physical control format indicator channel (PCFICH), and physical HARQ indicator channel (PHICH).

Architecture and Configuration

This figure shows the LTE HDL Transmitter architecture with PSS, SSS, CRS, and PBCH transmission chains.

The input sampling rate is assumed to be at 10.24 MHz. PSS, SSS, PBCH, and CRS signals are generated in parallel, based on the input configuration. A single stream of PSS and SSS signals is used for all the antennas. Multiple streams of PBCH data are generated for multiple antennas through the layer mapping and precoding stages. Each antenna is associated with a corresponding

LTE memory bank, which is sized to store one subframe of LTE data samples. These generated data streams are written into LTE memory bank corresponding to indices generated, based on the output *ready* signal of LTE OFDM Modulator. Then, the data is read out of all LTE memory bank in parallel, modulated and transmitted on the antennas simultaneously. The LTE OFDM Modulator block uses a 2048-point FFT to support all NDLRBs.

In this example, the transmitter transmits LTE MIMO signals for the following configurations:

```
        Property                Value
    _____      _____
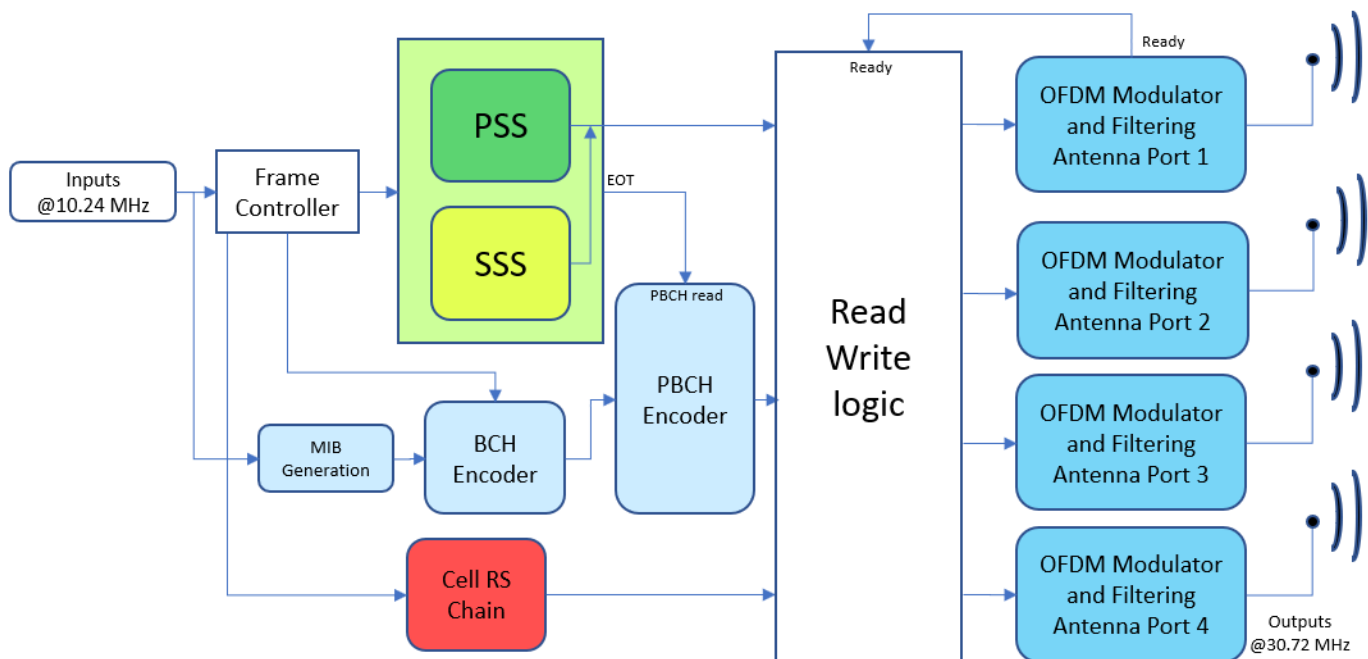
    Duplex mode           FDD
    CellRefP              1/2/4
    Bandwidth             1.4 - 20 MHz
    Cyclic prefix         Normal/Extended
    Initial subframe      0
    Initial frame         0
    Ng                    Sixth/Half/One/Two
    PHICH duration        Normal/Extended
```

Structure of Example Model

The top level structure of the **ltehdlTransmitter** model is shown below. You can generate HDL code for the **HDL LTE MIMO Transmitter** subsystem.

Input *start* is a pulse signal to trigger the transmission. You can configure other parameters, including *NDLRB*, *NCellID*, *Cyclic prefix*, *Ng*, *PHICH duration* and *CellRefP* in the workspace after loading or opening the `ltehdlTransmitter.slx` model. The `ltehdlTransmitter_init.m` script is executed automatically by the model's `InitFcn` callback. This script configures the individual blocks in the **HDL LTE MIMO Transmitter** subsystem. The default transmitter configuration used by the `ltehdlTransmitter_init.m` script is:

```
enb.NDLRB = 6;                     % {6,15,25,50,75,100}
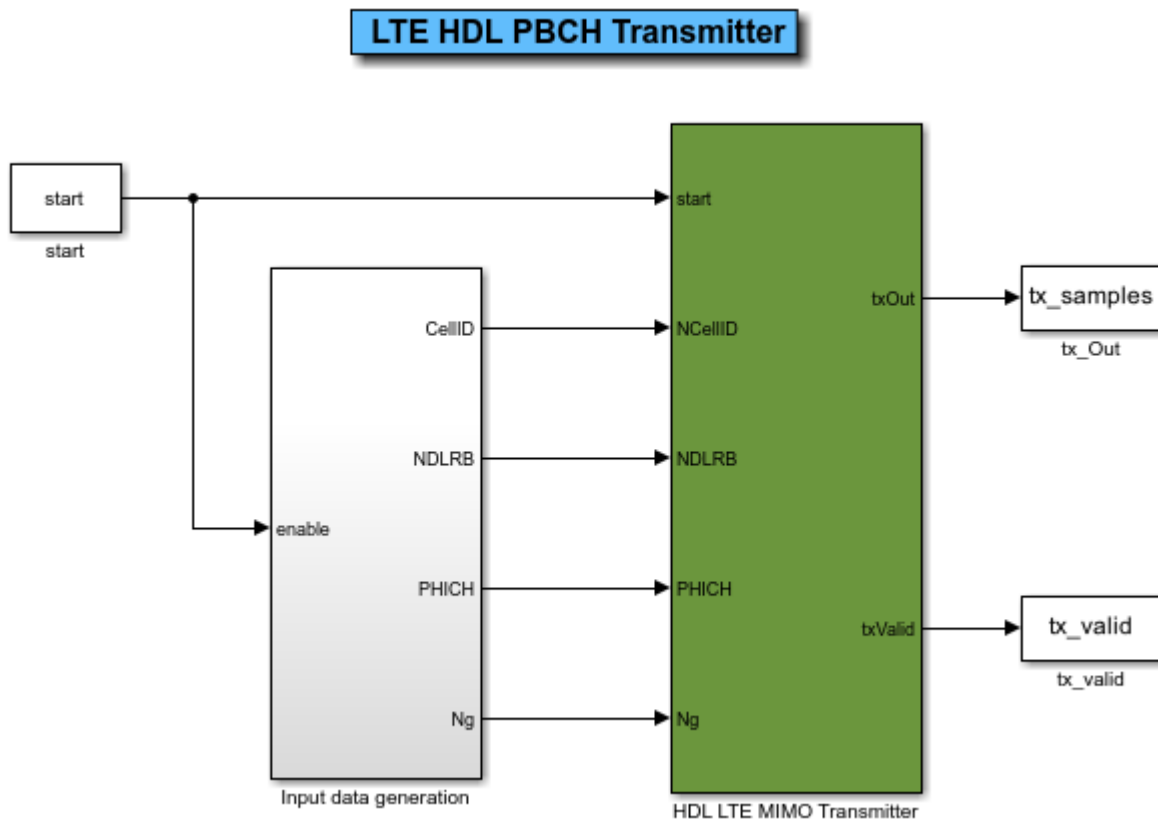
enb.CyclicPrefix = 'Normal';       % {'Normal','Extended'}

enb.Ng = 'Sixth';                  % {'Sixth','Half','One','Two'}

enb.PHICHDuration = 'Normal';      % {'Normal','Extended'}

enb.CellRefP = 4;                  % {1,2,4}

tx_cellids = [390 89 501 231 500]; % {0 to 503}

outRate = 1;                       % {1,2}

TotalSubframes = 45;               % {positive integer}
```

This default configuration can be changed to use other possible values for each variable, as noted in the comment on each line.

HDL LTE MIMO Transmitter

The structure of the **HDL LTE MIMO Transmitter** subsystem is shown below. The **Frame Controller** controls the subframe and frame indices. The **Input Sampler** samples the inputs *NDLRB* and *NCellID* and then propagates the values to the subsequent blocks. The **PSS & SSS generation** generates PSS, SSS, and the corresponding memory address based on NDLRB and subframe index. The **MIB generation** block generates the serial MIB data. The **BCH Encoder** and **PBCH Encoder** generate information for PBCH channel and memory addresses for all the antennas. The **CellRS Chain** generates cell-specific reference signals and corresponding addresses for each antenna. The **Read Write Logic** writes and reads the grid data from each **LTE Memory Bank** and provides the data to the corresponding **LTE OFDM Modulator**. The **Discrete FIR Filter** block filters the modulated data using coefficients that are calculated based on the input configuration.

Frame Controller

This subsystem assumes an input sampling rate of 10.24 MHz. It controls the subframe and radio frame boundaries by providing *cellEnb* signal to sample *NCellID*. It returns radio frame and subframe indices. It also provides *syncStart*, *bchStart*, and *cellRSStart* trigger signals to control the downstream blocks.

Assumes input @ 10.24 MHz. Provides 'syncStart', 'bchStart' and 'cellRSStart' trigger signals for PSS & SSS Chain, BCH Encoder and CellRS Chain subsystems. Also provides cellEnb to Input sampler to sample the input NCellID. Controls subframe and radio frame (nf mod 4) boundaries by providing its indices

PSS & SSS Generation

This subsystem generates the primary synchronization signal (PSS), secondary synchronization signal (SSS), and respective write addresses for LTE Memory Bank based on inputs *NDLRB* and *NCellID*. *syncStart* triggers the generation of PSS and SSS. The PSS and SSS occupy the same central 62 subcarriers of two OFDM symbols in a resource grid [1]. This subsystem generates both the signals and their corresponding addresses at the same time, so that a single stream of both PSS and SSS can be written to all the LTE Memory Banks corresponding to each antenna simultaneously.

The PSS sequence is generated from a frequency-domain Zadoff-Chu sequence [1]. The Zadoff-Chu root sequence index depends on NCellID2, which is derived from *NCellID*. There are three possible NCellID2 values, so all possible PSS sequences are precalculated and stored in `PSS_LUT`.

- **PSS_generation**: Determines NCellID2 and reads the corresponding PSS sequence out of `PSS_LUT` sequentially.
- **PSS_indices**: Computes the memory addresses required to write PSS data into LTE Memory Bank. This subsystem is equivalent to the LTE Toolbox™ function `ltePSSIndices`.

The SSS sequence is an interleaved concatenation of two 31-bit length binary sequences. The concatenated sequence is scrambled with a scrambling sequence given by PSS. The combination of these sequences differs between subframe 0 and subframe 5 [1]. The indices m0 and m1 are derived from the physical-layer cell identity group, NCellID1 [1]. These indices and the sequences s(n), c(n), and z(n) are calculated and stored in `m0_LUT`, `m1_LUT`, `S_LUT`, `C_LUT`, and `Z_LUT` respectively.

- **SSS_generation**: Computes m0 and m1 based on the *NCellID* and calculates indices required for sequences s(n), c(n), and z(n) based on the subframe index. Generates SSS sequence as specified in [1].

- **SSS_indices**: Computes memory addresses required to write SSS data into LTE Memory Bank. This subsystem is equivalent to the LTE Toolbox™ function `lteSSSIndices`.

'pbchread' enable reading of PBCH data during nfmod4 = 0 (radio frame index mod 4)

BCH Encoder

Broadcast Channel (BCH) processes the MIB information arriving to the block in the form of a maximum of one transport block for every transmission time interval (TTI) of 40 ms. The block implements the following coding steps.

- **CRC Encoding**: The entire transport block is used to calculate the CRC parity bits for a polynomial specified in [2]. The parity bits are then appended to the transport block. After appending, CRC bits are scrambled according to the transmit configuration. The LTE CRC Encoder block uses the CRC mask set by the `ltehdlTransmitter_init.m` script based on the input configuration.

- **Channel Coding**: The LTE Convolutional Encoder block encodes the information bits using tail-biting convolutional code with constraint length 7, and polynomial $G_0 = 133, G_1 = 171, G_2 = 165$ in octal. Because the coding rate of the encoder is 1/3, the coded bits are then serialized using a Serializer1D (HDL Coder) block and control signals are resampled to 30.72 MHz (3 * 10.24 MHz).

- **Rate Matching**: The coded bits are interleaved, followed by selection of bits for a particular length using an interleaved address [2]. For broadcast channel, because the length of the MIB is constant, interleaved write and read addresses are precalculated and stored in `wr_addrLUT` and `rd_addrLUT` respectively. Once all serialized coded bits have been written into interleaved addresses of RAM, the bits are read back using interleaved read addresses.

PBCH Encoder

The physical broadcast channel processes the coded bits in the following steps.

- **Scrambling**: Coded bits from **BCH Encoder** are scrambled with a cell-specific sequence using a LTE Gold Sequence Generator block. The sequence is initialized with NCellID in each radio frame(n_f) fulfilling $n_f mod 4 = 0$. The generated cell-specific sequence is scrambled with the input coded bits.

- **QPSK Mapping**: The modulation scheme specified for PBCH channel is QPSK [1]. The LTE Symbol Modulator block generates complex-valued QPSK modulation symbols.

- **Layer Mapping**: Three subsystems are defined for the layer mapping. These subsystems are placed inside a variant subsystem. Based on the number of antennas used in the input configuration `enb.CellRefP`, the `ltehdlTransmitter_init.m` script selects one of the three subsystems in the variant subsystem. This **Layer Mapping** block separates the input streaming samples into 1, 2, or 4 sequences based on the number of antennas used. The input is streamed out without any processing for a single antenna. For multiple antennas, this block generates a valid signal for each antenna. Only one of the valid signals will be high for each input sample.

- **Precoding**: This block also uses variant subsystem to process input samples differently based on the number of antennas in the transmitter configuration. For `enb.CellRefP` set to 1 the input is streamed out without any processing. For `enb.CellRefP` set to 4 (or 2), every four (or two) consecutive samples X0, X1, X2, X3 (or X0, X1) are processed to generate four (or two) streams of 4 (or 2) samples each in four (or two) time instants.

The subsystem shown generates the output sequence for 4 antennas as specified in [1].

- **Memory**: Complex modulated symbols corresponding to the physical broadcast channel for the initial radio frame are stored in PBCH_RAM. For four consecutive radio frames, the number of bits to be transmitted on the physical broadcast channel is 1920 for normal cyclic prefix and 1728 for extended cyclic prefix. The Read Write Controller controls read and write addresses based on $n_f mod 4$, since the periodicity of the broadcast channel (BCH) is 40 ms.

- **PBCH Indexing**: Computes the memory addresses required to write PBCH data into LTE Memory Bank. The **PBCH_indices** subsystem is equivalent to the LTE Toolbox™ function `ltePBCHIndices`.

CellRS Chain

The cell-specific reference sequence is complex modulated values of a pseudo-random sequence as defined in [1]. The pseudo-random sequence generator is initialized with c_{init} at the start of each OFDM symbol, as specified in [1].

- **CellRS_generation**: Input *cellRSStart* triggers the generation of CRS signals. Since the CRS is available in six OFDM symbols (four OFDM symbols in antenna port 0 and port 1, and two OFDM symbols in antenna port 2 and port 3) of a single subframe, this subsystem calculates a 6-element c_{init} vector for every subframe. The LTE Gold Sequence Generator block is initialized with vector c_{init} to represent multiple channels and provides six different cell-specific pseudo-random sequences. The Write Controller controls writing of these sequences into six memory banks in `CellRS_RAM`. It also returns *rd_en*, which enables reading data out of `CellRS_RAM`. The Read Controller controls reading of CRS data. It reads six OFDM symbols if four antennas are used, and reads only 4 OFDM symbols if one or two antennas are used. It returns *rd_bank* and *rd_valid* signals to select an appropriate symbol for the six/four OFDM symbols. The sequence is then mapped to complex QPSK modulated symbols.

- **CellRS_indices**: This subsystem computes the addresses for each **LTE Memory Bank** required to write CRS data. It is equivalent to the LTE Toolbox™ function `lteCellRSIndices`.

Read Write Logic

The **Read Write Logic** subsystem contains a Write Selector, Read Selector, four LTE Memory Banks with a Grid Bank Select associated with each of the LTE Memory Bank. The LTE Memory Bank storage capacity is one subframe of complex modulated symbols at the largest supported LTE bandwidth (20 MHz). Each LTE Memory Bank can store 14 x 2048 x 16-bit complex values, that is, 14 OFDM symbols, each containing 2048 complex values.

The Write Selector writes subframes of data into the memory banks. The PSS and SSS occupy central subcarriers. A single stream of PSS and SSS data is used for all the antennas. The PBCH data consists of multiple streams corresponding to each antenna port. The CRS data generated is mapped to the grid based on the four addresses generated for each **LTE Memory bank** in **CellRS_indices** block. The Write Selector first writes PSS and SSS simultaneously into corresponding locations in all LTE Memory Banks. Then, it writes PBCH data and CRS data into the corresponding LTE Memory Banks and returns *rd_enb* to indicate that the write is complete.

The Read Selector reads the samples from each **LTE Memory Bank** based on *rd_enb* and *ready* from the LTE OFDM Modulator block. Each LTE Memory Bank returns a 14 element vector corresponding to a single subcarrier. The **Grid Bank Select** selects the appropriate sample from the 14 element vector to form the resource grid output for each antenna.

Since the scope of this example is limited to PSS, SSS, CRS, and PBCH transmission, all the LTE Memory Banks are erased at the start of every subframe, before writing new data into the memory.

OFDM Modulation and Filtering

Grid data from LTE Memory Bank is OFDM-modulated using the LTE OFDM Modulator block with 'Output data sample rate' parameter set to 'Match output data sample rate to NDLRB'. The modulated data is filtered using a Discrete FIR Filter block with coefficients generated at a sampling rate corresponding to the NDLRB. Variant subsystems control the number of OFDM modulators and FIR filters used based on the number of antennas, which reduces the resource utilization when a single antenna is used.

Verification and Results

After running the simulation, the `ltehdlTransmitter_PostSim.m` script is executed automatically by the `StopFcn` callback of the model. In this example, the transmitter output is verified by the following methods:

Verification of model's transmitted signal:

The transmitter output signal in this model is cross-verified with a reference transmitter signal that is generated using LTE Toolbox™ functions by the following two subplots for each antenna.

1 The first subplot shows the Power Spectral Density (PSD) output of the filtered data. The result is compared with the PSD of the reference output signal generated using LTE Toolbox™. This comparison shows the equivalence of the two signals. The figure shows a transmission bandwidth of BW = 1.4MHz.

2 The second subplot shows the absolute-value of the transmitted waveform. The result is plotted on top of the absolute-value of the reference transmitter signal generated using LTE Toolbox™. The plot also shows the difference between the samples obtained through HDL implementation and the reference signal. This comparison shows the minimal error between the two transmitter signals.

Cell Search & MIB Decoding Results:

The valid samples of the transmitter output signal are stored to the workspace variable `txSamples`. These samples are passed through an LTE fading channel to create the receiver input signal, `rxSamples`. The `lteFadingChannel` (LTE Toolbox) function models the LTE fading channel.

This example uses the following channel configuration:

```
chcfg.NRxAnts = 1;
chcfg.MIMOCorrelation = 'Medium';
chcfg.NormalizeTxAnts = 'On';
chcfg.DelayProfile = 'EPA';                 % {'off','EPA'}
% The below model configuration exist only if Delay profile is not set
% to 'off'.
chcfg.DopplerFreq = 5;
chcfg.SamplingRate = 30.72e6;
chcfg.InitTime = 0;
chcfg.NTerms = 16;
chcfg.ModelType = 'GMEDS';
chcfg.NormalizePathGains = 'On';
chcfg.InitPhase = 'Random';
chcfg.Seed = 1;
```

To create a fading-free channel, set the `chcfg.DelayProfile` to 'off' in the `ltehdlTransmitter_PostSim.m` script.

This channel configuration works with the default `enb` structure, and supports changes only in the `enb.PHICHDuration` and `enb.Ng` fields.

The following figures show the results of the cell search and MIB decoding of the channel output, `rxSamples`, using LTE toolbox™ functions. These figures verify the transmitter performance and compare the HDL transmitter implementation against the input configuration defined in `tx_cellids` and `enb`.

- NCellID after Cell Search: Displays the LTE cell search results performed on the fading channel output.

- Cell-wide settings after MIB decoding: Displays the fields of MIB after MIB decoding - NDLRB, Ng, PHICH duration, and System Frame Number (SFN) performed on the fading channel output.

The example model does not support simulation in rapid accelerator mode.

Validation with Cell Search and MIB Recovery Example

You can verify the **LTE HDL PBCH Transmitter** example by connecting it to the "LTE HDL MIB Recovery" on page 5-141 example model and checking that the output of the transmitter is decoded correctly. To make the transmitter model compatible with the receiver model, make these changes to the transmitter:

- Set the `outRate` = 2 (default value 1) before running the model. This will set the output rate of each **LTE OFDM Modulator** and generate the fir filter coefficients associated with each antennas.
- Set the `enb.CellRefP` = 2 (default value 4) before running the model.
- Use the same `NCellID` for all radio frames in the transmission. i.e. set `tx_cellids` to a scalar value in the range 0-503.

The figure shows the **HDL LTE MIMO Transmitter** and **HDL LTE MIB Recovery** subsystems connected together. It also shows the result of simulating the model. The display blocks show the CellID and MIB fields (NDLRB, Ng, PHICH duration and System Frame Number (SFN)) that the receiver decoded from the output of the **HDL LTE MIMO Transmitter** subsystem.

You can also verify the design without using a channel by terminating the output from the second antenna and bypassing the channel system with the output from the first antenna.

HDL Code Generation

To check and generate HDL for this example, you must have an HDL Coder™ license. Use the `makehdl` and `makehdltb` commands to generate the HDL code and test bench for the **HDL LTE MIMO Transmitter** subsystem. Because the stopTime in this example depends on `TotalSubframes`, the test bench generation time depends on the `TotalSubframes`.

The **HDL LTE MIMO Transmitter** subsystem is synthesized on a Xilinx® Zynq®-7000 ZC706 evaluation board. The post place and route resource utilization results are shown in the table below.

Resources	No. of antennas used = 1	No. of antennas used = 2	No. of anten
Slice Registers	12788	23839	4578
Slice LUT	11984	22220	4280
RAMB36	41	82	164
RAMB18	11	21	41
DSP	49	93	177
Max. Frequency (MHz)	210.08	206.39	204

References

1 3GPP TS 36.211 "Physical channels and modulation".

2 3GPP TS 36.212 "Multiplexing and channel coding".

See Also

Related Examples

- "LTE HDL Cell Search" on page 5-106
- "LTE HDL MIB Recovery" on page 5-141
- "LTE HDL SIB1 Recovery" on page 5-123

Deploy LTE HDL Reference Applications on FPGAs and SoCs

This section contains the list of examples that show how to deploy LTE Wireless HDL Toolbox™ reference applications on FPGAs and SoCs.

- "LTE MIB Recovery and Cell Scanner Using Analog Devices AD9361/AD9364" (Communications Toolbox Support Package for Xilinx Zynq-Based Radio): Implement an LTE master information block (MIB) recovery system partitioned across the processing system (PS) and the programmable logic (PL).
- "LTE SIB1 Recovery Using Analog Devices AD9361/AD9364" (Communications Toolbox Support Package for Xilinx Zynq-Based Radio): Recover the first system information block (SIB1) from an LTE downlink signal.

These examples reuse the LTE Simulink® models to generate HDL for the FPGA logic. They use hardware-software co-design modeling techniques and hardware support packages to add all the software modeling and interfacing required to implement the algorithm in real-time on hardware.

See Also

Related Examples

- "LTE HDL Cell Search" on page 5-106
- "LTE HDL MIB Recovery" on page 5-141
- "LTE HDL SIB1 Recovery" on page 5-123

HDL OFDM MATLAB References

This example shows how to model OFDM transmitter, additive white Gaussian noise (AWGN), and OFDM receiver hardware algorithms in MATLAB® as steps toward developing a Simulink® implementation for hardware. The HDL OFDM MATLAB References example bridges the gap between a mathematical algorithm and its hardware implementation. This example provides MATLAB references of the HDL OFDM Transmitter, HDL AWGN, and HDL OFDM Receiver algorithms. You can use these MATLAB references to generate test vectors for verifying the HDL implementation of the "HDL OFDM Transmitter" on page 5-182, "HDL Implementation of AWGN Generator" on page 4-44, and "HDL OFDM Receiver" on page 5-198 Simulink models.

HDL OFDM Transmitter MATLAB Reference

This section describes the MATLAB reference of HDL OFDM Transmitter.

This MATLAB reference accepts a modulation order, code rate index, number of frames, and data bits to be transmitted as a `txParam` structure or array of structures. `txParam` has these fields.

- `modOrder` — Specify 2, 4, 16, or 64 for 'BPSK', 'QPSK', '16QAM', and '64QAM', respectively. The default value is 4 ('QPSK').
- `codeRateIndex` — Specify 0, 1, 2, or 3 for the rates '1/2', '2/3', '3/4', and '5/6' respectively. The default value is 0 ('1/2').
- `numFrames` — Specify a positive integer. The default value is 5.
- `txDataBits` — Specify binary values in a row or column vector of length `trBlkSize` x `txParam.numFrames`. The default value is a column vector containing randomly generated binary values of length `trBlkSize` x `txParam.numFrames`.

Calculate the transport block size (`trBlkSize`) by using these parameters.

- numSubCar — Number of subcarriers per symbol
- pilotsPerSym — Number of pilots per symbol
- numDataOFDMSymbols — Number of data OFDM symbols
- bitsPerModSym — Number of bits per modulated symbol
- codeRate — Punctured code rate
- dataConvK — Constraint length of the convolutional encoder
- dataCRCLen — CRC length

trBlkSize = ((numSubCar - pilotsPerSym) x numDataOFDMSymbols x bitsPerModSym x codeRate) - (data(

For example, to generate a time-domain OFDM transmitter waveform of 5 frames with a modulation scheme of 16QAM and code rate of 1/2 using random data bits in the transport block, format the inputs as structure.

```
txParam.modOrder = 16; % Modulation order corresponding to 16-QAM
txParam.codeRateIndex = 0; % Code rate index corresponding to 1/2
txParam.numFrames = 5; % Number of frames to be generated

% Calculate transport block size (trBlkSize) using parameters
numSubCar = 72; % Number of subcarriers per symbol
pilotsPerSym = 12; % Number of pilots per symbol
numDataOFDMSymbols = 32; % Number of data OFDM symbols
bitsPerModSym = log2(txParam.modOrder); % Bits per modulated symbol
codeRate = 1/2; % Punctured code rate
dataConvK = 7; % Constraint length of convolutional code polynomial
dataCRCLen = 32; % Data CRC length
trBlkSize = ((numSubCar-pilotsPerSym)*numDataOFDMSymbols* ...
    bitsPerModSym*codeRate) - (dataConvK-1) - dataCRCLen;
txParam.txDataBits = randi([0 1],txParam.numFrames*trBlkSize,1);

% Generate complex baseband transmitter waveform
fprintf('\n------------------------\n');
fprintf('\n Transmitting %d frames ...\n',txParam.numFrames);
[txWaveform,txGrid,txDiagnostics] = whdlexamples.OFDMTx(txParam);
fprintf('\n Transmission successful.\n');
fprintf('\n------------------------\n');
```

```
------------------------

 Transmitting 5 frames ...

 Transmission successful.

------------------------
```

The whdlexamples.OFDMTx function returns arguments txWaveform, txGrid, and txDiagnostics.

- txWaveform is the generated time-domain waveform and is returned as a column vector of length ((fftLen + cpLen) x txParam.numFrames x numSymPerFrame), where:

1 fftLen is the FFT length.
2 cpLen is the cyclic prefix length.

3 `txParam.numFrames` is the number of OFDM frames generated.

4 `numSymPerFrame` is the number of OFDM symbols per frame.

If `txParam` is an array of structures, then in the expression `txParam.numFrames` is replaced with the sum of all `numFrames` attributes present in the array. The frame structure of the generated time-domain waveform `txWaveform` is similar to the Simulink HDL OFDM Transmitter output waveform. For the detailed explanation of the frame structure, see the "HDL OFDM Transmitter" on page 5-182 example.

- `txGrid` is the transmitter grid output and is returned as a matrix of size `numSubCar`-by-(`txParam.numFrames` x `numSymPerFrame`), where `numSubCar` is the number of active subcarriers.

- `txDiagnostics` is a structure or array of structures and consists of these three fields:

1 `headerBits` represents the header bits as a column vector of size 14, which includes 3 bits for the FFT length index, 2 bits for the symbol modulation type, 2 bits for the code rate index, and 7 spare bits.

2 `dataBits` represents actual data bits transmitted in the given number of frames (`txParam.numFrames`). This field is a binary-valued row or column vector of length equal to (`txParam.numFrames` x `trbBlkSize`). Whether `dataBits` is a row or column vector depends on the dimension of `txParam.dataBits`. The default size is a column vector of length equal to `txParam.numFrames` x `trbBlkSize`.

3 `ofdmModOut` represents the OFDM modulator output as a column vector of length equal to (`fftLen` + `cpLen`) x `txParam.numFrames` x `numSymPerFrame`.

OFDMTx

`whdlexamples.OFDMTx` function is used to generate OFDM transmitter waveform with synchronization, reference, header, pilots, and data signals. This function returns `txWaveform`, `txGrid`, and `txDiagnostics` using transmitter parameters `txParam`. This function internally calls these individual functions.

- `generateOFDMSyncSignal` — This function generates the synchronization signal `SyncSignal`. This function uses Zadoff-Chu sequence with a root index of 25 and length of 62.

- `generateOFDMRefSignal` — This function generates the reference signal `refSignal` for the given FFT length `fftLen`. This function uses a BPSK-modulated pseudo random binary sequence.

- `generateOFDMPilotSignal` — This function generates the pilot signal `pilot`. This function uses a BPSK-modulated pseudo random binary sequence.

- `OFDMSymbolModulate` — This function modulates input bits to complex modulation symbols based on the specified modulation scheme BPSK, QPSK, 16QAM, and 64QAM.

Plot the resource grid of the transmitter waveform. The plot indicates the magnitude variations of each resource grid element.

```
plotResourceGrid(txGrid);
```

OFDM Resource Grid

HDL AWGN MATLAB Reference

This section describes the MATLAB reference of HDL AWGN.

This MATLAB reference is used for performance evaluation of the HDL OFDM Transmitter and Receiver algorithms. The HDL AWGN MATLAB reference generates AWGN by accepting the signal-to-noise ratio (SNR) in decibel (dB) and sets of seeds. For more details, see "HDL Implementation of AWGN Generator" on page 4-44. The generated AWGN is added to the HDL OFDM Transmitter output.

```
FFTLen = 128;
CPLen = 32;
usedSubCarr = 72; % Out of 128 subcarriers, 72 subcarriers are loaded with data
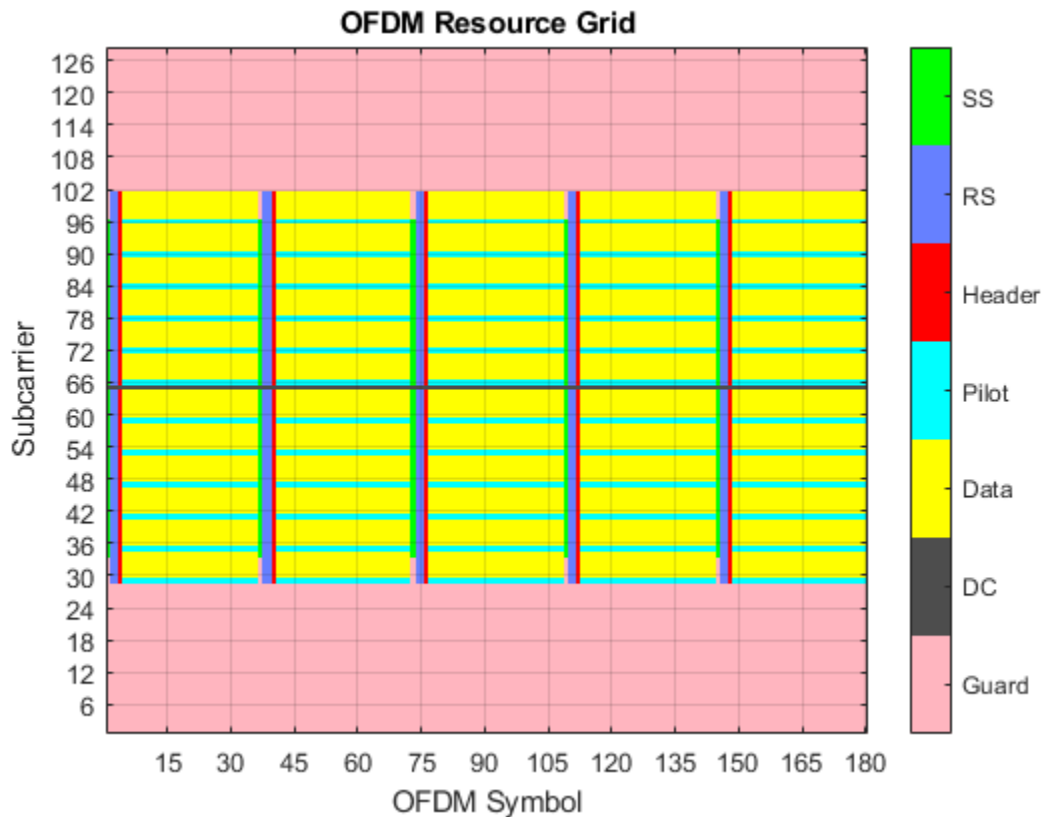
SNRdB = 30;
SNRdBSimInput = SNRdB*ones(length(txWaveform)+633,1);
seedsURNG1 = [121 719 511]; % Seeds for TausURNG1
seedsURNG2 = [2343 323 833]; % Seeds for TausURNG2
txScaleFactor = FFTLen/sqrt(usedSubCarr);

awgnNoise = whdlexamples.hdlawgn(SNRdBSimInput,seedsURNG1,seedsURNG2);

rxWaveform = txWaveform + (1/txScaleFactor)*awgnNoise(634:end);
fprintf('\n Applying the AWGN channel at %d dB...\n', SNRdB);
```

```
 Applying the AWGN channel at 30 dB...
```

HDL OFDM Receiver MATLAB Reference

This section describes MATLAB reference of HDL OFDM Receiver.

This MATLAB reference includes time synchronization, CFO estimation and correction, OFDM demodulation, header recovery, CPE estimation and correction, and data recovery.

The `whdlexamples.OFDMRx` function accepts `rxWaveform`, a transmitted waveform passed through an AWGN channel.

The `whdlexamples.OFDMRx` function returns decoded bits `rxBits` and an array of structures, `rxDiagnostics`, consisting of these eight fields.

- `estCFO` — Estimated carrier frequency offset
- `rxConstellationHeader` — Demodulated header constellation symbols
- `rxConstellationData` — Demodulated data constellation symbols
- `softLLR` — Demodulated soft LLR bits
- `decodedCodeRateIndex` — Decoded code rate index from header
- `decodedModOrder` — Decoded modulation order from header
- `headerCRCErrorFlag` — Status of header CRC
- `dataCRCErrorFlag` — Status of data CRC

OFDMRx

The `whdlexamples.OFDMRx` function is used to demodulate and decode the received `rxWaveform`. This function internally calls these individual functions.

- `OFDMFrequencyOffset` — This function estimates the carrier frequency offset based on cyclic prefix (CP) technique. The cyclic prefix portion of the received time-domain waveform is correlated to estimate frequency offset.

- `OFDMFrequencyCorrect` — This function corrects the carrier frequency offset on the received waveform using the estimated frequency offset.

- `OFDMFrameSync` — This function synchronizes the received waveform by performing correlation using the reference signal. This step reduces the intersymbol interference while demodulating the received waveform.

- `OFDMDemodulation` — This function converts the time-domain waveform to frequency-domain waveform for further decoding. The object `dsp.HDLFFT` is used for HDL implementation of the receiver.

- `OFDMChannelEstimation` — This function performs the estimation of the channel using two reference signals. It uses least squares (LS) estimation technique. LS estimates are averaged to improve channel estimation accuracy.

- `OFDMChannelEqualization` — This function performs zero forcing (ZF) equalization using the estimated channel. Then the received waveform that is free of the channel is used for header recovery and data recovery.

- `OFDMHeaderRecovery` — This function recovers header information by performing symbol demodulation, deinterleaving, and Viterbi decoding. The CRC status indicates the success or failure of header information recovery. This header recovery CRC status is given as an output of

the receiver to indicate frame loss or recovery. When the CRC check fails, the header CRC status is `1`. Otherwise, it is `0`.

- `OFDMDataRecovery` — This function performs symbol demodulation, deinterleaving, depuncturing, Viterbi decoding, and descrambling. The function processes the data only when the header CRC check passes. After descrambling the decoded data, CRC check is performed on the recovered data bits to indicate if the packet is errored. When the CRC check fails, the header CRC status is `1`. Otherwise, it is `0`.

```
fprintf('\n Receiving process started...\n');
[rxDataBits,rxDiagnostics] = whdlexamples.OFDMRx(rxWaveform);
fprintf('\n Reception completed\n\n');

% Plot constellation of header and data
scatterplot(rxDiagnostics.rxConstellationHeader(:),1,0,'b.')
title('Header Constellation')
axisObj = gca;
axisObj.XColor = 'w';
axisObj.YColor = 'w';

scatterplot(rxDiagnostics.rxConstellationData(:),1,0,'b.')
title('Data Constellation');
axisObj = gca;
axisObj.XColor = 'w';
axisObj.YColor = 'w';
```

```
 Receiving process started...

 Estimating carrier frequency offset ...

 First four frames are used for carrier frequency offset estimation.

 Estimated carrier frequency offset is -1.913549e-01 Hz.

 Detected and processing frame 5
-----------------------------------------

 Header CRC passed

 Modulation: 16QAM, codeRate=1/2 and FFT Length=128

 Data CRC passed

 Data decoding completed
-----------------------------------------
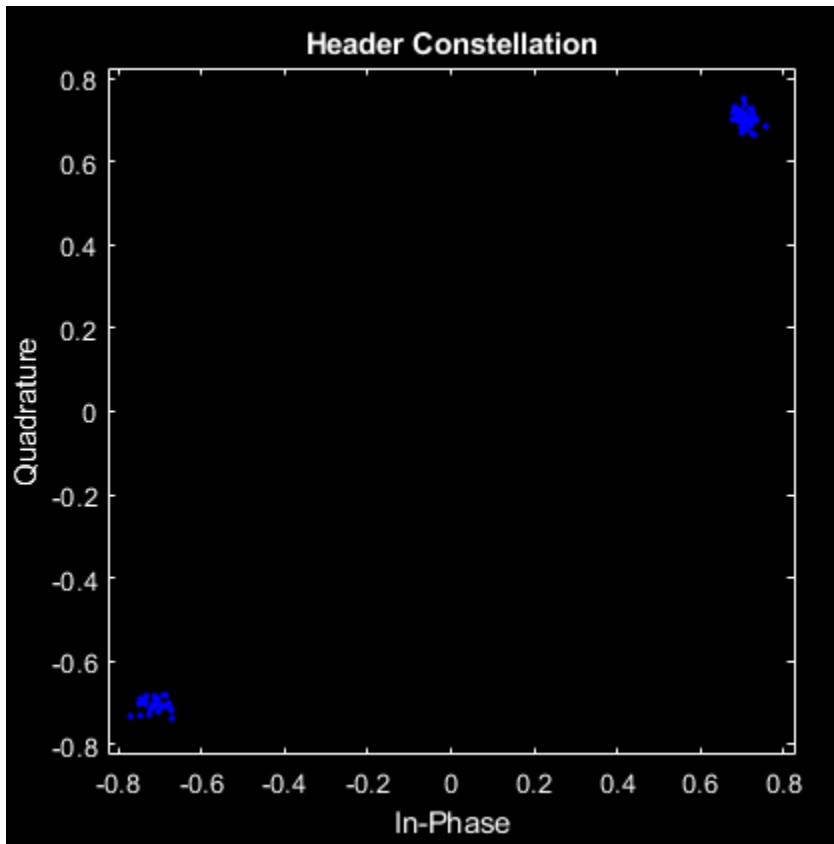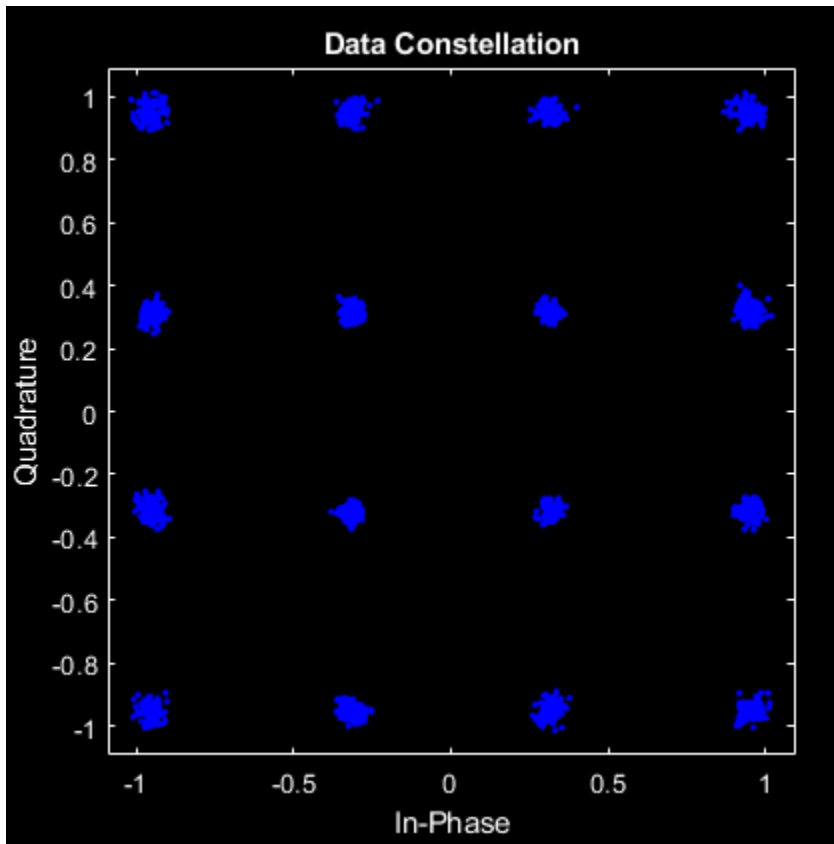
 Reception completed
```

Header Constellation

Verify Simulink Model with MATLAB Reference

In this section, the Simulink HDL OFDM Transmitter, AWGN generator, and Simulink HDL OFDM Receiver implemented in fixed point are compared with the equivalent MATLAB HDL reference models implemented in floating point.

The Simulink model consists of an OFDM Transmitter that generates a time-domain waveform for a user-defined modulation order and code rate. The time-domain waveform is then passed through the AWGN channel that introduces AWGN noise of the desired SNR in dB. Then, the OFDM Receiver is used to demodulate and decode information bits. The outputs of the Simulink model are verified with the MATLAB reference at each stage.

```
open HDLOFDMTxRx;
sim HDLOFDMTxRx;

### Starting serial model reference simulation build
### Model reference simulation target for whdlOFDMRx is up to date.
### Model reference simulation target for whdlOFDMTx is up to date.

Build Summary

0 of 2 models built (2 models already up to date)
Build duration: 0h 0m 1.576s
```

Verify Simulink HDL OFDM Transmitter with MATLAB HDL OFDM Transmitter

In this section, plot the real and imaginary parts of the HDL OFDM Transmitter MATLAB reference function output `txWaveform` and compare with the output of the "HDL OFDM Transmitter" on page 5-182 block.

```
matlabTxWaveform = txWaveform;
simulinkTxWaveform = simTxOut;

figure;
plot(real(matlabTxWaveform),'-bo')
hold on
plot(real(simulinkTxWaveform(1:length(matlabTxWaveform))),'-r.')
legend('MATLAB Tx waveform','Simulink Tx waveform');
title('Comparison of MATLAB Tx and Simulink Tx (Real Part)');
ylim([-0.2 0.2]);
xlabel('Time-Domain Samples');
ylabel('Amplitude');

figure;
plot(imag(matlabTxWaveform),'-bo')
hold on
plot(imag(simulinkTxWaveform(1:length(matlabTxWaveform))),'-r.')
legend('MATLAB Tx waveform','Simulink Tx waveform');
title('Comparison of MATLAB Tx and Simulink Tx (Imaginary Part)');
ylim([-0.2 0.2]);
xlabel('Time-Domain Samples');
ylabel('Amplitude');
```

Comparison of MATLAB Tx and Simulink Tx (Real Part)

Verify Simulink HDL AWGN Generator with MATLAB HDL AWGN

In this section, plot the real and imaginary parts of the MATLAB HDL AWGN is compared with the output of the Simulink AWGN Generator block.

```
matlabChannelOut= rxWaveform;
simulinkChannelOut = simChannelOut;
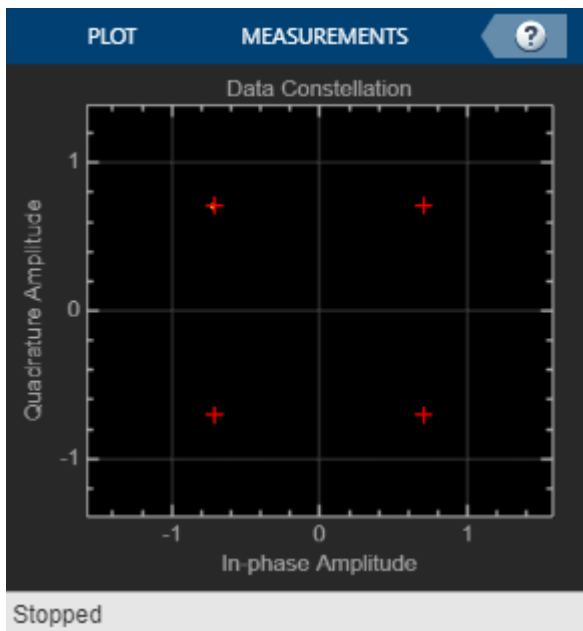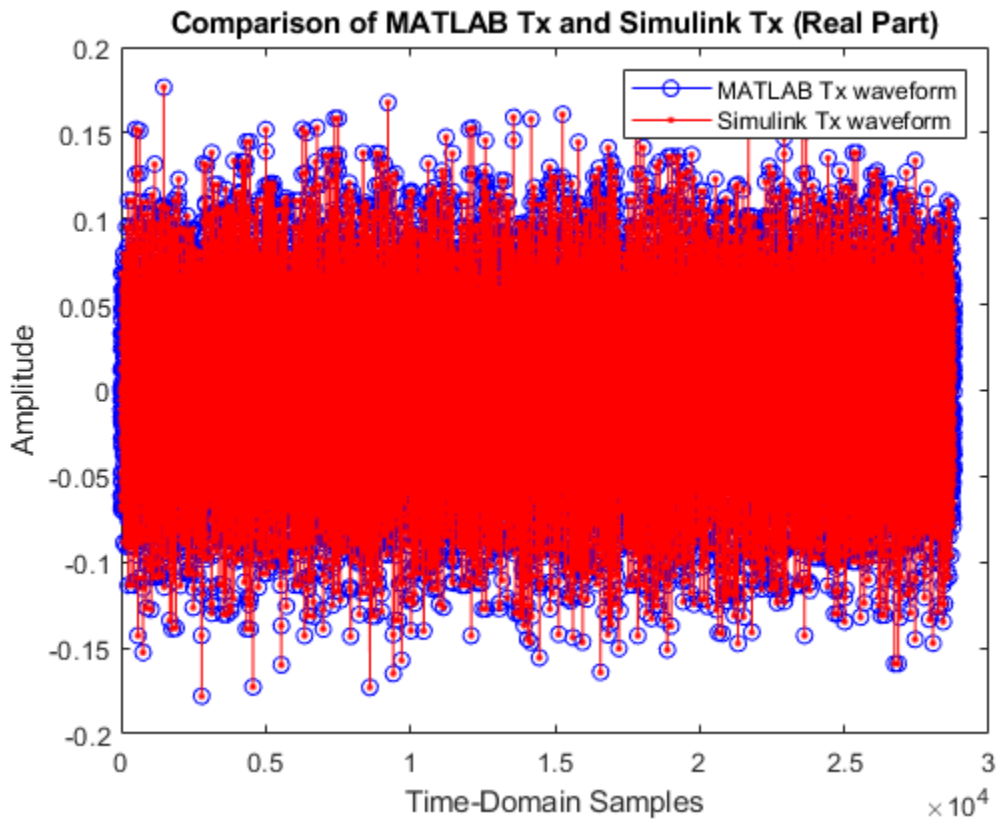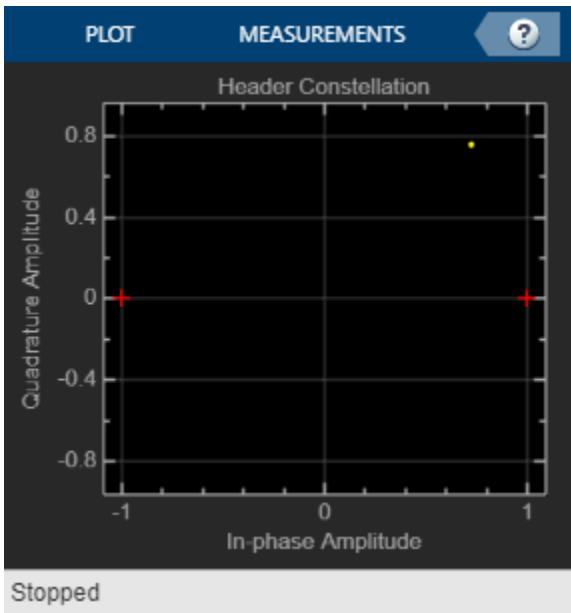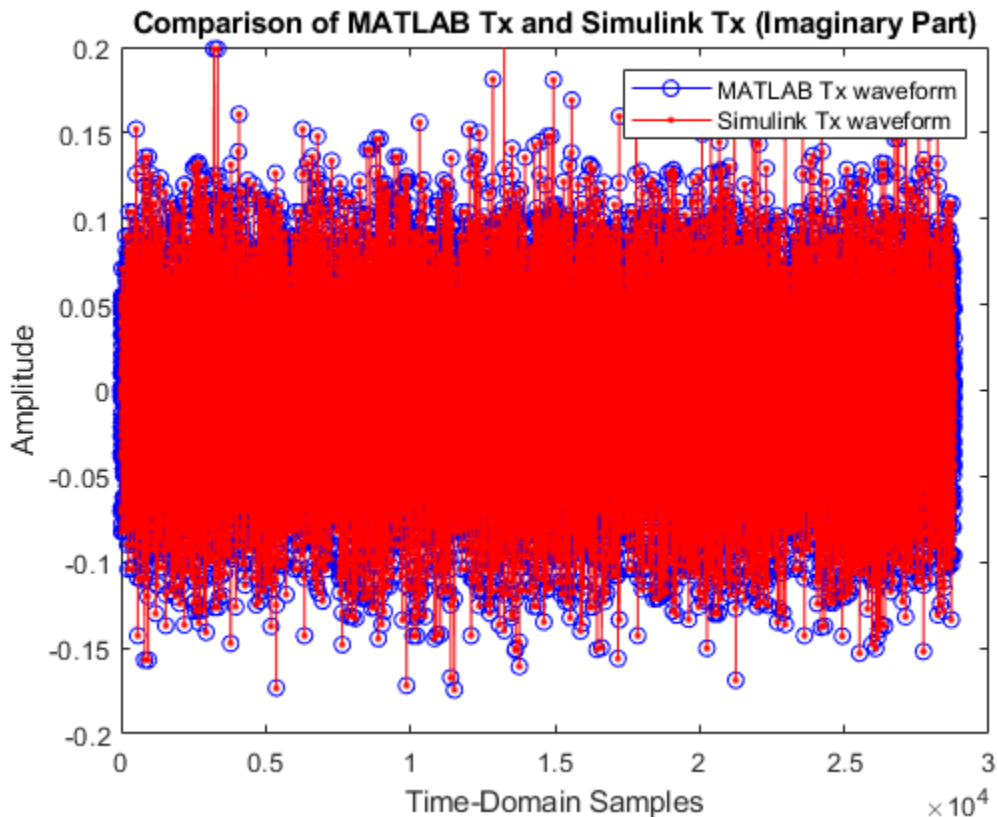
figure;
plot(real(matlabChannelOut),'-bo');
hold on;
plot(real(simulinkChannelOut(1:length(matlabChannelOut))),'-r.');
legend('MATLAB channel output','Simulink channel output');
title('Comparison of MATLAB Channel and Simulink Channel (Real Part)');
ylim([-0.2 0.2]);
xlabel('Time-Domain Samples');
ylabel('Amplitude');


figure;
plot(imag(matlabChannelOut),'-bo');
hold on;
plot(imag(simulinkChannelOut(1:length(matlabChannelOut))),'-r.');
legend('MATLAB channel output','Simulink channel output');
title('Comparison of MATLAB Channel and Simulink Channel (Imaginary Part)');
ylim([-0.2 0.2]);
```

```
xlabel('Time-Domain Samples');
ylabel('Amplitude');
```


Comparison of MATLAB Channel and Simulink Channel (Real Part)

Comparison of MATLAB Channel and Simulink Channel (Imaginary Part)

Verify Simulink HDL OFDM Receiver with MATLAB HDL OFDM Receiver

In this section, plot the decoded bits of the MATLAB receiver as compared with the decoded bits of the Simulink receiver.

```
matlabRxOut= rxDataBits;
simulinkRxOut = simRxDataBits;
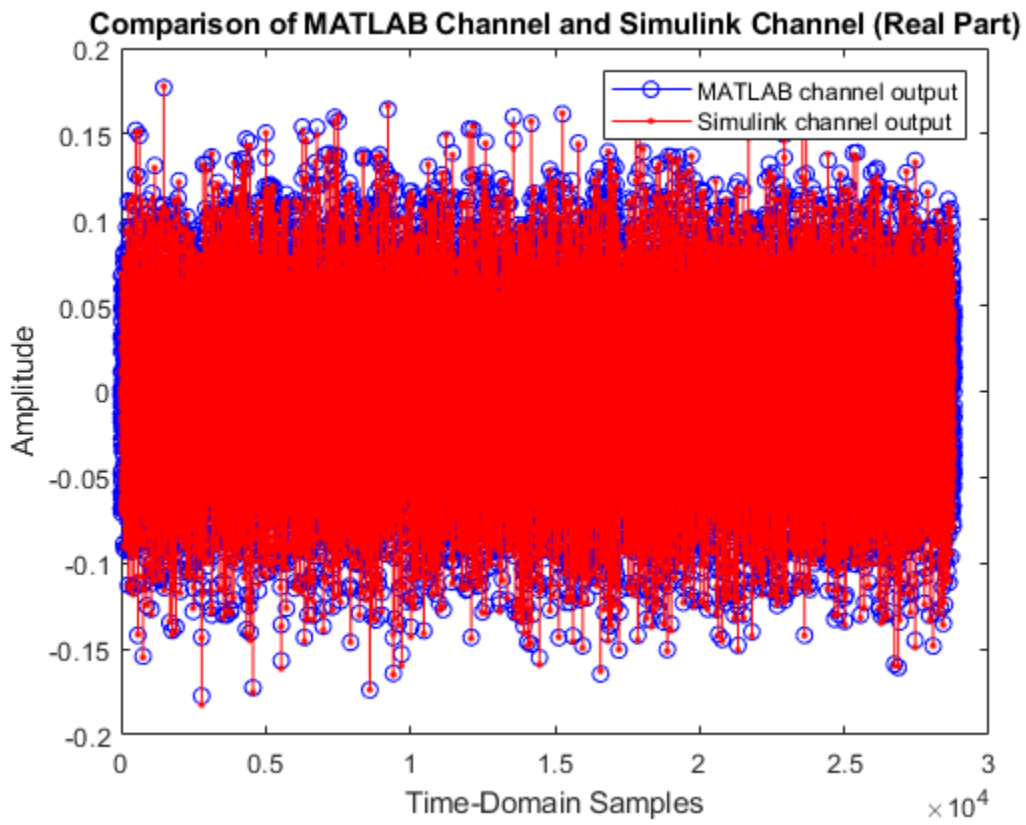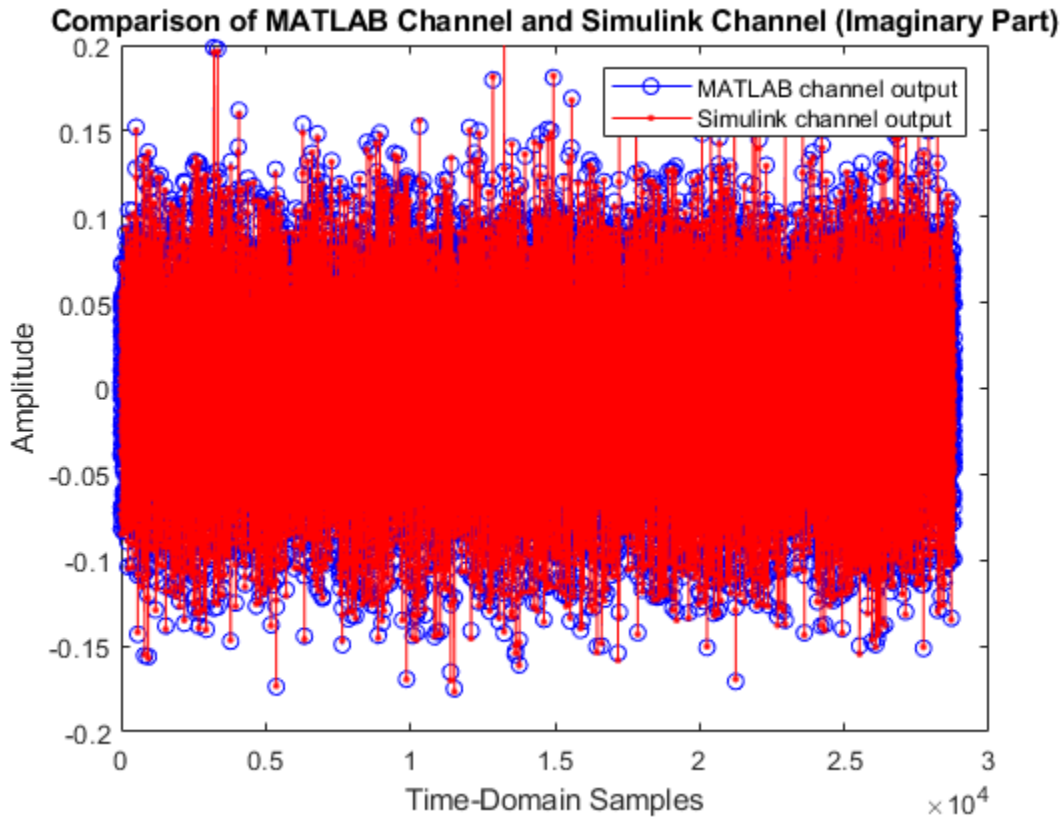
figure;
plot(rxDataBits,'-bo');
hold on;
plot(simulinkRxOut(1:length(rxDataBits)),'-r.');
legend('MATLAB Rx bits','Simulink Rx bits');
title('MATLAB and Simulink Decoded Bits');
ylim([-0.25 1.25]);
xlabel('Time-domain Samples');
ylabel('Amplitude');
```

See Also

Related Examples

- "HDL OFDM Receiver" on page 5-198
- "HDL OFDM Transmitter" on page 5-182
- "HDL Implementation of AWGN Generator" on page 4-44

HDL OFDM Transmitter

This example shows how to implement an OFDM-based wireless transmitter in Simulink® that is optimized for HDL code generation and hardware implementation.

This example shows the custom design of an orthogonal frequency-division multiplexing (OFDM) based transmitter. This transmitter model accepts payload data through the input port. It enables you to choose the modulation type and the punctured convolutional code rate of the data from a set of values. These two parameters control the effective data rate of transmission and are provided through the input ports of transmitter. The maximum data rate supported by the transmitter is 3 Mbps. The transmitter also accepts an input valid signal to control the transmission.

The transmitter in this example works in conjunction with the receiver in the "HDL OFDM Receiver" on page 5-198 example. The transmitter has a MATLAB® floating point equivalent function described in the "HDL OFDM MATLAB References" on page 5-169 example.

Transmitter Specification

This section explains the specifications of the transmitter related to the OFDM frame configuration and structure, bandwidth, and sample rate.

The transmitter model accepts two parameters, modTypeIndex and codeRateIndex, which allow you to specify the modulation type and punctured convolutional code rate, respectively, of the data. These two parameters are explained in the following tables:

modTypeIndex

```
Value     Represents Modulation Type
_____     _____

  0               BPSK
  1               QPSK
  2               16QAM
  3               64QAM
```

codeRateIndex

```
Value     Represents Code Rate
_____     _____

  0               1/2
  1               2/3
  2               3/4
  3               5/6
```

OFDM Frame Structure

Every OFDM system has a frame structure that shows the distribution of samples in the frequency domain across all its subcarriers. The frame structure is as shown in the figure. Each OFDM symbol is comprised of 72 subcarriers, and each OFDM frame consists of 36 OFDM symbols. The frame duration is 3 ms. The first OFDM symbol is formed by synchronization sequence (SS), second and third symbols are formed by reference signals (RS), and the fourth symbol is formed by Header. Data

is filled from the fifth symbol to the last (36th) symbol. Pilots are inserted between data such that there is one pilot for every five data subcarriers as shown below. These pilots help to detect and correct phase errors at the receiver.

The OFDM parameters used in the model are given below:

Parameter	Value
Sample rate	1.92 Msps
Subcarrier spacing	15 kHz
FFT Length	128
Bandwidth of OFDM signal	1.4 MHz
Active Subcarriers	72
Left guard subcarriers	28
Right guard subcarriers	27
Cyclic Prefix length	32
Data symbols per frame	32
Pilots per data symbol	12

Model Architecture

The following figure shows the high-level architecture of the OFDM transmitter. There are five different signals that form the OFDM frame: SS, RS, Header, Pilots, and Data. SS, RS, and Pilots are same for every frame. They are stored in separate look up tables (LUT) and accessed whenever

required. Header and Data vary based on the inputs given to the transmitter. Header bits are formed based on the input modulation type and code rate values. These header bits are processed through the Header chain as shown in the figure. Payload data is provided as an input to the transmitter. This data is processed through multiple stages in the Data chain. Individual stages in the Header and Data chains are explained in further sections.

These five signals are multiplexed based on their valid signals and stored in a RAM. The RAM holds these signals for a duration of one frame. Data stored in the RAM is read out and modulated by the OFDM Modulator block. The OFDM modulated signal is filtered with a passband frequency of 1.4 MHz and sent out as transmitter output.

File Structure

This example contains two Simulink models, an initialization script, and a MATLAB function:

- `whdlOFDMTransmitter.slx` — This is the top-level model in this example. It has an OFDM Transmitter subsystem that refers to the `whdlOFDMTx.slx` model. There is an external interface circuit for the OFDM Transmitter subsystem, which provides inputs and collects outputs from the subsystem. Simulating this model runs the remaining three files.

- `whdlexamples.OFDMTransmitterInit` — This script initializes the `whdlOFDMTransmitter.slx` model. The script is called in the `InitFcn` callback of the model.

- `whdlOFDMTx.slx` — This model implements the transmitter with total configurability.

- `whdlexamples.OFDMTxParameters` — This function generates parameters required for the `whdlOFDMTx.slx` model. This function is called in the Model Workspace of the model.

Transmitter Interface

The `whdlOFDMTransmitter.slx` model shows the OFDM Transmitter subsystem and its interface.

Copyright 2020 The MathWorks, Inc.

Model Inputs:

- *modTypeIndex* — Selects the type of symbol modulation to be applied to payload data, specified as a ufix2 scalar. This port accepts values 0, 1, 2, and 3, which correspond to modulation types BPSK, QPSK, 16QAM, and 64QAM.

- *codeRateIndex* — Selects the code rate of punctured convolutional code to be applied to payload data, specified as a ufix2 scalar. This port accepts values 0, 1, 2, and 3, which correspond to code rates 1/2, 2/3, 3/4, and 5/6.

- *data* — Input payload data, specified as a Boolean scalar.

- *valid* — Valid signal for the input data, specified as a Boolean scalar.

All input ports run at a sample rate of 30.72 Msps to support different configurations.

Model Outputs:

- *txData* — Transmitter output, returned as a complex scalar with fixdt(1,16,13) datatype sampled at 1.92 Msps.

- *txValid* — Control signal that validates *txData*, returned as a Boolean scalar sampled at 1.92 Msps.

- *ready* — Control signal that is used to sample input *data*, *modTypeIndex*, and *codeRateIndex* values, specified as a Boolean scalar sampled at 30.72 Msps.

Index Selector

The Index Selector subsystem samples the *modTypeIndex* and *codeRateIndex* signals at the rising edge of the *ready* signal. The subsystem retains the previous outputs if no rising edge exists on the *ready* signal.

This subsystem is active only when there is a rising edge on the input ready signal. The subsystem selects the Input modOrder and Input codeRateInd values for the current frame and outputs corresponding modTypeIndex and codeRateIndex values.

Data and Valid Selector

The Data and Valid Selector subsystem selects the input payload data and input valid signal based on the *ready* signal.

This subsystem is active only when the input ready signal is active. The subsystem selects the Input Valid value and outputs the valid signal. The subsystem also selects the payload data from an LUT based on the Input Valid signal and outputs it as data signal.

Structure of the Transmitter

The `whdlOFDMTx.slx` model is called within the OFDM Transmitter subsystem. It generates an OFDM transmitter waveform by processing input signals in multiple stages as shown below.

whdlOFDMTx

Copyright 2020-2021 The MathWorks, Inc.

Frame Controller and Input Sampler

The Frame Controller and Input Sampler subsystem generates control signals for later stages of the model. The subsystem also generates a *ready* output signal that is used for external interfacing. This

subsystem samples the input *modTypeIndex* and *codeRateIndex* values along with the first valid input sample. The transport block size for the current frame is selected from the Transport Block Size LUT based on the sampled *modTypeIndex* and *codeRateIndex* values. The subsystem also generates control signals for header generation followed by the preamble generation along with the first valid sample. Preamble generation refers to the generation of SS, RS, and Pilot signals. The control signal for data generation is asserted either after 9562 (maximum transport block size corresponding to 64-QAM modulation and 5/6 code rate) clock cycles from the first valid sample or after the transport block length of valid input data is stored for the current frame, whichever is later. Along with the data control signal, the *ofdmModReady* signal is asserted, which indicates the OFDM Modulator block to start modulation.

Select transport block size of the current frame based on the modTypeIndex and codeRateIndex values.

Frame Generator

The Frame Generator subsystem generates SS, RS, Header, Pilot, and Data signals, which are later OFDM-modulated. The Generate Preamble Control Signals subsystem that is in the Frame Generator subsystem, splits the input *preambleSet* control signal into *ss set*, *rs set*, and *pilot set* control signals, which generate SS, RS, and Pilot signals, respectively.

Frame Generator/Synchronization Sequence

The Synchronization Sequence subsystem accepts *ss set* control signal generated from the Frame Controller and Input Sampler subsystem. It is generated considering the length of SS sequence. The counter keeps incrementing and returns SS from an LUT. Once *ss set* becomes inactive, the counter stops. Output from LUT is upsampled by a factor of 2 to maintain the same sample time as that of the Header and Data subsystems. Reference Signals and Pilot subsystems operate in a similar way by storing the sequences in LUTs and accessing them whenever required.

Access synchronization sequence from LUT.

Upsampling is done to compensate the rate change due to Convolutional Encoder in Header and Data subsystems.

Frame Generator/Header

The Header subsystem accepts *modTypeIndex*, *codeRateIndex* and *fftLenIndex* as inputs. A *headerSet* signal starts the header formation. The Header Formation function converts the *modTypeIndex* and *codeRateIndex* values into their binary equivalents. For example, a *modTypeIndex* value of 1 is converted into two bits 01. Similarly, *codeRateIndex* values are converted into two equivalent bits. To learn more about these indices, refer to **Transmitter Specification**. *fftLenIndex* is not configurable and its value is fixed to 0. It is converted to 000, which represents an FFT length of 128. *fftLenIndex*, *modTypeIndex*, and *codeRateIndex* are represented using 3, 2, and 2 bits, forming a total of 7 bits. Additionally, 7 spare bits are added, all currently set to 0, forming a total of 14 Header bits.

These 14 bits are processed as shown in the figure. For proper error detection, General CRC Generator HDL Optimized block pads 16 CRC bits with [16 12 5 0] as the CRC polynomial. The Convolutional Encoder block encodes these 30 bits, that is 14 + 16, with [171 133] as the polynomial and a constraint length as 7. The encoding is processed in terminated mode, adding 6 null bits, that is 7&endash;1, to the CRC padded data. After encoding, these 36 bits result in 72 bits due to the 1/2 rate encoding. The output of the Convolutional Encoder block is a two-element vector that is serialized in the Serialized subsystem using the Serializer1D (HDL Coder) block, leading to rate transition by a factor 2. The serialized data is interleaved using the Interleaver block with 72 as the maximum block size and 18 as the number of columns. For more information on the Interleaver block, see the "HDL Interleaver and Deinterleaver" on page 5-242 example. The interleaved bits are BPSK-modulated using the LTE Symbol Modulator block to form a Header symbol.

Header bits are formed based on input modTypeIndex, codeRateIndex, and fftLenIndex values. Header bits are then processed through Header chain as shown below.

Frame Generator/Data

The Data subsystem stores input payload data, *dataIn*, and processes it through the Data chain.

Frame Generator/Data/Data and Control Signal Generation

The Data and Control Signal Generation subsystem consists of a RAM, where the input payload data, *dataIn*, is stored. A *dataSet* signal reads data from this RAM. This subsystem generates *start*, *end*, and *valid* control signals for the RAM data. It also selects the puncture vector based on the *codeRateIndex*.

Frame Generator/Data/Data Chain

The General CRC Generator HDL Optimized block appends a 32-bit CRC to the payload data from the RAM with [32 26 23 22 16 12 11 10 8 7 5 4 2 1 0] as the CRC polynomial. This CRC-padded data is scrambled with $x^7 + x^4 + 1$ as the polynomial and [1 0 1 1 1 0 1] as the initial state. The Convolutional Encoder block encodes the scrambled data in terminated mode with [171 133] as the polynomial and constraint length as 7. The encoded output is punctured using the Puncturer block with the puncture vector selected in the Data and Control Signal Generation subsystem. The output of the Puncturer block is a two-element vector and is serialized using Serializer1D (HDL Coder) block. The resultant data is interleaved in the Symbol Interleaver subsystem where the Split Data Into Symbols subsystem splits the input data into symbols and each of these symbols are bit-interleaved using the Interleaver block with 360 as the maximum block size and 15 as the number of columns. The supported input data symbol sizes to the Interleaver block are 60, 120, 240, and 360 for BPSK, QPSK, 16-QAM, and 64-QAM modulations, respectively. For more information on the Interleaver block, see the "HDL Interleaver and Deinterleaver" on page 5-242 example. The LTE Symbol Modulator block modulates the interleaved data using the modulation pattern selected based on the input *modTypeIndex*.

This subsystem shows the input payload data processed through Data Chain. Puncturer and Symbol Modulator blocks control the effective data rate.

Multiplexer

The Multiplexer subsystem multiplexes the SS, RS, and Pilot signals in the Multiplex Preamble Signals subsystem and the Header and Data signals in the Multiplex Header and Data Signals subsystem based on the valid signals generated by the Frame Generator subsystem.

Multiplex Preamble Signals

Multiplex Header and Data Signals

Frame Formation and OFDM Modulation

The Frame Formation and OFDM Modulation subsystem accepts the *preambleData* and *data* signals, and then multiplexes and writes them into a Dual Rate Dual Port RAM (HDL Coder). This RAM reads and writes data at different rates. The RAM writes data at 61.44 Msps. The RAM is filled with data such that it forms an OFDM frame structure as shown in the **Transmitter Specification** section.

The Generate OFDM Modulator Valid subsystem generates a valid input signal for the OFDM Modulator block at a sample rate of 1.92 Msps and generates a RAM address to read data from the RAM. The valid signal is in synchronization with the ready signal of the OFDM Modulator. The Make OFDM Valid Continuous subsystem selects the OFDM Modulator output based on the *validIn* signal. It gives out valid OFDM output in the presence of the *validIn* signal and a dummy OFDM symbol in the absence of *validIn* signal.

Discrete FIR Filter

The Discrete FIR Filter block filters the output of the Make OFDM Valid Continuous subsystem with a passband frequency of 1.4 MHz. The `whdlexamples.OFDMTxParameters` function computes the filter coefficients. The output of the filter is the final output of the transmitter.

Run the Transmitter

The transmitter can be connected back-to-back with the receiver that is explained in the "HDL OFDM Receiver" on page 5-198 example. For more information on how to use the transmitter and receiver Simulink models back-to-back, refer to the "HDL OFDM MATLAB References" on page 5-169 example.

To run the transmitter model, `OFDMTxVerification.m` script is provided with this example. The script chooses a custom frame configuration, payload data, and simulates the model. The script also collects the simulation outputs and validates them.

NOTE: These files are not available on the MATLAB search path. To copy these files locally to the user path, you must open this example.

Verification and Results

In this section, the OFDM Transmitter Simulink model is validated by comparing its output with its floating point equivalent function, `whdlexamples.OFDMTx`. For more information on this MATLAB function, see the "HDL OFDM MATLAB References" on page 5-169 example. To compare the output of the Simulink model with the MATLAB function, run the `OFDMTxVerification.m` script.

```
>> OFDMTxVerification

### Starting serial model reference simulation build
### Model reference simulation target for whdlOFDMTx is up to date.

Build Summary

0 of 1 models built (1 models already up to date)
Build duration: 0h 0m 10.643s
```

Imaginary Part of Tx Output

HDL Code Generation

To generate HDL code for this example, you must have HDL Coder™. Use `makehdl` and `makehdltb` commands to generate HDL code and HDL testbench for the OFDM Transmitter subsystem. Testbench generation time depends on the simulation time.

The resulting HDL code is synthesized for the Xilinx® Zynq®-7000 ZC706 evaluation board. The post place and route resource utilization is shown in the table below. The maximum frequency of operation is 230 MHz.

```
    Resources        Usage
 _____     _____
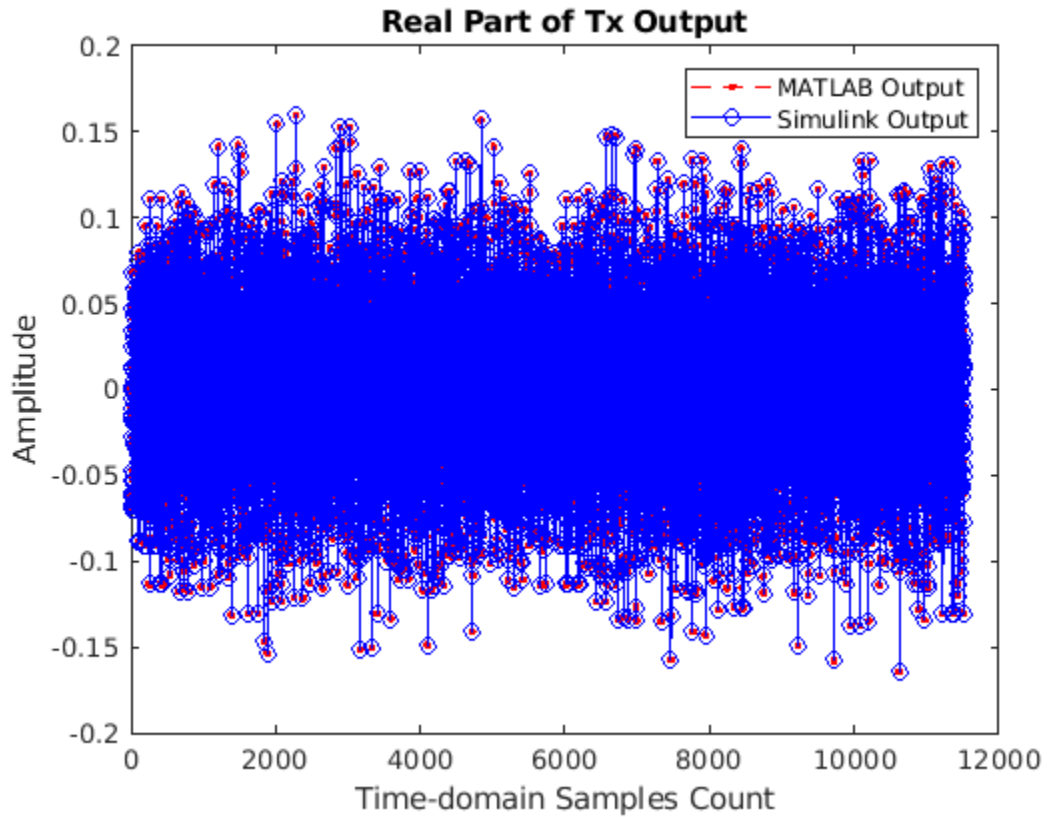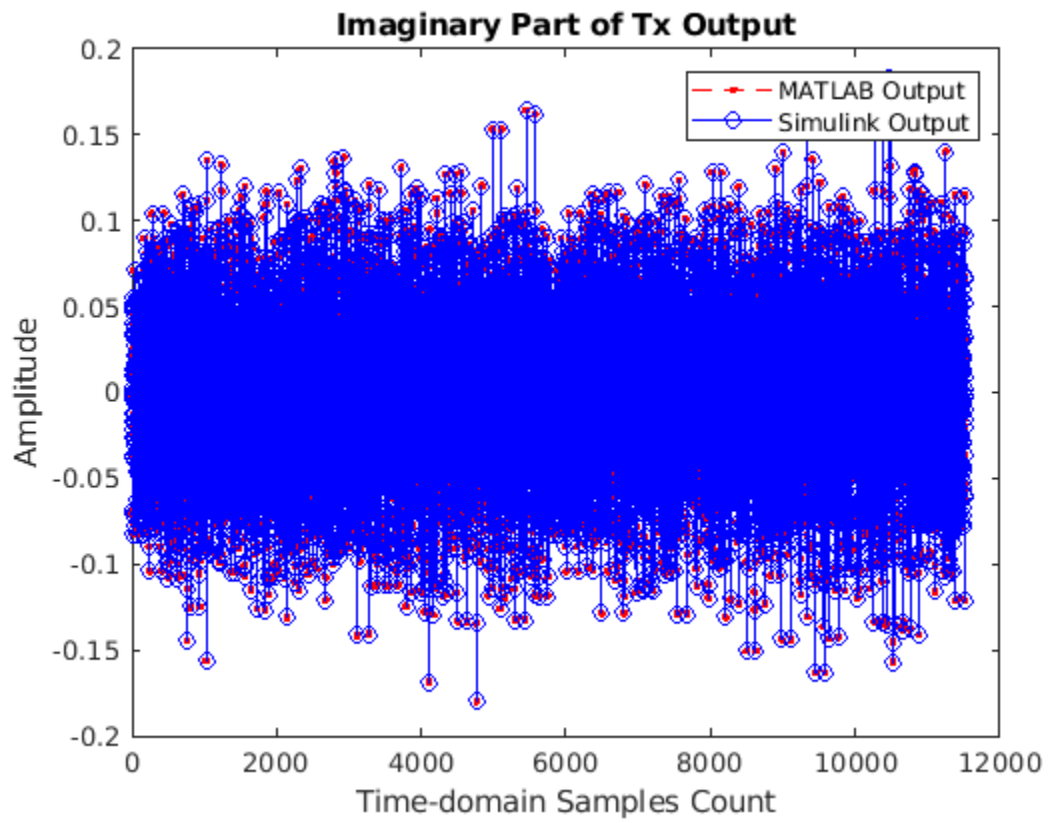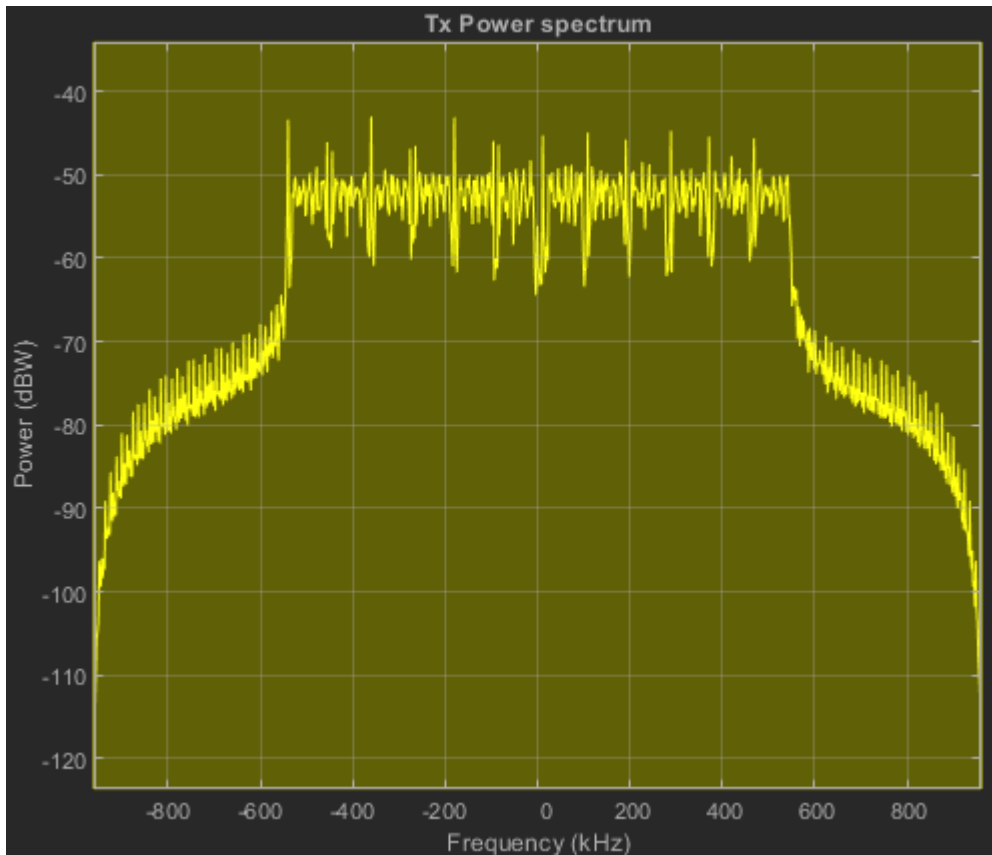
 Slice Registers     6373
 Slice LUT           4197
 RAMB36              5
 RAMB18              15
 DSP48               24
```

See Also

Blocks
OFDM Modulator | Puncturer | LTE Symbol Modulator | General CRC Generator HDL Optimized | Convolutional Encoder | Discrete FIR Filter | Serializer1D

Related Examples

- "HDL OFDM Receiver" on page 5-198
- "HDL OFDM MATLAB References" on page 5-169

HDL OFDM Receiver

This example shows how to implement an OFDM-based wireless receiver by using Simulink® blocks optimized for HDL code generation and hardware implementation.

The model shown in this example receives data and decodes it based on the orthogonal frequency division multiplexing (OFDM). The main purpose of this example is to model a custom HDL OFDM wireless communication receiver that can recover information in a real-time scenario and supports data rates up to 3 Mbps. This model enables you to configure parameters: symbol modulation types such as BSPK, QPSK, 16-QAM, and 64-QAM and code rates 1/2, 2/3, 3/4 and 5/6 through punctured convolution encoding. This model enables you to control impairments such as carrier frequency offset (CFO), carrier phase offset (CPO), and rayleigh fading channel, which significantly affect an OFDM-based communication system.

The receiver in this example works in conjunction with the transmitter in the "HDL OFDM Transmitter" on page 5-182 example. The receiver in this example has a MATLAB® floating point equivalent function described in the "HDL OFDM MATLAB References" on page 5-169 example.

Model Architecture

The following figure shows the architecture of an OFDM Receiver. The receiver samples the input at 1.92 Msps. These samples stream into the Rx Filter. The output from the Rx Filter stream into the Frequency Estimator and the Frequency Corrector. The Frequency Estimator and the Frequency Corrector estimate and correct CFO respectively and the samples stream into the Synchronizing Sequence (SS) Detector. The output of the SS Detector is used for the time synchronization. The time synchronized samples stream into the OFDM Demodulator, which demodulates the input and generates the frequency-domain subcarriers. The Subcarrier Parser parses the channel reference subcarriers, header subcarriers, and data subcarriers. The channel reference subcarriers stream into the Channel Estimator, which estimates the channel frequency response. The Channel Equalizer uses these estimates to equalize the header and data subcarriers in the frequency domain. The Header Recovery recovers the header information using the channel-equalized header subcarriers. The CPE Estimator estimates the common phase error (CPE) in the data sub carriers that get corrected by CPE Corrector. The Data Recovery uses the header information and the CPE-corrected data subcarriers to decode the data bits.

File Structure

Two Simulink models and three MATLAB files are used to construct this example.

- `whdlOFDMReceiver.slx` — Top level OFDM receiver Simulink model
- `whdlOFDMRx.slx` — Reference model used by the `whdlOFDMReceiver.slx` model
- `whdlexamples.OFDMReceiverInit.m` — Initialization script for `whdlOFDMReceiver.slx` initialized in the model's `InitFcn` callback.
- `whdlexamples.OFDMRxParameters.m` — Initialization function for `whdlOFDMRx.slx` initialized in the Model Workspace and model's `InitFcn` callback
- `whdlexamples.OFDMTx.m` — MATLAB floating-point equivalent transmitter function for generating a transmitter waveform. The generated transmitter waveform is used in the `whdlexamples.OFDMReceiverInit.m` script

Receiver Interface

This figure shows the top-level model in this example.

Model Inputs:

- *dataIn* — Input data, specified as a complex signed 16-bit signal sampled at 1.92 Msps.
- *validIn* — Control signal to validate the *dataIn*, specified as a Boolean scalar.
- *impairmentControl* — Bus signal to control the channel impairments.

The *impairmentControl* bus comprises following signals:

- *frequencyOffsetCorrectionType* — Control signal to indicate whether to select `Use internally estimated frequency offset` or `Use externally provided frequency offset` option for offset correction, specified as a Boolean scalar.
- *externalFrequencyOffset* — Real signed 14-bit CFO with range from -7400 Hz to 7400 Hz provided externally for CFO correction.
- *channelEqualizerControl* — Control signal to indicate whether to enable or disable channel equalization, specified as a Boolean scalar.
- *CPECorrectionControl* — Control signal to indicate whether to enable or disable CPE correction, specified as a Boolean scalar.

Model Outputs:

- *dataOut* — Decoded output data bits, returned as a Boolean scalar.
- *validOut* — Control signal to validate the *dataOut*, returned as a Boolean scalar.

- *diagBus* — Status signal with diagnostic outputs, returned as a bus signal.

Structure of the Receiver

The OFDM Receiver subsystem performs a set of operations in a sequence. This subsystem uses the `whdlOFDMRx.slx` reference model. This reference model is initialized in its Model Workspace and in the model `InitFcn` callback using the `whdlexamples.OFDMRxParameters` function. The following figure shows the top-level subsystems in the reference model.

Synchronization and OFDM Demodulation

The Synchronization and OFDM Demodulation subsystem performs frequency and time synchronization and OFDM demodulation.

The Frequency and Time Synchronization subsystem comprises Timing Adjust subsystem and CFO Estimation and Correction and SS Detection subsystem.

Count the input valid and synchronize the data with the timingOffset

CFO Estimation and Correction and SS Detection

The Delay length must be equal to the searchDuration in **Max Peak Searcher** subsystem

The CFO Estimation and Correction and SS Detection subsystem comprises CFO Estimation and SS Detection subsystem and Frequency Correction Nx subsystem, which perform frequency correction for the input signal. The estimate from the CFO Estimation and SS Detection subsystem is used for frequency correction if the *frequencyOffsetCorrectionType* signal on the top-level model interface is set to `Use internally estimated frequency offset`. The *externalFrequencyOffset* is used for frequency correction if the *frequencyOffsetCorrectionType* signal is set to `Use externally provided frequency offset`.

The CFO Estimation and SS Detection subsystem comprises CFO Estimation subsystem, Start Controller subsystem, Sync Signal Search subsystem, and Frequency Correction 1x subsystem that perform frequency correction on input signal.

The CFO Estimation subsystem uses the cyclic prefix correlation technique to estimate the CFO of the input signal. The CyclicPrefixCorrelator subsystem estimates one CFO value for every six OFDM symbols by averaging all the estimates in six OFDM symbols. The AngleAtMaximum subsystem selects the strongest correlation peak for every six OFDM symbols and records its phase angle. The AngleFilter subsystem implements an averaging filter to average all the recorded phase angles for a duration of 12 ms. The resulting phase angle serves as a final CFO estimate.

The Sync Signal Search subsystem implements the SS correlation. SS detection is performed by continuously cross-correlating the received signal with the SS signal in the time domain. In addition, the energy of the signal in the span of the correlator is computed on each time step and then scaled to generate a threshold. The Max Peak Searcher subsystem begins searching for the maximum correlation peak after 12 ms and searches for every 3 ms time window. The subsystem records the timing offset of the synchronization. The Start Controller function block indicates to the Max Peak Searcher subsystem the end of the 12 ms duration.

The timing offset recorded at the maximum correlation value by the Max Peak Searcher is transferred to the Timing Adjust subsystem to synchronize timing.

The OFDM Demodulator block demodulates the synchronized samples and generates subcarriers.

Channel and CPE Estimation and Correction

The Channel and CPE Estimation and Correction subsystem estimates the channel frequency response, equalizes the channel, performs CPE estimation, and corrects the CPE.

The Reference Signal Parsing MATLAB function block separates the OFDM symbols reserved for computing channel estimates.

The OFDM symbols reserved for computing channel estimates are streamed through Channel Estimation subsystem. The OFDM Channel Estimator block averages the estimates from the two symbols and outputs the final channel estimates. The estimates are streamed into the Channel Equalization subsystem that stores the estimates in a RAM and performs frequency-domain channel equalization using the OFDM Equalizer block for all the remaining OFDM symbols in the frame.

The Header and Data Parsing MATLAB function block separates the OFDM symbols corresponding to header and data symbols.

The frequency domain channel-equalized data subcarriers stream through the Common Phase Error Estimation and Correction subsystem. In the frequency estimation process, there is always a small estimation error due to the channel impairments. This estimation error results in a residual frequency offset in the channel-equalized subcarriers. This results a CPE in all the subcarriers in an OFDM symbol and changes from symbol to symbol. The CPE Estimation subsystem estimates the CPE on each OFDM symbol using the 12 pilot subcarriers. The pilots are the known subcarriers and any phase rotation in the received symbols is estimated by using the pilots. The estimates drawn from the same symbol are averaged to get the final estimate. The symbol is stored in the Symbol Buffer MATLAB function block during estimation. Once the estimate is ready, the symbol is read from this buffer block and the CPE Correction subsystem corrects the CPE in the data subcarriers with that estimate.

Header and Data Recovery

The Header and Data Recovery subsystem recovers header information and data bits.

The Header Recovery subsystem recovers the header information to decode data bits. The frequency domain channel-equalized header subcarriers stream into the Header Recovery subsystem. The LTE Symbol Demodulator block performs BPSK soft symbol demodulation. The Channel Coding subsystem is equipped with a Deinterleaver subsystem and Viterbi Decoder block. The Deinterleaver subsystem performs deinterleaving with a maximum block size of 72 and the number of columns as 18. The Viterbi Decoder block performs 1/2 rate viterbi decoding. For more information about the Deinterleaver subsystem, see the "HDL Interleaver and Deinterleaver" on page 5-242 example. The General CRC Syndrome Detector HDL Optimized block uses a 16-bit CRC checksum to validate the decoded bits from the Viterbi Decoder block. If the CRC checksum fails, the General CRC Syndrome Detector HDL Optimized block generates an error signal.

The Data Recovery subsystem uses header information to decode data bits. The header information is stored in the registers. These registers are used to access the header information. The LTE Symbol Demodulator block performs soft bit BPSK, QPSK, 16-QAM, or 64-QAM symbol demodulation associated with the modulation type retrieved from the header information. The Channel Coding

subsystem is equipped with the Deinterleaver, Depuncturer, and Viterbi Decoder blocks. Each code rate is assigned a predefined punctured vector pattern. Based on the code rate retrieved from the header information, the Channel Coding subsystem performs deinterleaving and depuncturing followed by viterbi decoding. For more information on the Deinterleaver block, see the "HDL Interleaver and Deinterleaver" on page 5-242 example. The decoded bits are streamed through the Descrambler subsystem. The General CRC Syndrome Detector HDL Optimized block uses a 32-bit CRC checksum to validate the descrambled bits. If the CRC checksum fails, the General CRC Syndrome Detector HDL Optimized block generates an error signal.

Diagnostic Bus Formation

The Diagnostic Bus Formation subsystem creates a bus signal for some status signals of the receiver. This bus can be used to analyze the receiver when deployed onto the hardware.

The data bits are decoded in the Data Recovery subsystem. The decoded bits stream out of the receiver and stored to workspace in the Capture Data Bits subsystem in the top-level receiver model. The Diagnostics Decoder subsystem decodes the source-coded header information and counts the number of synchronized frames, number of header CRC passes and failures, and the number of data CRC passes and failures in the bus signal formed in the Diagnostic Bus Formation subsystem. The Simulink display blocks display the Diagnostics Decoder information.

Diagnostic Decoder

Run the Receiver

Connect the receiver back-to-back with the transmitter in the "HDL OFDM Transmitter" on page 5-182 example and run the Simulink model. For more information on how to connect the transmitter and the receiver Simulink models back-to-back see the "HDL OFDM MATLAB References" on page 5-169 example.

The following files describe a procedure to initialize, generate inputs, run, and verify the `whdlOFDMReceiver.slx` model using the `whdlexamples.OFDMReceiverInit.m` initialization script. You can choose a custom transmitter waveform and a channel impairment of your choice from the Custom Frame Configuration section in these files.

- `OFDMRxRealTimeSimulationDisplay.m` — This script mimics a channel in a real-time scenario. You can choose any available channel impairment and run the script. The script displays the outputs and generates plots of estimated frequency offset and SS correlation.

- `OFDMRxFadingChannelResponseDisplay.m` — This script mimics only the fading channel. You can choose only the fading channel impairment and run the script. The script displays the outputs

and generates the plots of channel impulse response and the comparison of estimated frequency response with the frequency response, derived from the impulse response.

Note: These files are not available on the MATLAB search path. To copy these files locally to the user path, you must open this example.

Verification and Results

The `whdlexamples.OFDMRx.m` script is a MATLAB floating point equivalent of the reference model `whdlOFDMRx.slx`. The Simulink model and MATLAB floating point equivalent script are compared in the "HDL OFDM MATLAB References" on page 5-169 example.

Run the `OFDMRxRealTimeSimulationDisplay.m` script to run the receiver.

```
>> OFDMRxRealTimeSimulationDisplay

### Starting serial model reference simulation build
### Model reference simulation target for whdlOFDMRx is up to date.

Build Summary

0 of 1 models built (1 models already up to date)
Build duration: 0h 0m 4.893s

 Number of header CRC failed = 0 per 4

 Number of bit errors = 0 per 15208
```


Frequency Estimate

Run the `OFDMRxFadingChannelResponseDisplay.m` script to run the receiver.

```
>> OFDMRxFadingChannelResponseDisplay
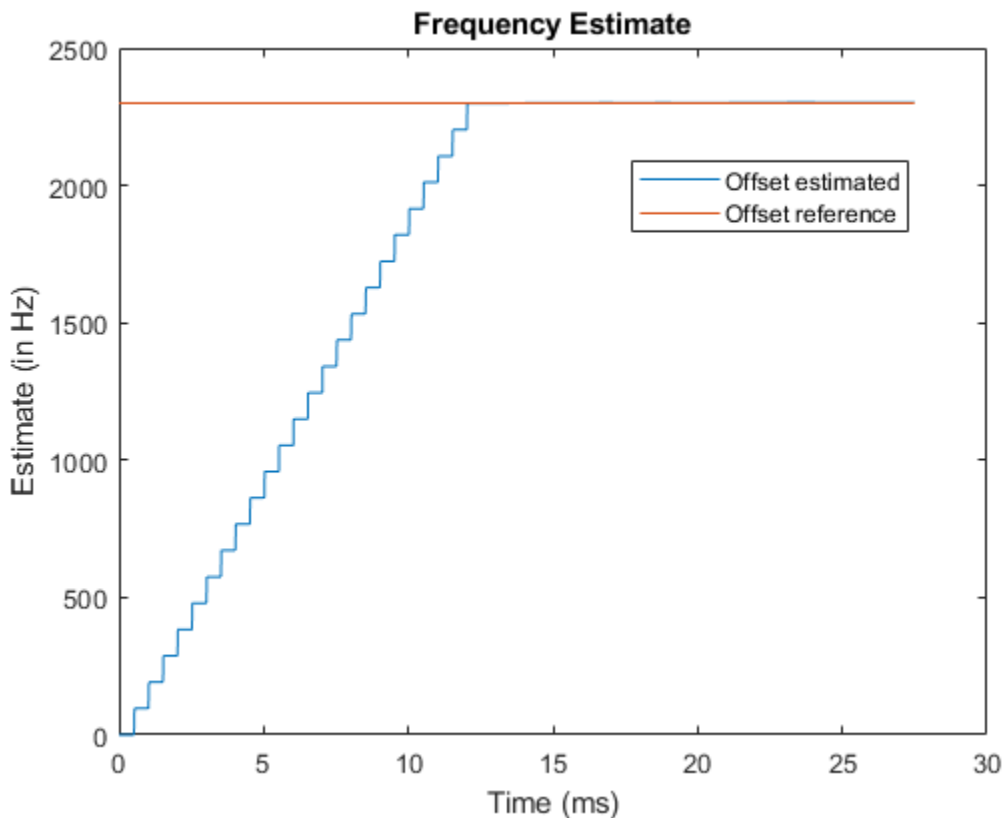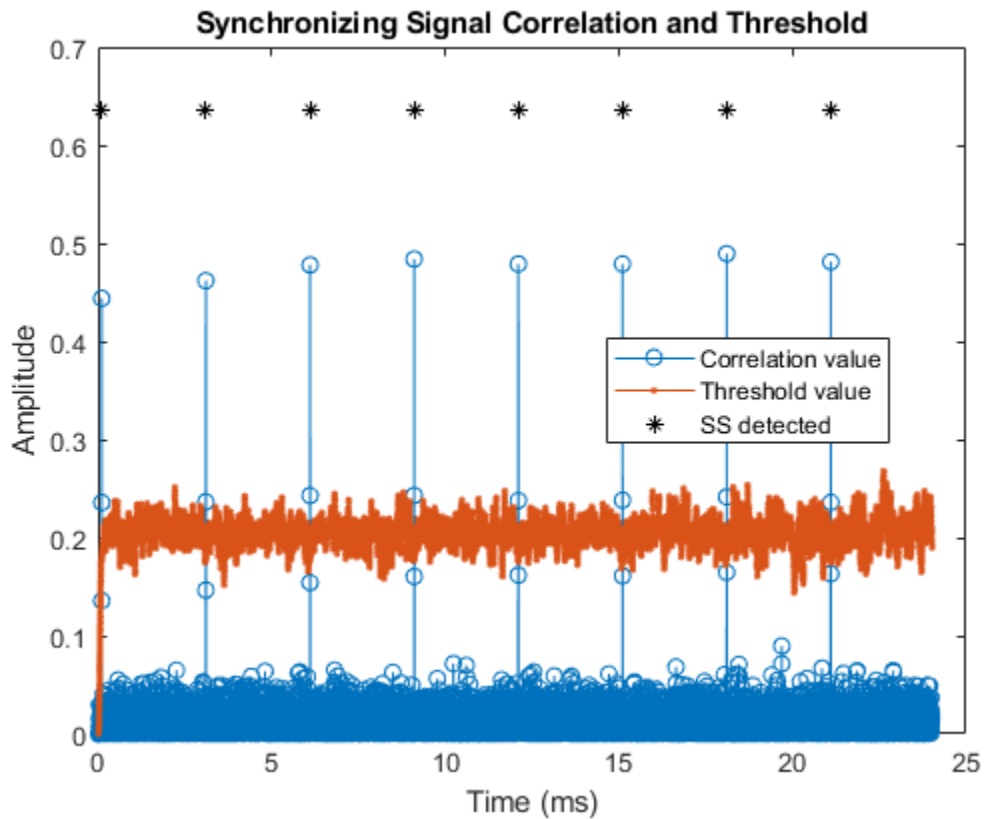
### Starting serial model reference simulation build
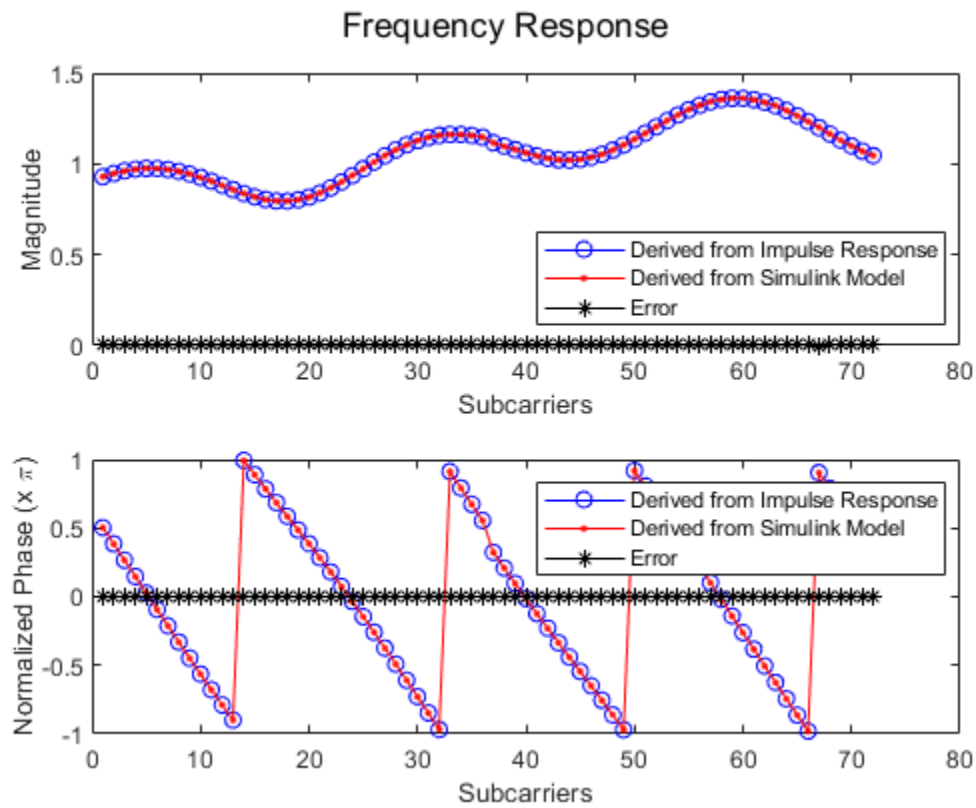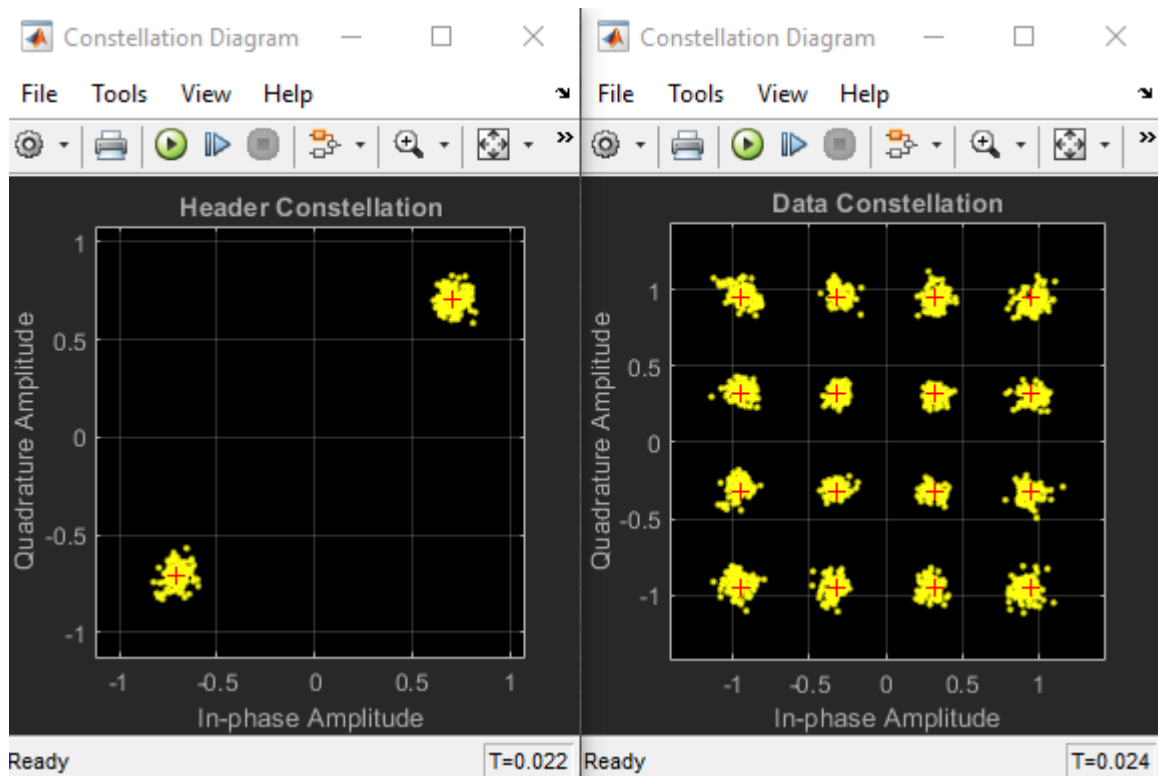### Model reference simulation target for whdlOFDMRx is up to date.

Build Summary

0 of 1 models built (1 models already up to date)
Build duration: 0h 0m 0.928s

 Number of header CRC failed = 0 per 1

 Number of bit errors = 0 per 3162
```

Frequency Response

You can see the constellation plot on the constellation scope. The scopes can be activated by using the Control Scope button in the `whdlOFDMReceiver.slx` model.

HDL Code Generation

To generate the HDL code for this example, you must have HDL Coder™. Use `makehdl` and `makehdltb` commands to generate HDL code and HDL testbench for the OFDM Receiver subsystem. The testbench generation time depends on the simulation time.

The resulting HDL code is synthesized for a Xilinx® Zynq®-7000 ZC706 evaluation board. The post place and route resource utilization and are shown in the table below. The maximum frequency of operation is 202 MHz.

```
        Resources        Usage
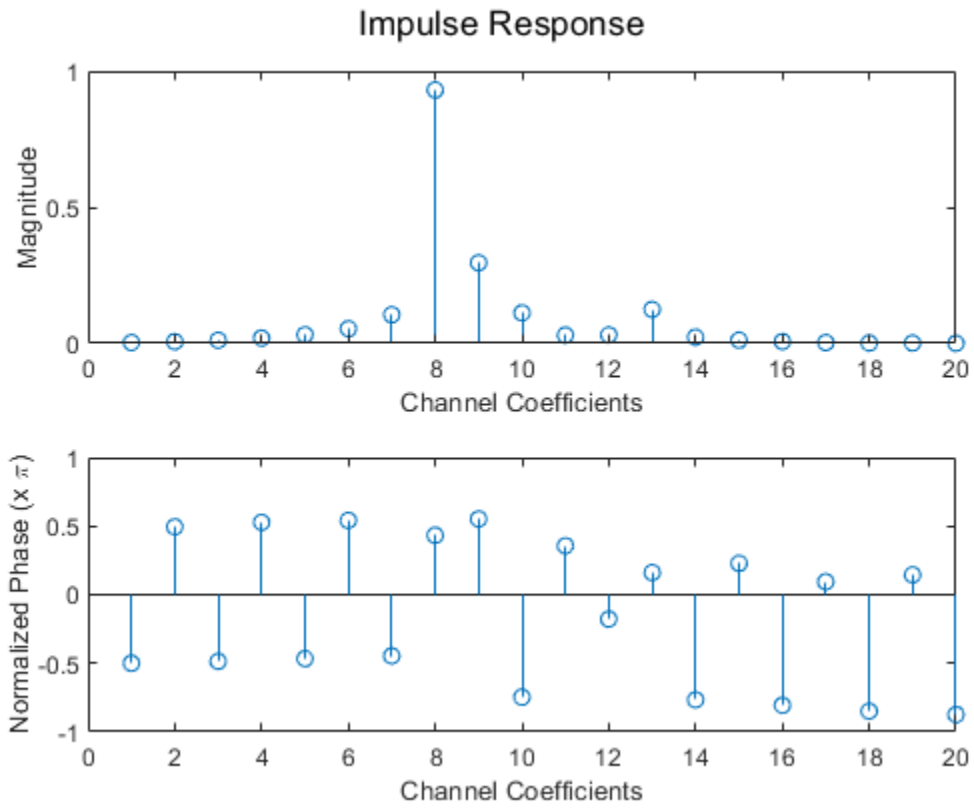
        _____  _____

        Slice Registers  46642
        Slice LUT        38457
        RAMB36           14
        RAMB18           12
        DSP48            88
```

See Also

Blocks
LTE Symbol Demodulator | Depuncturer | Viterbi Decoder | General CRC Syndrome Detector HDL Optimized | OFDM Demodulator | OFDM Channel Estimator

Related Examples

- "HDL OFDM Transmitter" on page 5-182
- "HDL OFDM MATLAB References" on page 5-169

Deploy Custom Communication Systems on FPGAs and SoCs

This section contains the list of examples that show how to deploy custom communication systems Wireless HDL Toolbox™ reference applications on FPGAs and SoCs.

- "OFDM Transmit and Receive Using Analog Devices AD9361/AD9364" (Communications Toolbox Support Package for Xilinx Zynq-Based Radio): Deploy an orthogonal frequency division multiplexing (OFDM) transmit and receive algorithm.

This example reuses the custom communication systems Simulink® models to generate HDL for the FPGA logic. They use hardware-software co-design modeling techniques and hardware support packages to add all the software modeling and interfacing required to implement the algorithm in real-time on hardware.

See Also

Related Examples
- "HDL OFDM Receiver" on page 5-198
- "HDL OFDM Transmitter" on page 5-182

WLAN HDL Time and Frequency Synchronization

This example shows how to implement a WLAN time and frequency synchronization model that is optimized for HDL code generation and hardware implementation. Time and frequency synchronization are the key steps to recover wireless local area network (WLAN) packet information.

The model estimates and corrects the time and frequency offsets in the received WLAN signal that are introduced by wireless channel and radio frequency (RF) front-end impairments. Initially, the model performs coarse time and frequency estimation and corrections on the received signal. Then, the model fine tunes the time and frequency estimation and corrections on the received signal to remove any residual offsets. The model supports 20, 40, and 80 MHz bandwidth options for non-high throughput (Non-HT), high throughput (HT), very high throughput (VHT), and high efficiency (HE) frame formats. The example compares the Simulink® model output with the MATLAB® functions by using WLAN Toolbox™ features.

WLAN packet decoding includes these stages: time and frequency synchronization, OFDM demodulation, channel estimation & equalization, format detection, signal decoding, and data decoding.

In this decoding procedure, only the time and frequency synchronization stage can be optimized for HDL code generation. The HDL support is extended for other stages in a future release.

In MATLAB, run this command to open the example model.

```
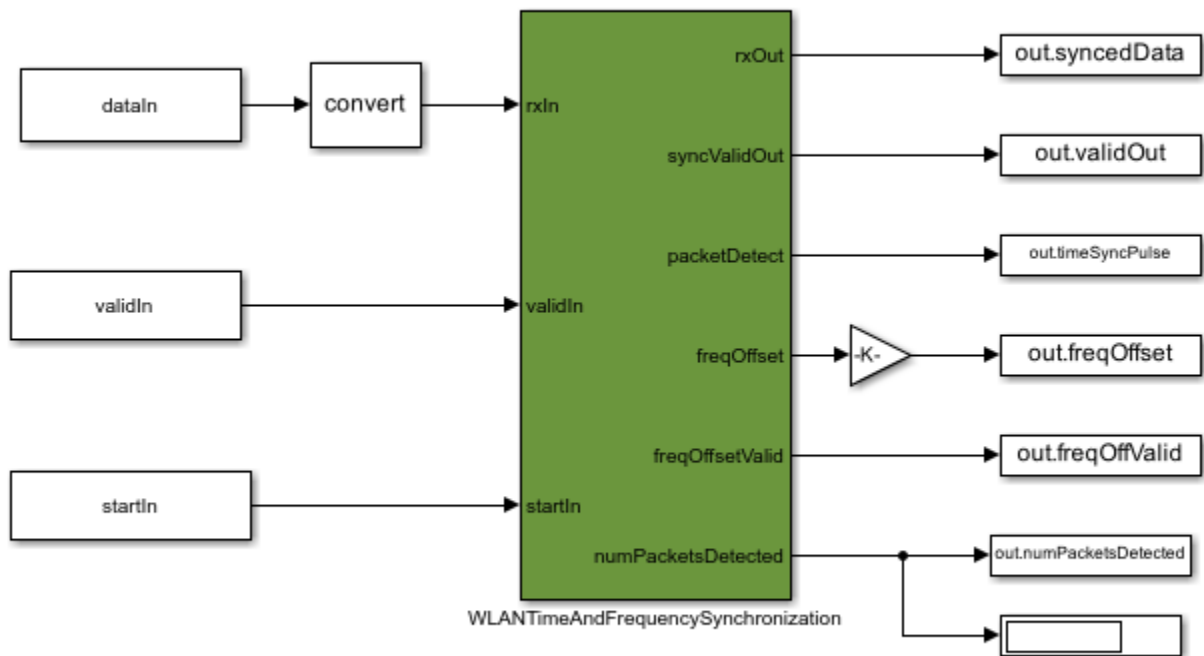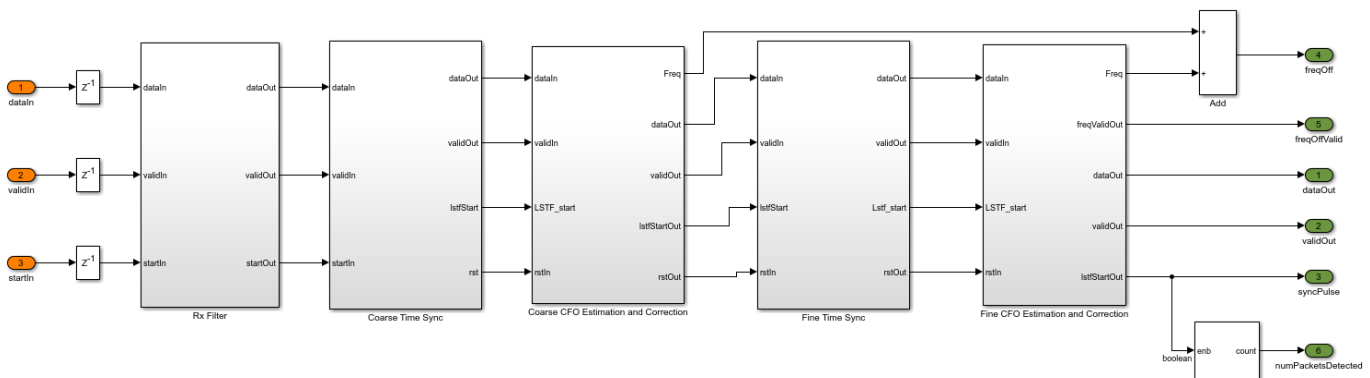model_name = 'wlanhdlTimeAndFrequencySynchronization';
open_system(model_name);
```

Copyright 2020 The MathWorks, Inc.

The WLANTimeAndFrequencySynchronization model contains these subsystems: Coarse Time Sync, Coarse CFO Estimation and Correction, Fine Time Sync, and Fine CFO Estimation and Correction.

In MATLAB, run this command to open the WLANTimeAndFrequencySynchronization subsystem.

```
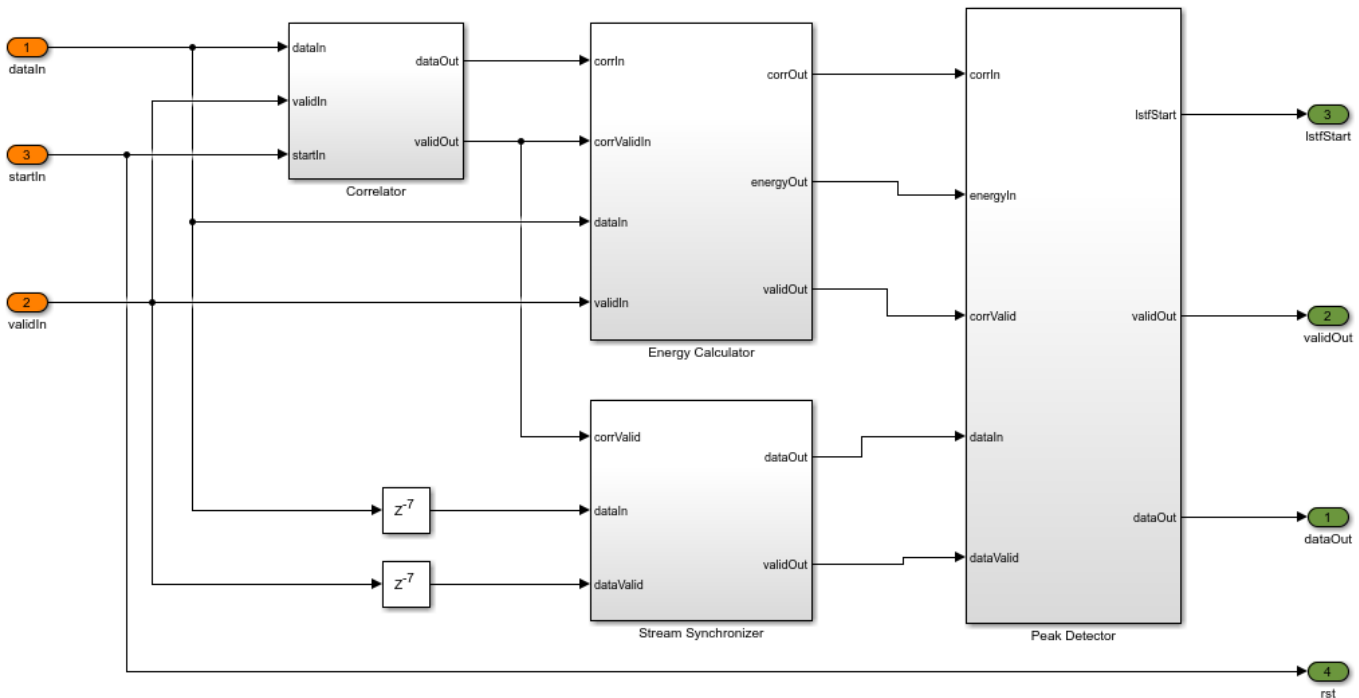open_system([model_name '/WLANTimeAndFrequencySynchronization'],'force');
```


Coarse Time Synchronization

The coarse time synchronization algorithm implements a double sliding window for correlation as described in the MATLAB function wlanPacketDetect.m. The Coarse Time Sync subsystem uses

the autocorrelation of legacy short training field (L-STF) symbols to return an estimated packet-start offset. The Peak Detector subsystem compares the correlation metrics with the energy of the signals and determines the start of the packet. In the next stage, the fine symbol timing detection refines this packet start estimate using the legacy long training field (L-LTF).

In MATLAB, run this command to open the Coarse Time Sync subsystem.

```
open_system([model_name '/WLANTimeAndFrequencySynchronization/Coarse Time Sync']);
```

Coarse CFO Estimation and Correction

Considering the start of the packet from the Coarse Time Sync subsystem, Coarse CFO Estimation and Correction subsystem performs autocorrelation on the input using a L-STF and averages the calculated correlation metrics over a window of the L-STF duration. Then, the subsystem estimates the carrier frequency offset (CFO) by considering the angle of the resulted metric.

In MATLAB, run this command to open the Coarse CFO Estimation subsystem.

```
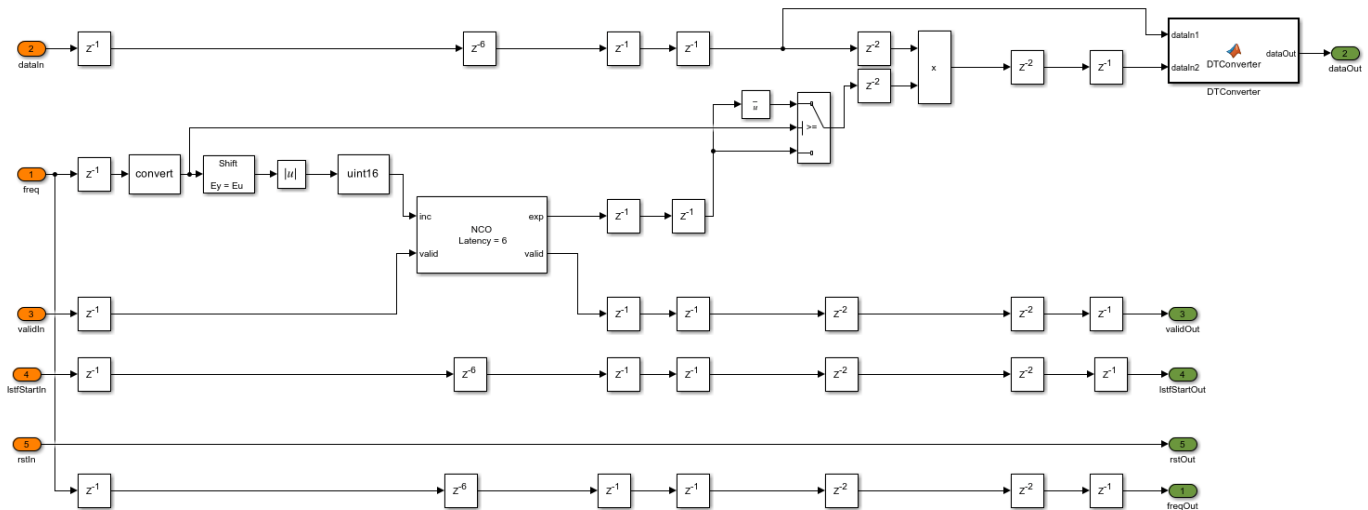open_system([model_name '/WLANTimeAndFrequencySynchronization/Coarse CFO Estimation and Correctic
```

This subsystem uses the CFO estimate to correct the frequency offset.

In MATLAB, run this command to open the Coarse CFO Correction subsystem.

```
open_system([model_name '/WLANTimeAndFrequencySynchronization/Coarse CFO Estimation and Correctic
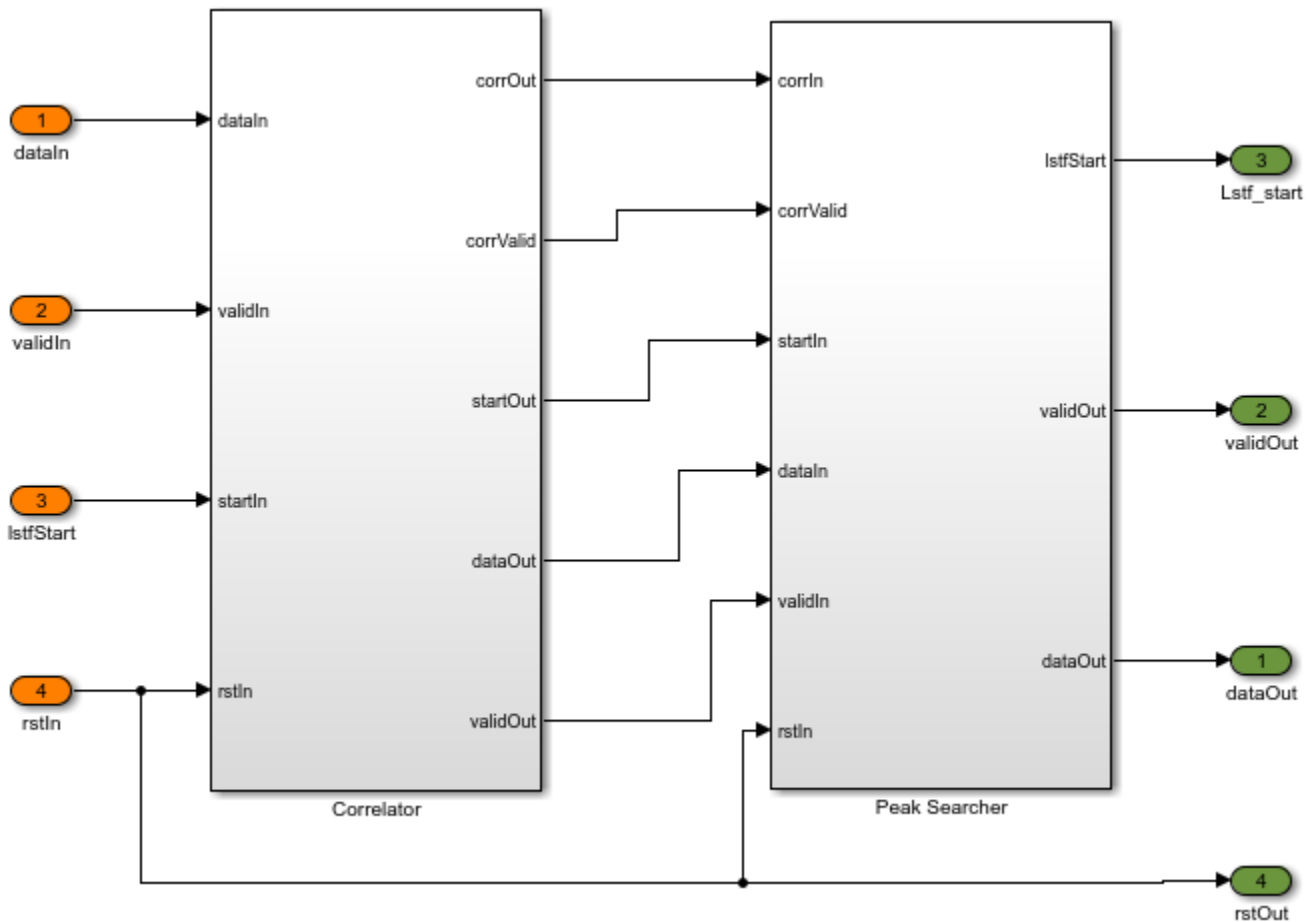```



**Fine Time Synchronization**

The Fine Time Sync subsystem takes the coarsely corrected time and frequency offset waveform for fine time offset synchronization. The Correlator subsystem cross correlates the received signal with the locally generated L-LTF. The Peak Searcher subsystem searches the maximum correlation peak and then synchronizes the signal.

In MATLAB, run this command to open the Fine Time Sync subsystem.

```
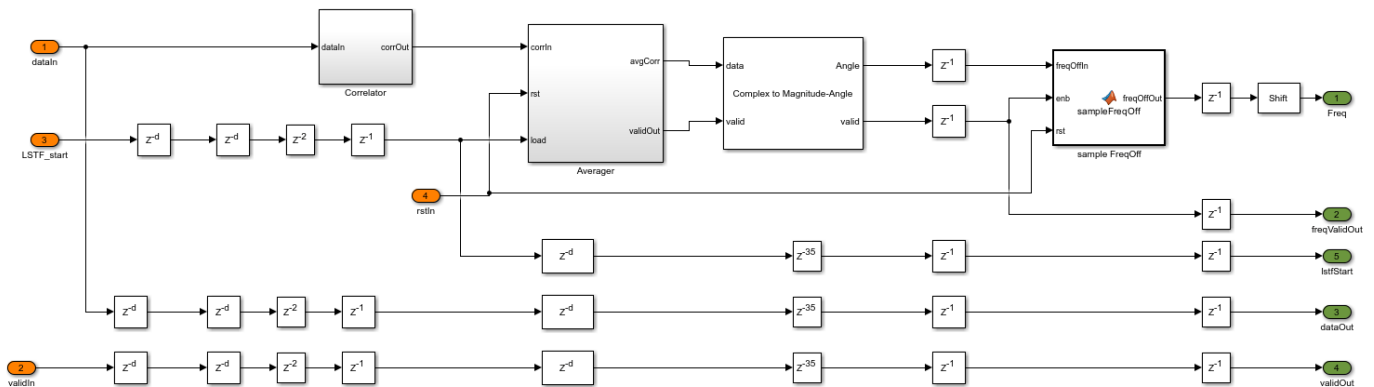open_system([model_name '/WLANTimeAndFrequencySynchronization/Fine Time Sync']);
```

Fine CFO Estimation and Correction

The Fine CFO Estimation and Correction subsystem takes a fine time synced waveform as an input for fine tuning the frequency offset. This subsystem estimates and corrects CFO to remove any residue left after coarse frequency correction, performs fine CFO estimation similar to coarse estimation by using the L-LTF instead of the L-STF, and estimates the frequency offset by considering the angle of the averaged correlations.

In MATLAB, run this command to open the Fine CFO Estimation subsystem.

```
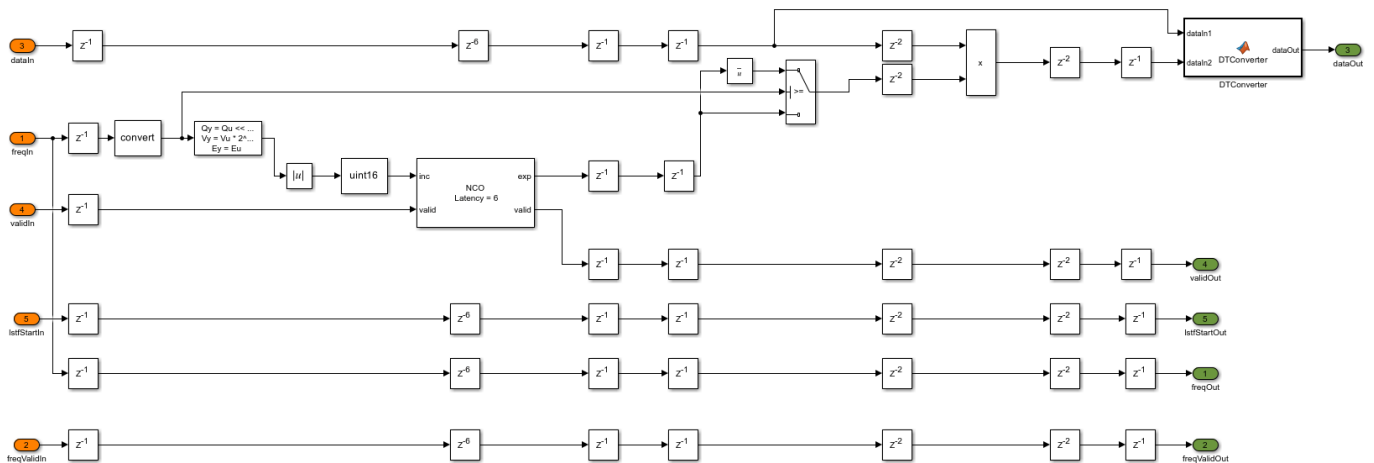open_system([model_name '/WLANTimeAndFrequencySynchronization/Fine CFO Estimation and Correction/
```

The Fine CFO Correction subsystem uses the estimated fine CFO for correcting the residual frequency offset and then outputs the corrected WLAN received signal.

In MATLAB, run this command to open the Fine CFO Correction subsystem.

```
open_system([model_name '/WLANTimeAndFrequencySynchronization/Fine CFO Estimation and Correction,
```


Model Interface and Verification

The example model accepts the received waveform as an input along with valid and start signals. The model returns a synchronized waveform as an output along with a valid signal. The other outputs in the example include a packet detected flag, a CFO estimate along with its valid and the number of packets detected as an output. CFO estimate is the sum of coarse CFO and fine CFO estimates. The `wlanFrontEndInit` script provides the input to the model. The `wlanWaveformGenerator.m` function in the script generates the VHT 20 MHz frame, which is passed through the TGac channel with a delay profile of Model A. The additive white Gaussian noise (AWGN) at 30 dB signal-to-noise ratio (SNR) is added with other channel impairments of a 10 kHz CFO and a timing offset of '25'.

```
fprintf('\n Simulating HDL time and frequency synchronization \n');
out = sim('wlanhdlTimeAndFrequencySynchronization.slx');
fprintf('\n HDL simulation complete. %d packet detected.',out.numPacketsDetected(end));
```

```
Simulating HDL time and frequency synchronization
```

```
 HDL simulation complete. 1 packet detected.
```

The outputs of example are verified by using WLAN Toolbox functions. Specify the same input waveform for the Simulink model and its MATLAB equivalent function and then compare outputs.

```
fprintf('\n Comparing WLAN MATLAB time and frequency synchronization \n')
inputWaveformRef = inputWaveform(1:end-length(Hd.Numerator)+1);
inputWaveformRef = filter(Hd.Numerator,1,inputWaveformRef);

% WLAN packet detection
[startOffset,Mn]=wlanPacketDetect(inputWaveformRef,CBW);
rxWave1 = inputWaveformRef(startOffset+1:end);

% Coarse CFO estimation and correction
coarseFreqOff = wlanCoarseCFOEstimate(rxWave1,CBW);
rxWave2 = hwlanFrequencyOffsetCorrect(rxWave1,fs,coarseFreqOff);

% Fine time synchronization
searchBufferLLTF = rxWave2(1:wlanConfig.lstfLen*10+wlanConfig.lltfLen*3);
[offset,MN] = wlanSymbolTimingEstimate(searchBufferLLTF,CBW);
rxWave3 = rxWave2(offset+1:end);

% Fine CFO estimation and correction
LTFs = rxWave3(10*wlanConfig.lstfLen+(1:wlanConfig.lltfLen*2));
fineFreqOff = wlanFineCFOEstimate(LTFs,CBW);

matOut = hwlanFrequencyOffsetCorrect(rxWave3,fs,fineFreqOff);
fprintf('\n MATLAB simulation complete. \n');

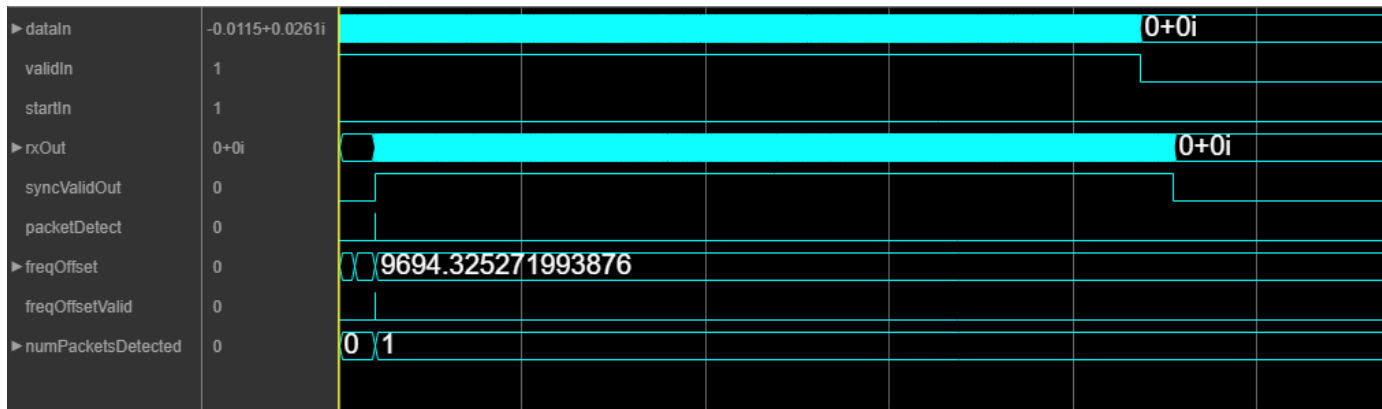simData = out.syncedData;
simValid = out.validOut;

simOut = double(simData(simValid));
```

```
 Comparing WLAN MATLAB time and frequency synchronization

 MATLAB simulation complete.
```

Simulation Results

The example synchronizes the time and frequency of the input waveform generated using the `wlanFrontEndInit.m` script and outputs the time and frequency corrected waveform as shown in this timing diagram.

The timing diagram shows that the output rxOut is synchronized at the start of the L-STF and that the estimated frequency offset is 9.695 kHz, which is close to the introduced frequency offset of 10 kHz.

Comparison of Simulink Output and MATLAB Reference Output

```
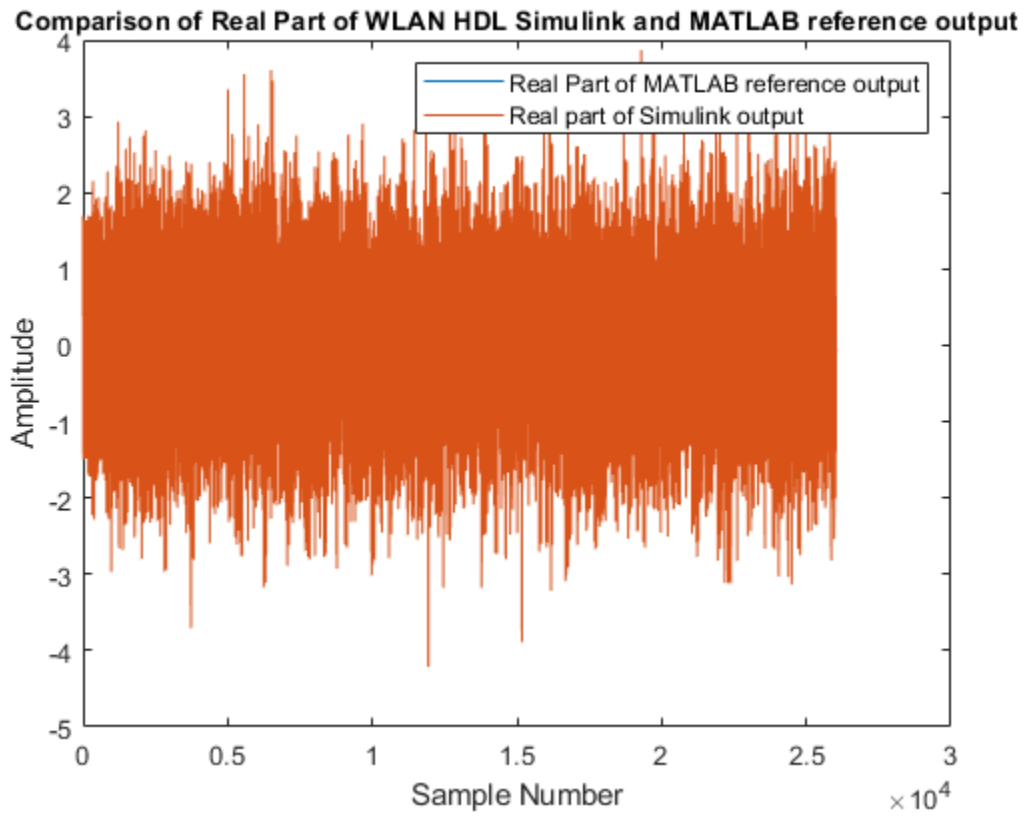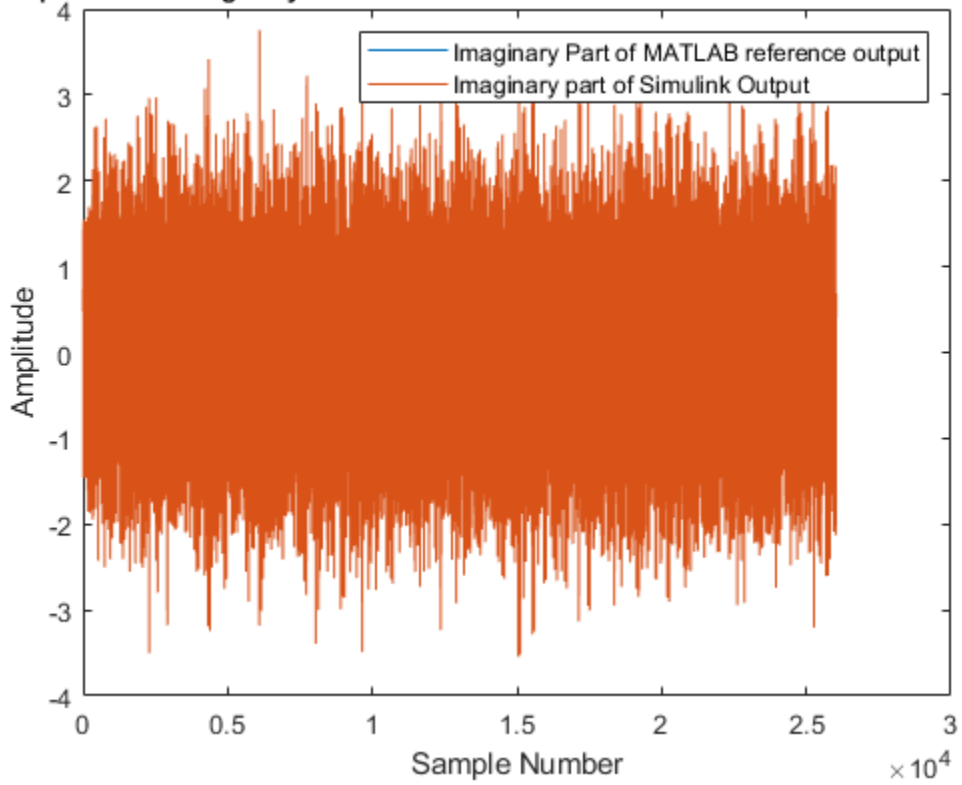plot(real(matOut));
hold on;
simOut = simOut(1:length(matOut));
plot(real(simOut));
title('Comparison of Real Part of WLAN HDL Simulink and MATLAB reference output','FontSize', 10)
xlabel('Sample Number');
ylabel('Amplitude');
legend('Real Part of MATLAB reference output','Real part of Simulink output');

figure;
plot(imag(matOut));
hold on;
simOut = simOut(1:length(matOut));
plot(imag(simOut));
title('Comparison of Imaginary Part of WLAN HDL Simulink and MATLAB reference output','FontSize'
xlabel('Sample Number');
ylabel('Amplitude');
legend('Imaginary Part of MATLAB reference output','Imaginary part of Simulink Output');
```

Comparison of Real Part of WLAN HDL Simulink and MATLAB reference output

Comparison of Imaginary Part of WLAN HDL Simulink and MATLAB reference output

See Also

Functions
wlanPacketDetect | wlanFineCFOEstimate | wlanCoarseCFOEstimate | wlanSymbolTimingEstimate

HDL Implementation of WLAN Receiver

This example shows how to design a wireless local area network (WLAN) receiver that can recover signal and data field information from a WLAN signal. This example supports both binary convolutional coding (BCC) and low-density parity-check (LDPC) channel coding techniques. The Simulink® model in this example is optimized for HDL code generation and hardware implementation.

The example supports single-input single-output (SISO) orthogonal frequency division multiplexing (OFDM) modulation, 20 MHz bandwidth option for non-high-throughput (non-HT), high-throughput (HT), and very-high-throughput (VHT) frame formats and 40 MHz bandwidth option for high-throughput (HT) and very-high-throughput (VHT) frame formats for BCC channel coding. The example supports 20 MHz and 40 MHz bandwidth options for HT and VHT frame formats for LDPC channel coding. You can select the channel coding type from the `Input Generation` subsystem. For more information on the WLAN frame formats and frame structure, see "WLAN PPDU Structure" (WLAN Toolbox). The block diagram shows the high-level overview of a WLAN receiver design. The "WLAN HDL Time and Frequency Synchronization" on page 5-217 explains the functionality of the `Time and Frequency Synchronization` subsystem. This subsystem accepts Rx input waveform and outputs time and frequency synchronized waveform.

To design a WLAN receiver, along with the `Time and Frequency Synchronization` subsystem, the model requires a few more blocks as shown in the block diagram.

The OFDM Demodulation block converts the time-domain signal to frequency-domain subcarriers. The Channel Estimator block uses demodulated legacy long training fields (L-LTFs) of a WLAN signal to estimate the channel frequency response. To equalize the pilot and data subcarriers of non-HT portion of the WLAN signal, the channel equalizer uses the estimated channel frequency response. The non-HT portion of the WLAN signal includes legacy SIGNAL(L-SIG) field, high-throughput SIGNAL fields 1 and 2 (HT-SIG 1 and 2), very-high-throughput SIGNAL fields A and B (VHT-SIG-A and VHT-SIG-B) and legacy Data field. Similarly, the channel is estimated using the demodulated HT or VHT LTFs of a WLAN signal to equalize the pilot and data subcarriers of HT or VHT portion of the WLAN signal. The HT or VHT portion of the WLAN signal includes VHT-SIG-B, HT-Data, and VHT-Data.

After equalization, non-HT common phase error (CPE) estimation is performed using non-HT pilots. The estimated CPE is used to correct data subcarriers of the non-HT portion of the WLAN signal. Similarly, HT or VHT common phase error (CPE) estimation is performed using HT or VHT pilots. The estimated CPE is used to correct data subcarriers of the HT or VHT portion of the WLAN signal. Common phase noise error corrected data is used for frame format detection, signal, and Data field recovery.

5-227

Frame format detector and SIGNAL field recovery detects the frame format between non-HT, HT, and VHT frames and decodes the transmitted bits from WLAN signal fields L-SIG, HT-SIG 1 and 2 and VHT-SIG-A 1 and 2. If the detected frame format is non-HT, the frame format controller passes non-HT CPE corrected data to Data recovery. Alternatively, if the detected frame format is HT or VHT, the frame format controller passes HT or VHT CPE corrected data to Data recovery. Data recovery decodes the transmitted bits from WLAN data fields L-Data, HT-Data and VHT-Data using signal parameters such as modulation and coding scheme (MCS), physical layer convergence protocol service data unit (PSDU) length, and the channel coding type. VHT-SIG-B is also decoded as part of Data recovery for the VHT frame. The example validates the Simulink® WLAN receiver model output by using MATLAB® functions in WLAN Toolbox™.

Model Architecture

Open the `wlanhdlReceiver` model to run the example. This figure shows the high-level overview of a WLAN receiver model.

```
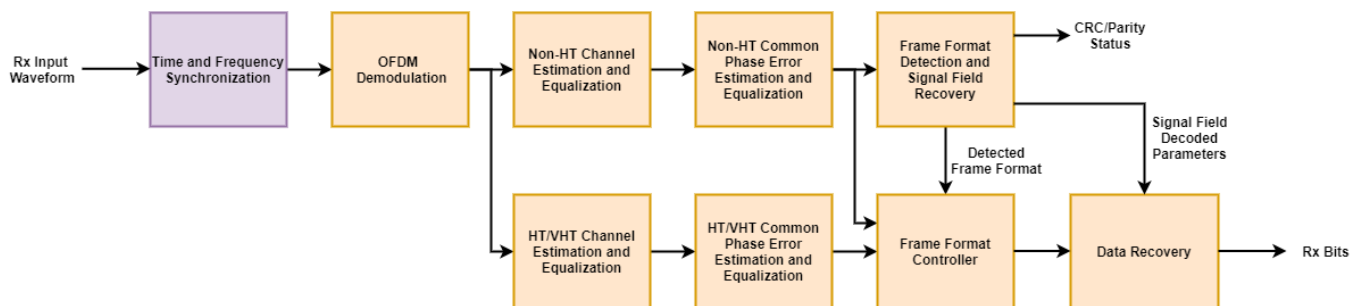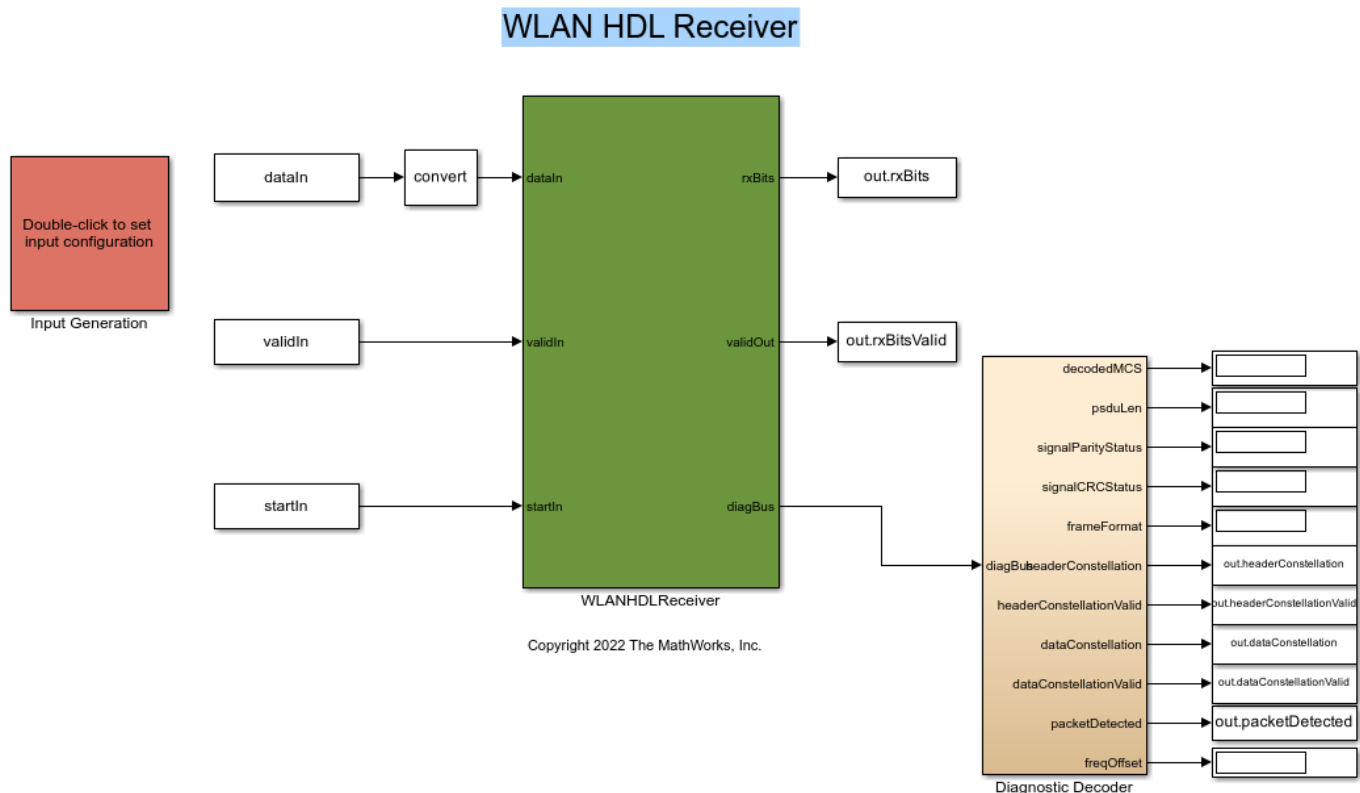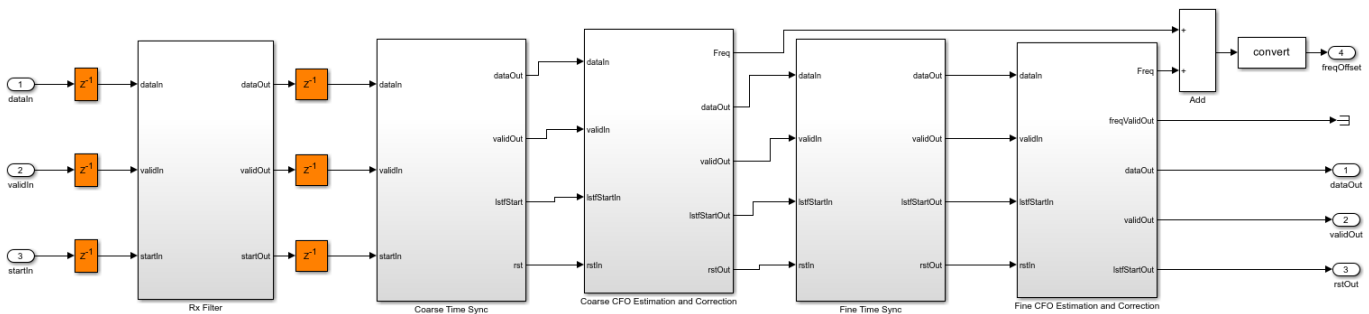modelname = 'wlanhdlReceiver';
open_system(modelname);
```


Time and Frequency Synchronization

The `Time and frequency synchronization` subsystem performs receiver filtering and coarse time and frequency estimation and corrections on the filtered signal. Then, the subsystem fine tunes the time and frequency estimation and corrections to remove any residual offsets. The `wlanhdlReceiverInit.m` file initializes filter coefficients.

Open the `WLANTimeAndFrequencySynchronization` subsystem to see the synchronization process.

`open_system([modelname '/WLANHDLReceiver/WLANTimeAndFrequencySynchronization'],'force');`

OFDM Demodulation

The OFDM Demodulator block converts time-domain signals to frequency-domain subcarriers. The block provides the flexibility to change the orthogonal frequency division multiplexing (OFDM) parameters **FFT length**, **Cyclic prefix length**, **Number of left guard subcarriers**, and **Number of right guard subcarriers** during the runtime. In this example, based on bandwidth option, the cyclic prefix (CP) length varies for different fields in the WLAN signal. For example, the first symbol of L-LTF uses a CP length of 32 or 64, the second symbol of the L-LTF uses a CP length of 0, and the remaining fields of the WLAN signal uses a CP length of 16 or 32 for 20 MHz and 40 MHz, respectively. In this example, the **FFT length** parameter is set to 64 for 20 MHz and 128 for 40 MHz and the **Number of left guard subcarriers** and **Number of right guard subcarriers** parameters are set to 4 and 3 for 20 MHz and 6 and 5 for 40 MHz, respectively.

`open_system([modelname '/WLANHDLReceiver/OFDMDemodulation']);`

The `OFDMDemodulationParameterCalculator` MATLAB function controls the OFDM Demodulator block parameters for different fields of the WLAN packet. The `OFDMDemodulationParameterCalculator` MATLAB function calculates the number of used subcarriers to determine the number of OFDM symbols in the WLAN packet.

Non-HT Channel Estimation and Equalization

The `NonHTChannelEstAndEqualize` subsystem is used for L-LTF channel estimation. The input is given to the OFDM Channel Estimator block. The OFDM Channel Estimator block implements least squares (LS) estimation for the channel estimation and performs averaging on the estimates from two L-LTF symbols of the WLAN signal. The OFDM Equalizer block uses the resultant averaged channel estimate to perform zero forcing (ZF) equalization on data.

```
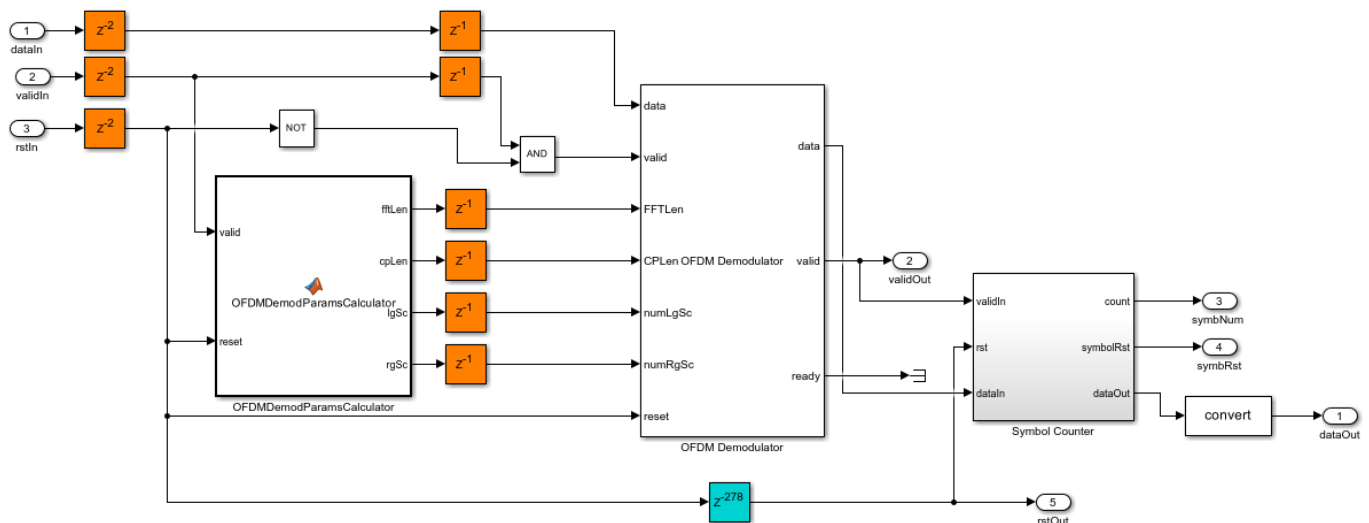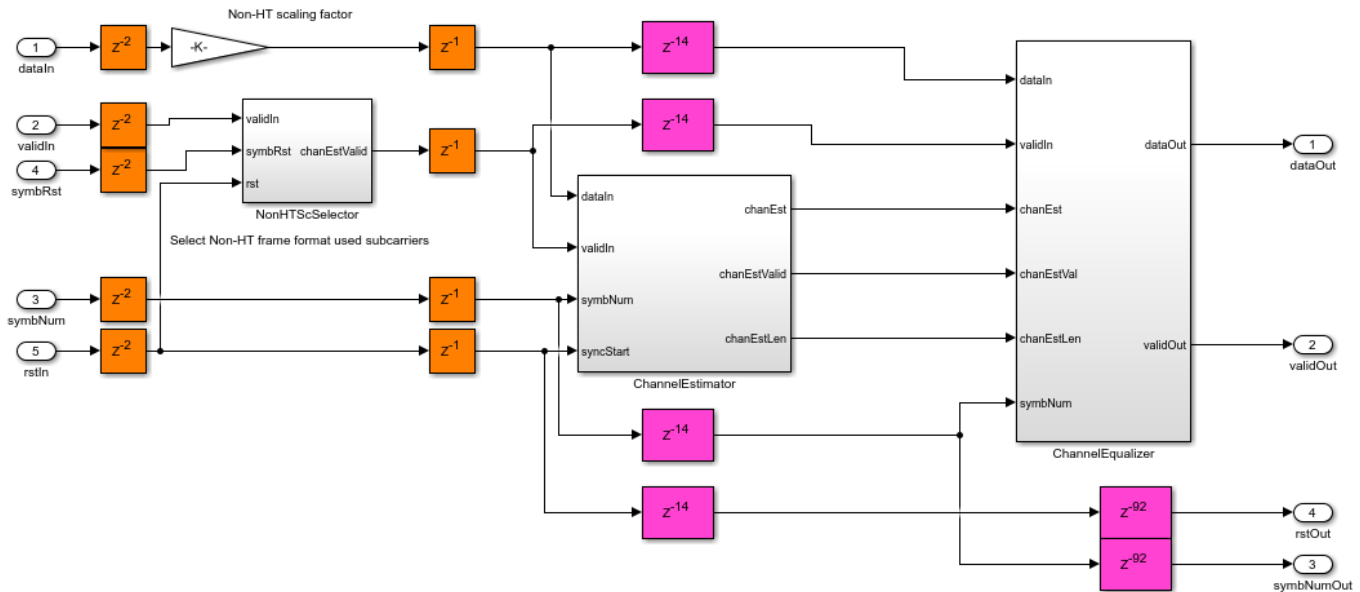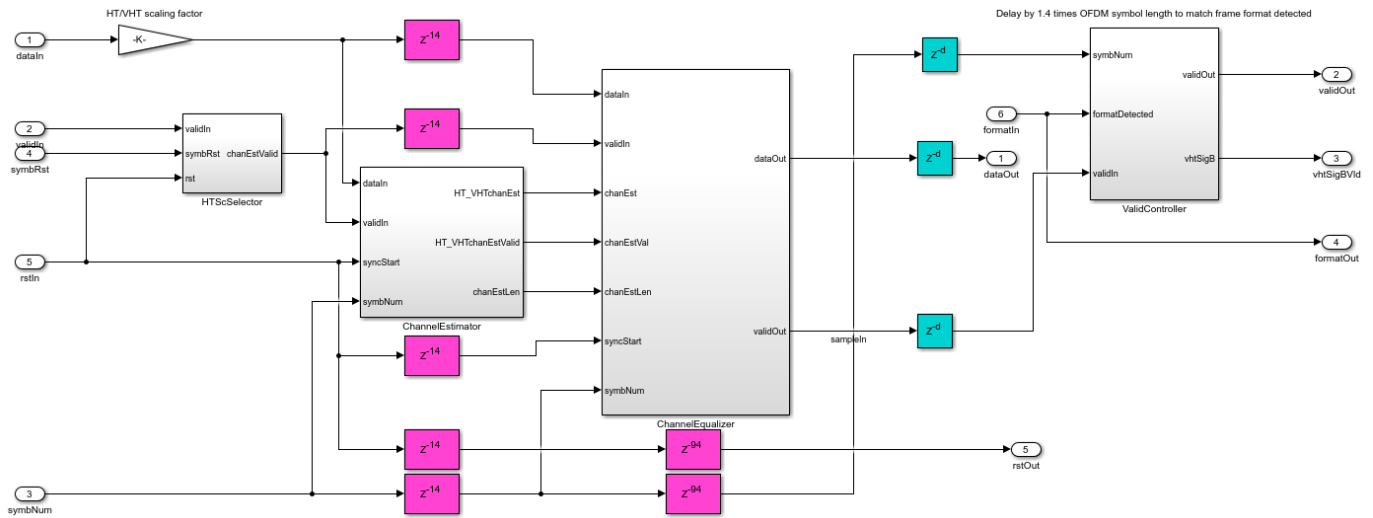open_system([modelname '/WLANHDLReceiver/NonHTChannelEstAndEqualize']);
```


HT or VHT Channel Estimation and Equalization

The `HTorVHTChannelEstAndEqualize` subsystem is similar to the `NonHTChannelEstAndEqualize` subsystem. For a SISO configuration, only one HT or VHT LTF exists, so averaging is disabled in the OFDM Channel Estimator block.

```
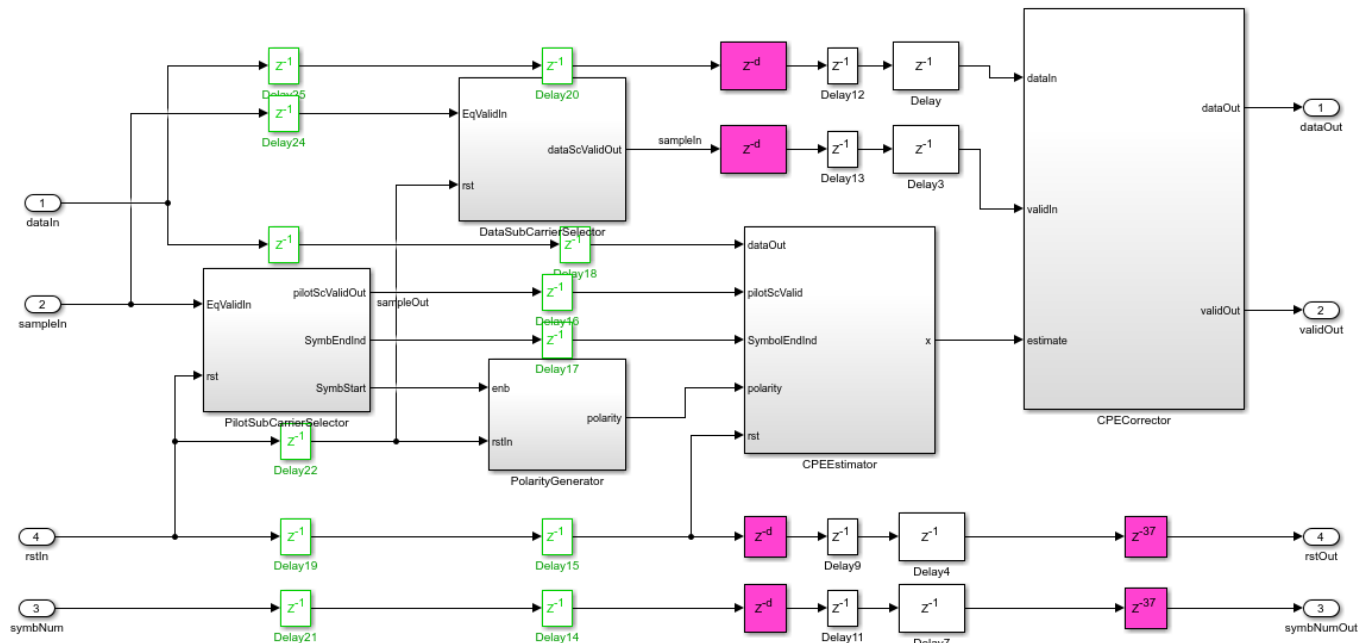open_system([modelname '/WLANHDLReceiver/HT_VHTChannelEstAndEqualize']);
```

Non-HT Common Phase Noise Estimation and Correction

The `NonHTCPEEstAndCorrection` subsystem estimates the common phase noise or residual frequency offset for the non-HT portion of the WLAN signal. CPE estimation requires references such as non-HT pilot positions, a non-HT pilot sequence, and pseudo-noise (PN) sequence as described in Equation 17-25 in [1]. The `wlanhdlRxinint.m` script initializes these known references and stores them in 1-D lookup tables in the subsystem. The `PolarityGenerator` subsystem gives the polarity of the pilots based on the symbol number. The reference pilots are multiplied with polarity for CPE estimation. The estimated CPE is averaged on all of the pilot subcarriers in an OFDM symbol and is used for the correction of data subcarriers of non-HT portion of the WLAN packet.

```
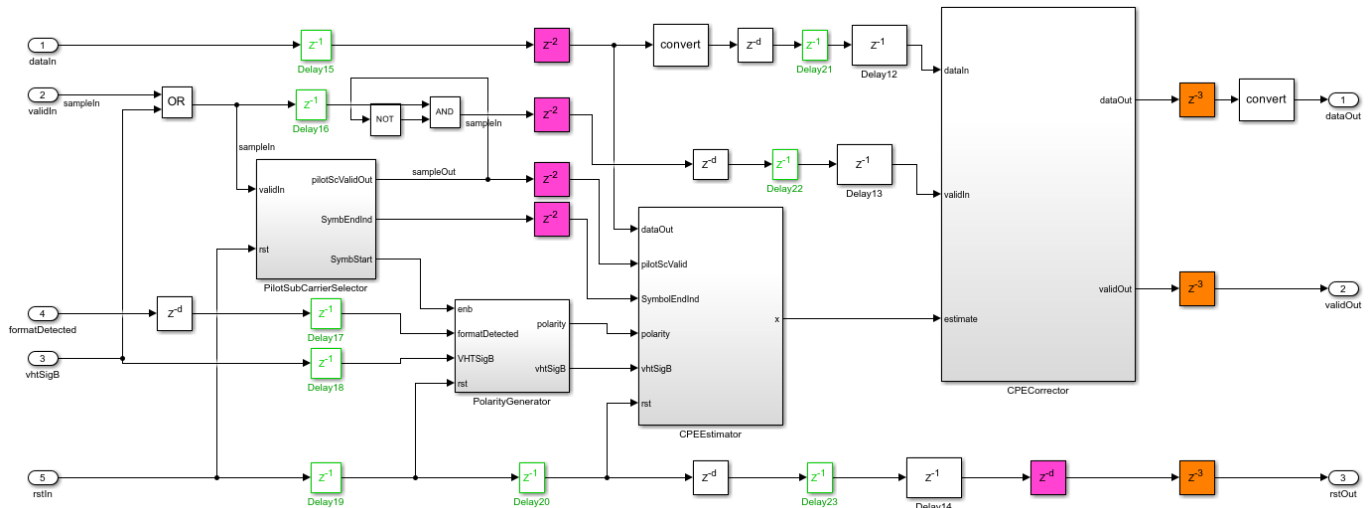open_system([modelname '/WLANHDLReceiver/NonHTCPEEstAndCorrection']);
```

HT or VHT Common Phase Noise Estimation and Correction

The `HTorVHTCPEEstAndCorrect` subsystem is similar to the `NonHTCPEEstAndCorrect` subsystem. This subsystem performs CPE estimation and correction using HT or VHT pilot positions and a HT or VHT pilot sequence.

```
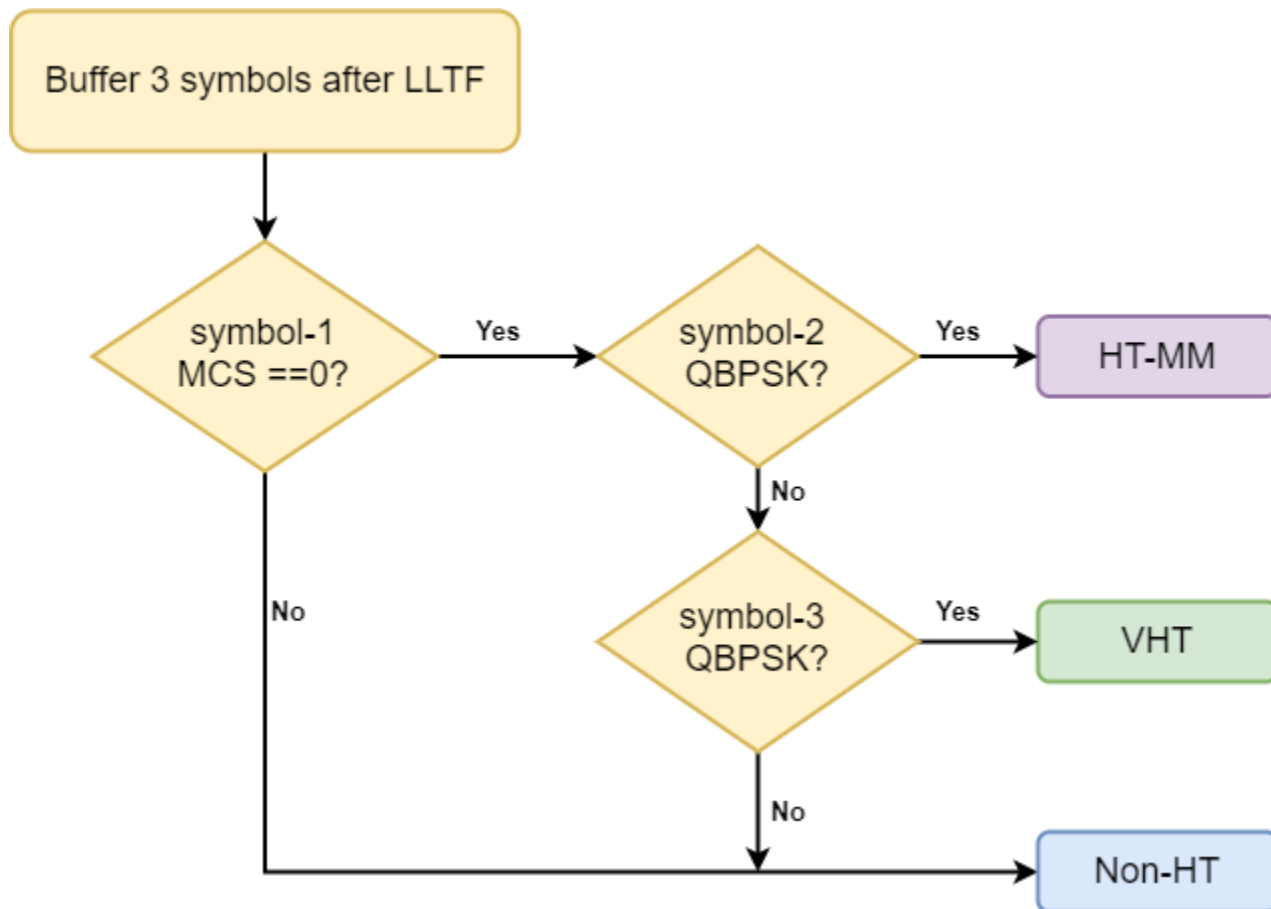open_system([modelname '/WLANHDLReceiver/HT_VHTCPEEstAndCorrection']);
```


Frame Format Identification and Controller

To recover the signal and data field information from a WLAN signal, you must use the WLAN receiver frame format. This example supports non-HT, HT, and VHT frame formats. The `FrameFormatDetector` subsystem detects the frame format as non-HT, HT, or VHT by buffering 3 OFDM symbols after L-LTFs, at the output of the `NonHTCPEEstAndCorrect` subsystem. This flow chart describes the frame format detection. For more information, see [2].

The `Signal Recovery` subsystem decodes the MCS from the first symbol L-SIG. If the MCS is not 0, the `FrameFormatDetector` subsystem detects the frame format as non-HT. If the MCS is 0, it checks the modulation scheme of OFDM symbol 2. If the modulation scheme of symbol 2 is QBPSK, the subsystem detects the format as HT. If the modulation scheme of symbol 2 is BPSK, it checks the modulation scheme of OFDM symbol 3. If the modulation scheme of symbol 3 is QBPSK, the subsystem detects the format as VHT. If the modulation scheme of symbol 3 is BPSK, the subsystem detects the format as non-HT.

If the `FrameFormatDetector` subsystem detects frame format as non-HT, then the remaining OFDM symbols, including OFDM symbols 2 and 3, are treated as L-Data. The `FrameFormatController` subsystem passes the output of the `NonHTCPEEstAndCorrect` subsystem to the `DataRecovery` subsystem to decode L-Data.

If the `FrameFormatDetector` subsystem detects the frame format as HT or VHT, the `FrameFormatController` subsystem passes the output of the `HT_VHTCPEEstAndCorrect` subsystem to the `DataRecovery` subsystem to recover HT-Data or VHT-Data.

Signal Recovery

The `SignalRecovery` subsystem recovers the header information to decode data bits from L-SIG, HT-SIG, and VHT-SIG fields. The output of the `NonHTCPEEstAndCorrect` subsystem corresponding to signal fields is streamed into the `SignalRecovery` subsystem. The Symbol Demodulator block performs BPSK and QBPSK soft symbol demodulation on signal fields in the WLAN packet. The channel decoding includes `Deinterleaver` subsystem and Viterbi Decoder block.

The `Deinterleaver` subsystem performs deinterleaving on the symbol demodulated data with a maximum block size of 48 and the number of columns as 16. The Viterbi Decoder block performs 1/2 rate viterbi decoding on deinterleaved data. For more information on the `Deinterleaver` subsystem, see "HDL Interleaver and Deinterleaver" on page 5-242.

L-SIG uses the parity to check the error in WLAN L-SIG field, whereas 8-bit cyclic redundancy check (CRC) is used to check the error in the WLAN HT-SIG 1 and 2 and VHT-SIG-B field. The General CRC Syndrome Detector HDL Optimized block is used for CRC error detection and `ParityCalculator` subsystem performs parity calculation. If the CRC checksum or parity fails, the signal field recovery returns the status of parity check or CRC (Pass or Fail).

open_system([modelname `'/WLANHDLReceiver/FrameFormatDetectionAndSignalRecovery/SignalBitRecovery`

Data Recovery

The `DataRecovery` subsystem uses the WLAN signal fields to decode data bits. The registers are used to store WLAN signal field information. These registers access the WLAN signal field information. The Symbol Demodulator block performs soft-bit BPSK, QPSK, 16-QAM, or 64-QAM symbol demodulation associated with the modulation type retrieved from the WLAN signal field information.

If the decoded WLAN channel coding type is BCC, the `Deinterleaver` and the `BCC Decoder` subsystems perform deinterleaving and Viterbi decoding on the symbol demodulated data. The `Deinterleaver` subsystem consists of different deinterleavers for non-HT and HT or VHT data. The deinterleaver for non-HT data is configured with a block size of 48 and number of columns as 16. The deinterleaver for HT or VHT data is configured with a block size of 52 and number of columns as 13 for 20 MHz and a block size of 216 and number of columns as 18 for 40 MHz, respectively. The `BCC Decoder` subsystem is equipped with Depuncturer and Viterbi Decoder blocks. Each code rate is assigned a predefined punctured vector pattern. Based on the code rate retrieved from the WLAN signal field information, the `BCC Decoder` subsystem performs depuncturing followed by Viterbi decoding.

If the decoded WLAN channel coding is LDPC, the `LDPCToneDemapping` subsystem in the `DataSelector` subsystem performs the LDPC tone demapping for VHT data before symbol demodulation. The `LDPC Decoder` subsystem in this example decodes one LDPC codeword data. The `LDPCParametersCalculator` subsystem computes the required LDPC parameters such as the LDPC length, the number of payload bits, the number of shortened bits, and the number of punctured

bits for the codeword. The `LDPC Codeword Formation` subsystem depunctures the symbol demodulated data to form one LDPC codeword using the LDPC parameters. The WLAN LDPC Decoder block decodes the depunctured data based on the codeword block length index and the code rate index.

The decoded bits are streamed through the `Descrambler` subsystem.

```
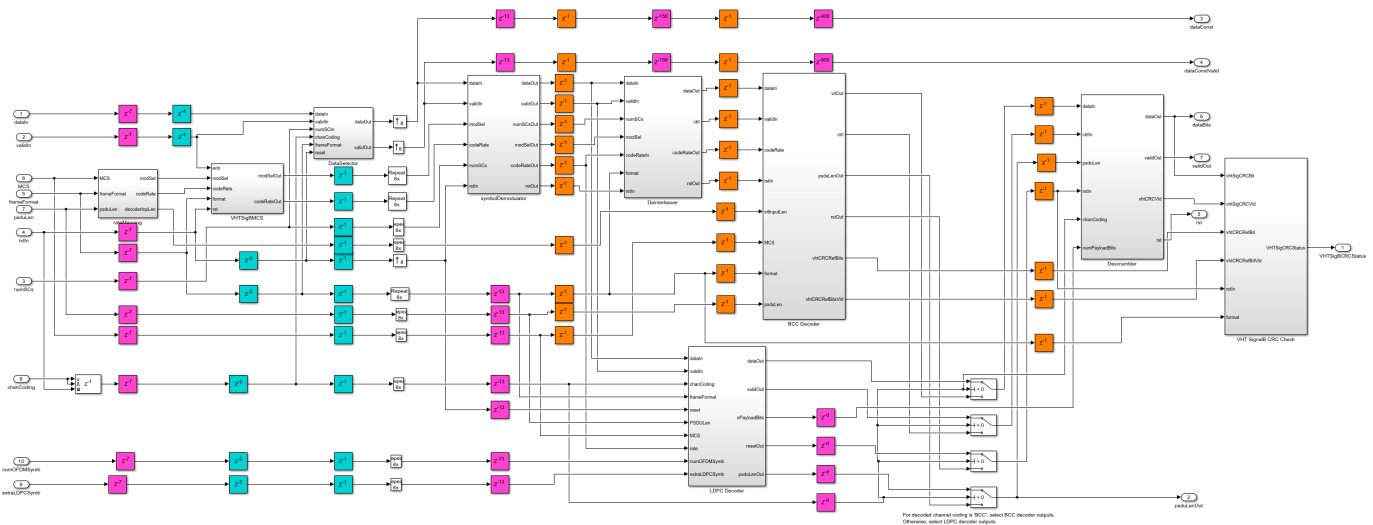open_system([modelname '/WLANHDLReceiver/DataRecovery']);
```


File structure

This example uses one Simulink models and three MATLAB files.

- `wlanhdlReceiver.slx` — Open the top-level OFDM receiver Simulink model.
- `wlanhdlReceiverInit.m` — This script is initialized in the `InitFcn` callback of the `whdlhdlReceiver.slx`. This script uses `wlanWaveformGenerator.m` to generate the input waveform to the example.
- `wlanhdlRxParameters.m` — Generate the input parameters according to the Standard IEEE 802.11-2016 to run the `wlanhdlReceiver.slx` model. The parameters correspond to non-HT, HT, and VHT frame formats for 20 MHz and 40 MHz bandwidth options.
- `wlanhdlMATLABRxReference.m` — Implement a MATLAB floating-point equivalent WLAN receiver using functions from the WLAN Toolbox.

Model Inputs and Outputs

The inputs and outputs to the example model are described below

- **dataIn** — Input data, specified as a complex signed 16-bit signal sampled at 20 Msps for 20 MHz and 40 Msps for 40 MHz bandwidth options.
- **validIn** — Control signal to validate the **dataIn**, specified as a Boolean scalar.
- **startIn** — Control signal to reset the receiver, specified as a Boolean scalar.
- **dataOut** — Decoded output data bits, returned as a bits.
- **validOut** — Control signal to validate the **dataOut** port, returned as a Boolean scalar.

- **diagBus** — Status signal with diagnostic outputs, returned as a bus signal.

Verification and Results

This example model accepts a WLAN transmit waveform as an input along with valid and start signals. The model returns decoded information bits as an output along with a valid signal. The `wlanhdlReceiverInit.m` script provides the input to the model. For the demonstration of the example, the `wlanWaveformGenerator.m` function in the script generates the VHT mode, 20 MHz BCC channel coded frame, which is passed through the TGac channel with a delay profile of Model A. The additive white Gaussian noise (AWGN) at a 30 dB signal-to-noise ratio (SNR) is added to the transmit waveform with other channel impairments such as 20 kHz CFO and a timing offset of 256. To verify the example for LDPC channel coding, set the **Channel coding** parameter to LDPC on the `Input Generation` subsystem, and run the example.

```
fprintf('\n Simulating WLAN HDL receiver \n');
out = sim(modelname);
close_system(modelname,0);
fprintf('\n HDL simulation complete. Data decoded. \n');
```

```
 Simulating WLAN HDL receiver

 HDL simulation complete. Data decoded.
```

Verify the outputs of this example using WLAN Toolbox™ functions. Specify the same input waveform to the Simulink model and to the MATLAB equivalent receiver. Compare the outputs to validate the example.

```
fprintf('\n Comparing WLAN MATLAB reference receiver \n')
wlanhdlMATLABRxReference;
fprintf('\n MATLAB simulation complete. \n');

simOut = squeeze(out.rxBits(out.rxBitsValid));
errSig = (bitxor(logical(psdu),simOut));
err = sum(errSig);

hConstData = out.headerConstellation(out.headerConstellationValid);
figure;
plot(hConstData,'o');
xlabel('In-Phase'); ylabel('Quadrature')
title('Equalized Signal Field Constellation');
m = double(max(abs([real(hConstData(:)); imag(hConstData(:))]))) * 1.1);
axis([-m m -m m]);

dConstData = out.dataConstellation(out.dataConstellationValid);
figure;
plot(dConstData(1:end-NSc*4),'o'); % Remove last 4 symbols corresponding to idle time
xlabel('In-Phase'); ylabel('Quadrature')
title('Equalized Data Field Constellation');
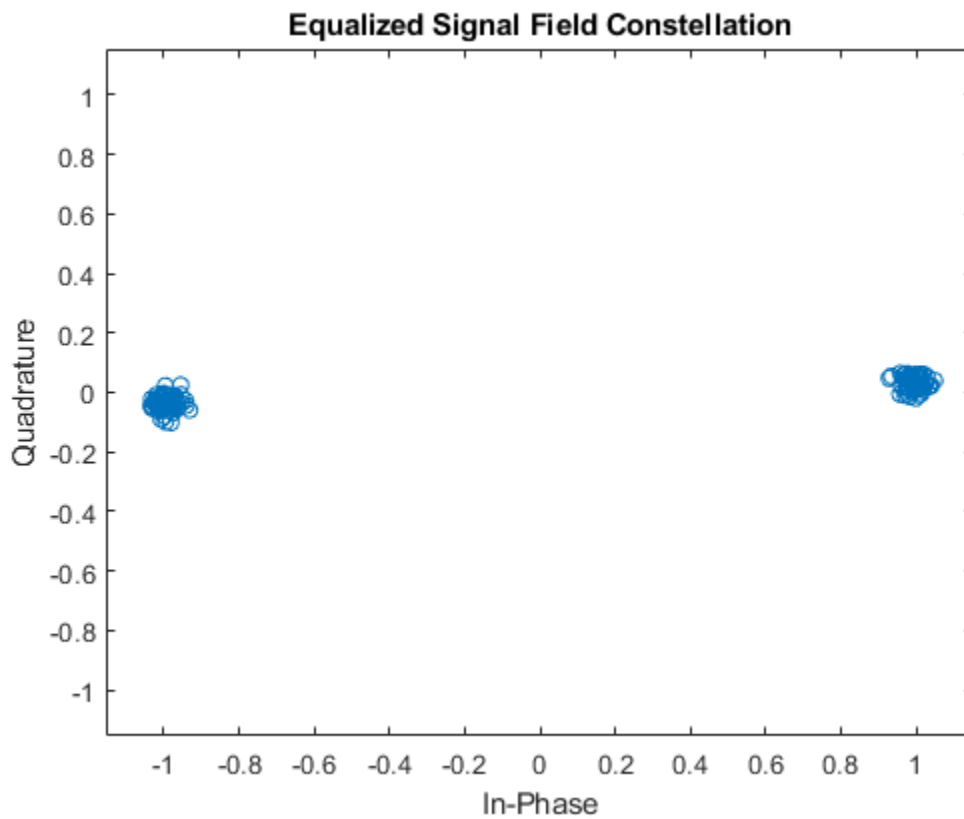m = double(max(abs([real(dConstData(:)); imag(dConstData(:))]))) * 1.1);
axis([-m m -m m]);

figure;
plot(errSig);
xlabel('Sample Number');
ylabel('Error Magnitude');
legend('Error')
```

```matlab
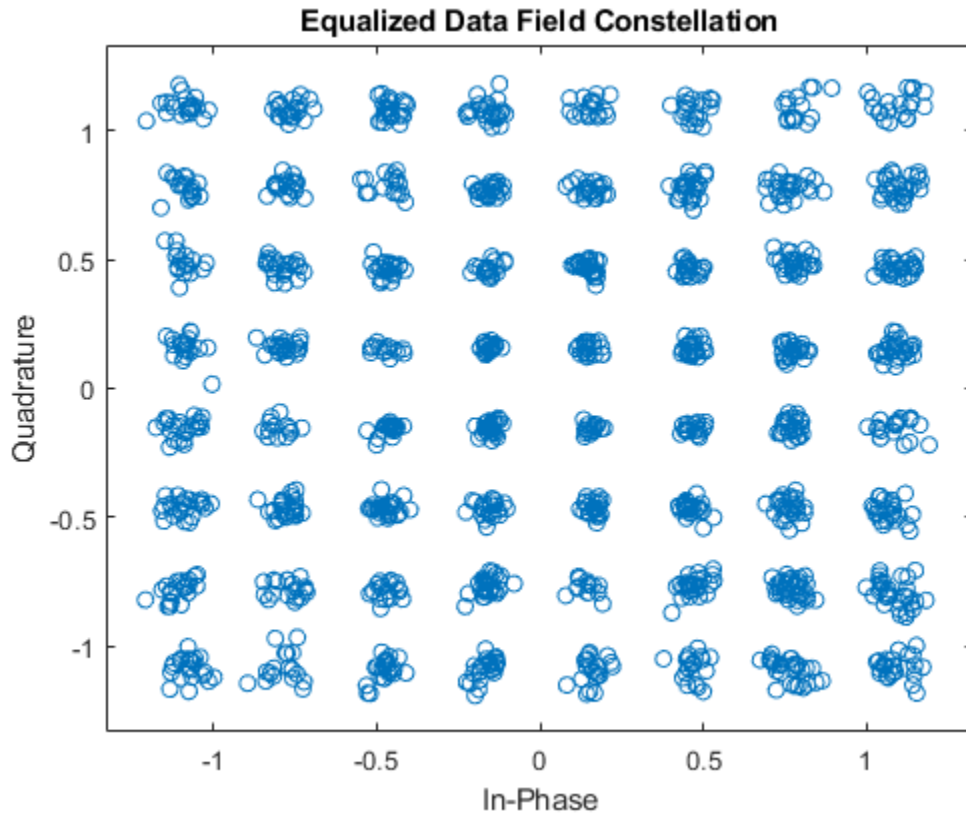title('Error Magnitude Between Simulink and MATLAB WLAN Receiver Output');

if err == 0
    fprintf('\n Simulink and MATLAB outputs match \n');
else
    fprintf('\n Simulink and MATLAB outputs do not match  \n');
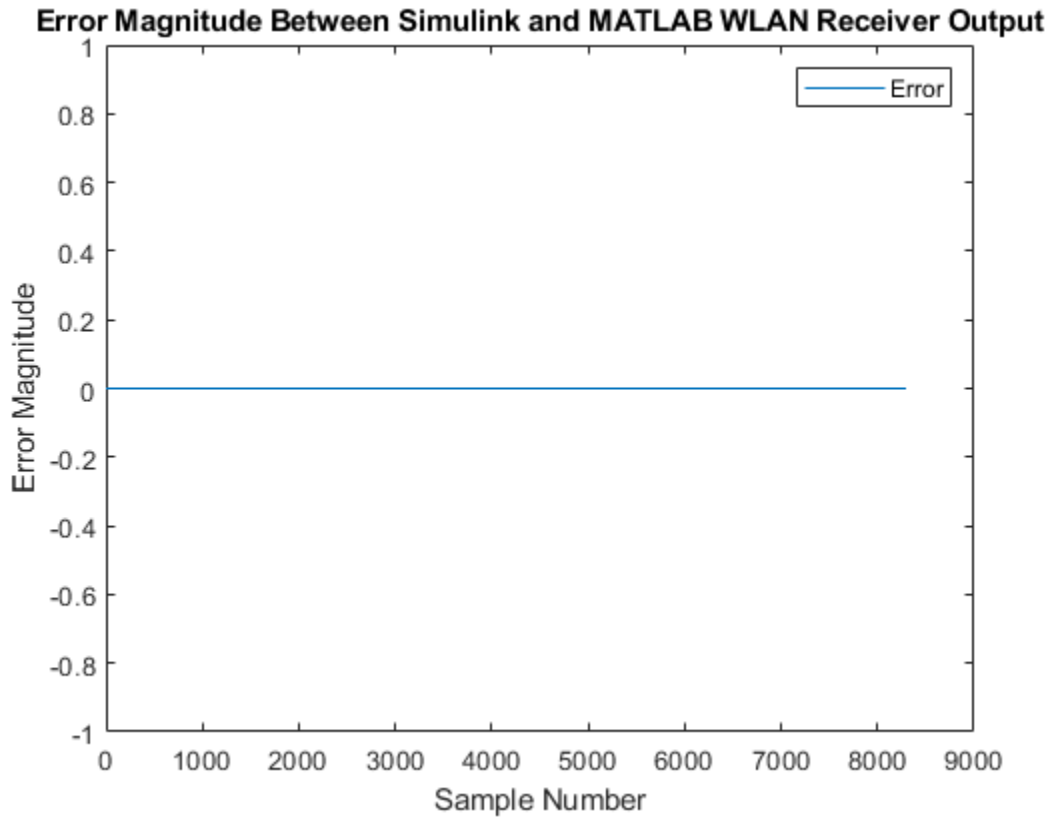end
```

Comparing WLAN MATLAB reference receiver

MATLAB simulation complete.

Simulink and MATLAB outputs match

Equalized Signal Field Constellation

Equalized Data Field Constellation

Error Magnitude Between Simulink and MATLAB WLAN Receiver Output

HDL Code Generation and Implementation Results

To generate the HDL code for this example, you must have the HDL Coder™ product. To generate HDL code and an HDL testbench for `WLANHDLReceiver` subsystem, use the `makehdl` and `makehdltb` commands. The resulting HDL code was synthesized for a Xilinx® Zynq® Ultrascale+ RFSoC ZCU111 evaluation board. The table shows the post place and route resource utilization results. The design meets timing with a clock frequency of 308 MHz for 20 MHz and 40 MHz bandwidth options.

```
F = table(...
    categorical({'CLB LUT'; 'CLB Registers'; 'RAMB36'; 'RAMB18';...
    'DSP48'}),...
    categorical({'88,475'; '87,697'; '284'; '12'; '223'}),...
    categorical({'90,820'; '89,725'; '288'; '16'; '251'}),...
    'VariableNames',...
    {'Resources','Usage for 20 MHz', 'Usage for 40 MHz'});

disp(F);
```

Resources	Usage for 20 MHz	Usage for 40 MHz
CLB LUT	88,475	90,820
CLB Registers	87,697	89,725
RAMB36	284	288
RAMB18	12	16

| DSP48 | 223 | 251 |

References

1 IEEE 802.11-2016 - IEEE Standard for Information technology--Telecommunications and information exchange between systems Local and metropolitan area networks--Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.

2 Nanda Kishore Chavali, 'System and method for detecting a frame format' (March 2013), US20130077718A1.

See Also

Blocks
Viterbi Decoder | Depuncturer | OFDM Channel Estimator | OFDM Demodulator | OFDM Equalizer

Related Examples

- "HDL Interleaver and Deinterleaver" on page 5-242
- "WLAN HDL Time and Frequency Synchronization" on page 5-217

Deploy WLAN HDL Reference Applications on FPGAs and SoCs

This section contains the list of examples that show how to deploy WLAN Wireless HDL Toolbox™ reference applications on FPGAs and SoCs.

- "WLAN Receiver Using Analog Devices AD9361/AD9364" (Communications Toolbox Support Package for Xilinx Zynq-Based Radio): Deploy a WLAN HDL receiver system to retrieve the signal and data field information from a WLAN signal for the 20 MHz channel bandwidth.

This example reuses the WLAN Simulink® models to generate HDL for the FPGA logic. They use hardware-software co-design modeling techniques and hardware support packages to add all the software modeling and interfacing required to implement the algorithm in real-time on hardware.

See Also

Related Examples
- "HDL Implementation of WLAN Receiver" on page 5-227
- "WLAN HDL Time and Frequency Synchronization" on page 5-217

HDL Interleaver and Deinterleaver

This example shows how to design block interleaver and block deinterleaver blocks and implement interleaving and deinterleaving in a communication system using these blocks.

Interleaving

Audio processing and radio transmission applications are often affected due to burst noise. Burst noise degrades the performance of forward error correction (FEC) codes. This degradation of performance results in the form of errors in the decoded data. Interleaving is a technique that spreads out the continuous burst of errors and improves data decoding using FEC codes. Interleaving is part of wireless standards such as digital video broadcasting - satellite-second generation (DVB-S2), wireless local area network (WLAN 802.11), and long term evolution (LTE). This block diagram shows the overview of a communication system with interleaver and deinterleaver.

Communication System
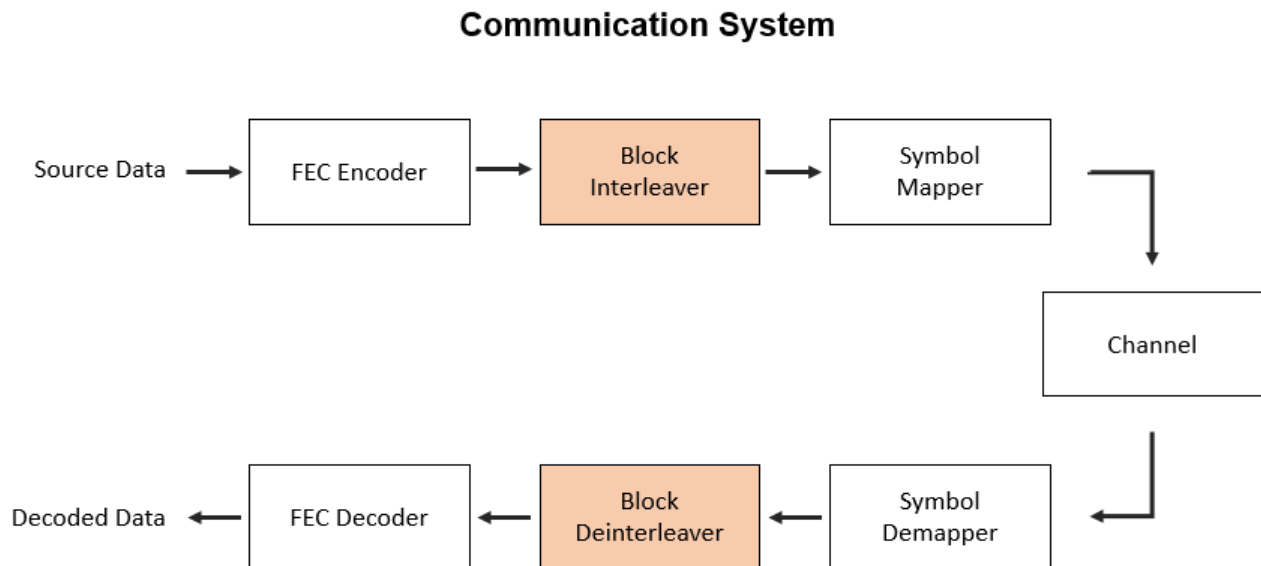
An interleaver writes the input data in a row-wise format to the memory and reads the output data in a column-wise format from the memory. A deinterleaver operates in the reverse manner by writing the input data in a column-wise format to the memory and reading the output data in a row-wise format from the memory. The number of rows and columns decide the extent of interleaving. This figure shows the working of a block interleaver and block deinterleaver, each with four rows and four columns.

Interleaver

Memory (RAM)

write ⟶

read

d1	d2	d3	d4
d5	d6	d7	d8
d9	d10	d11	d12
d13	d14	d15	d16

Input
d1, d2, d3, d4, d5 ...

Output
d1, d5, d9, d13, d2 ...

Deinterleaver

Memory (RAM)

read ⟶

write

d1	d2	d3	d4
d5	d6	d7	d8
d9	d10	d11	d12
d13	d14	d15	d16

Input
d1, d5, d9, d13, d2 ...

Output
d1, d2, d3, d4, d5 ...

HDL Interleaver Model

This section provides the overview of a communication system implemented using the `WHDLInterleaverModel.slx` model, which contains interleaver and deinterleaver blocks. The input data to the model, **dataIn**, is convolutionally encoded using the Convolutional Encoder block. The encoded data is then interleaved by the Interleaver Block that is in the Interleaver subsystem. Burst noise is added to the interleaved data by performing a XOR operation of the data with the burst noise. The corrupted data is given as an input to the Deinterleaver subsystem where the Deinterleaver Block spreads out the burst errors in the data. The Viterbi Decoder block decodes the deinterleaved data and outputs the final decoded data. The model contains additional subsystems that are used to synchronize the blocks in the model. A constant block with the `interleave` variable is provided in the Interleaver and Deinterleaver subsystems of the model. You can set or reset the `interleave` variable to enable or disable interleaving.

HDL Interleaver Model

This model shows encoding of input data, corrupting the encoded data with burst noise, and decoding the corrupted data. The encoding chain consists of convolutional encoder followed by interleaver.
Burst noise is introduced at the output of interleaver. The decoding chain reverses the operations of the encoding chain and performs deinterleaving and hard-decision viterbi decoding, which compensate for interleaving and convolutional encoding respectively.
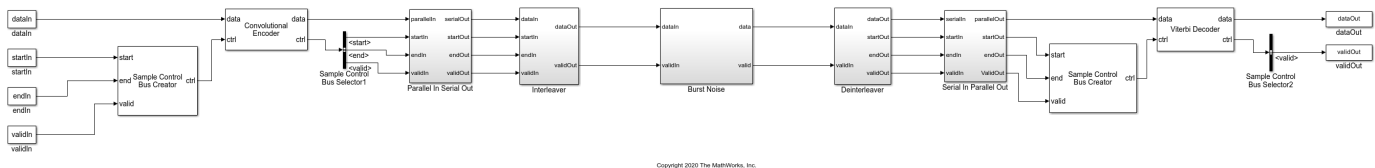
Copyright 2020 The MathWorks, Inc.

Port Description

This section explains the input and output ports of the Interleaver Block that is in the Interleaver subsystem of the `WHDLInterleaverModel.slx` model.
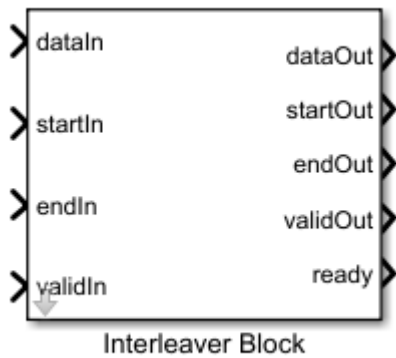
Input Ports:

- **dataIn** — Input data to be interleaved. As the block performs serial processing, **dataIn** is specified as a scalar. The block supports `double`, `single`, `Boolean`, `integer`, and `fixed point` data types.

- **startIn** — Start signal of the input data block, specified as a Boolean scalar.

- **endIn** — End signal of the input data block, specified as a Boolean scalar.

- **validIn** — Valid signal of the input data block, specified as a Boolean scalar.

Output Ports:

- **dataOut** — Interleaved output data returned as a scalar. The output data type is same as that of the **dataIn** port.

- **startOut** — Start signal of the output data block, returned as a Boolean scalar.

- **endOut** — End signal of the output data block, returned as a Boolean scalar.

- **validOut** — Valid signal of the output data block, returned as a Boolean scalar.

- **ready** — Ready output signal used for external interfacing, returned as a Boolean scalar. The interleaver accepts one new block of input data while still processing an earlier data block. If more than one block of data is given as input while processing an earlier data block, the ready signal deasserts, indicating that the interleaver is not ready to accept new data.

The input and output ports of the Deinterleaver Block, which is in the Deinterleaver subsystem of the `WHDLInterleaverModel.slx` model, are the same as that of the Interleaver Block.

Parameters

This figure shows the block mask of the Interleaver Block. You can use this block as an interleaver or a deinterleaver by modifying a parameter selection on the block mask.
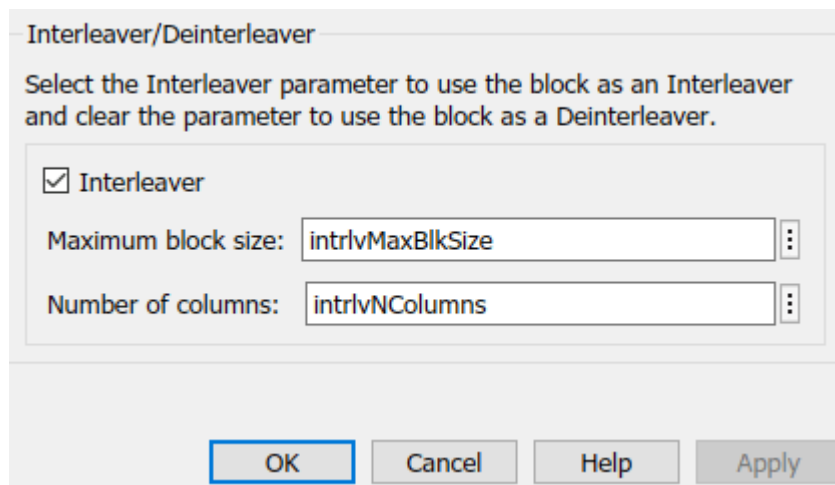
The Interleaver Block supports these parameters:

- **Interleaver** — Nontunable mask parameter. Select the **Interleaver** parameter to use the block as an Interleaver. Clear this parameter to use the block as a Deinterleaver.
- **Maximum block size** — Nontunable mask parameter. This parameter specifies the maximum supported block size. This value sets the size of the RAM used inside the block. The minimum value of this parameter is 4.
- **Number of columns** — Nontunable mask parameter. This parameter specifies the number of columns. The minimum number of columns is 2 and must be a factor of **Maximum block size**.

The **block size** of the interleaver is tunable, meaning it can be adjusted during the simulation by using the input **start**, **end**, and **valid** control signals. The block size is the number of input valid samples from the start to the end of the data block. The block size must be an integer multiple of **Number of columns**. The minimum value of the block size is **Number of columns** x 2 and the maximum value is **Maximum block size**.

For example, if you specify the **Maximum block size** parameter as 30 and the **Number of columns** parameter as 5, the possible values of the tunable block size during the simulation are 10, 15, 20, 25, and 30. The block automatically calculates the number of rows, which varies with the block size.

Architecture

This section explains the architecture of the Interleaver Block. The Interleaver Block accepts the input data in the form of data blocks along with control signals. The Interleaver Block interleaves each data block independently. This figure shows the architecture of the Interleaver Block.
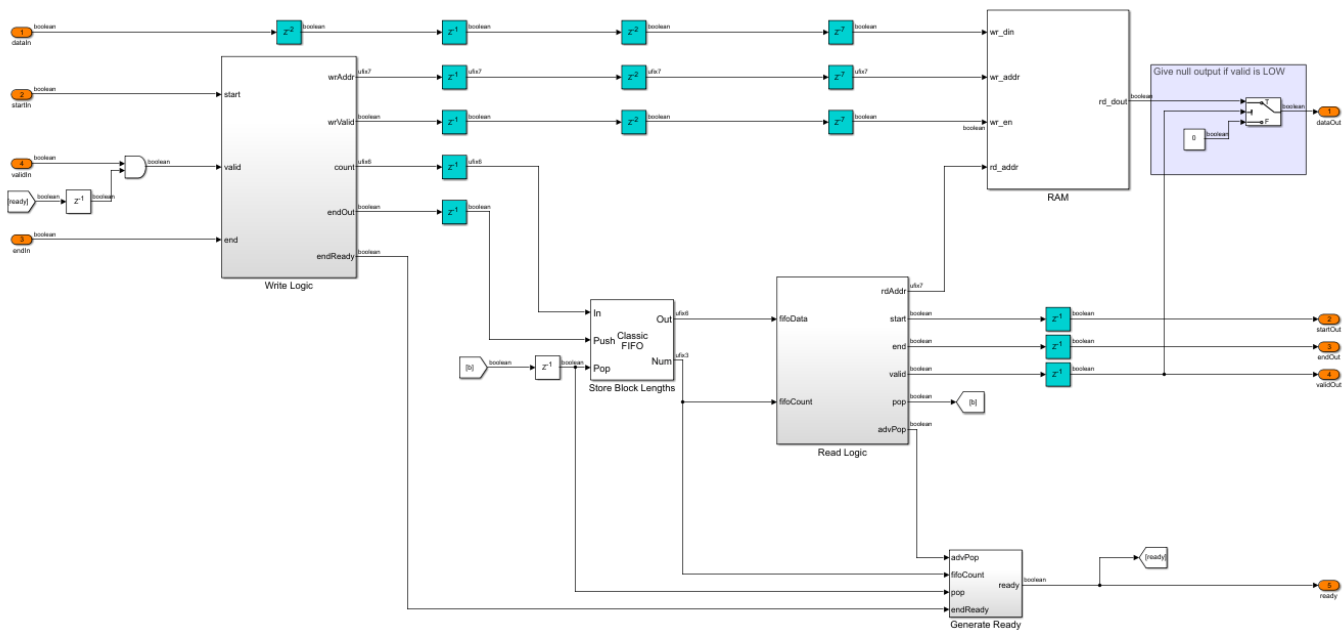
The Interleaver Block contains three subsystems and two blocks:

- Write Logic — This subsystem accepts input control signals and generates appropriate write valid and write address signals for writing the data into the RAM.

- Store Block Lengths — This is a FIFO block that stores the input data block lengths during the simulation.

- Read Logic — This subsystem performs the actual interleaving operation and generates the read address to read out the data from the RAM.

- Generate Ready — This subsystem generates the ready output signal for interfacing with other blocks.

- RAM — This block stores the input data and outputs interleaved data based on the input read address.

Only the Generate Read Address subsystem in the Read Logic subsystem of the Interleaver Block and Deinterleaver Block differs in its functionality, remaining other subsystems are same.

Model Simulation

Run the `runWHDLInterleaverModel.m` script to simulate the `WHDLInterleaverModel.slx` model. The script initializes, simulates, and validates the outputs of the model. For optimum results, tune the interleaving parameters in the script based on the burst noise parameters.

Disable interleaving and then run the script to simulate the model, validate the outputs, and display errors.

```
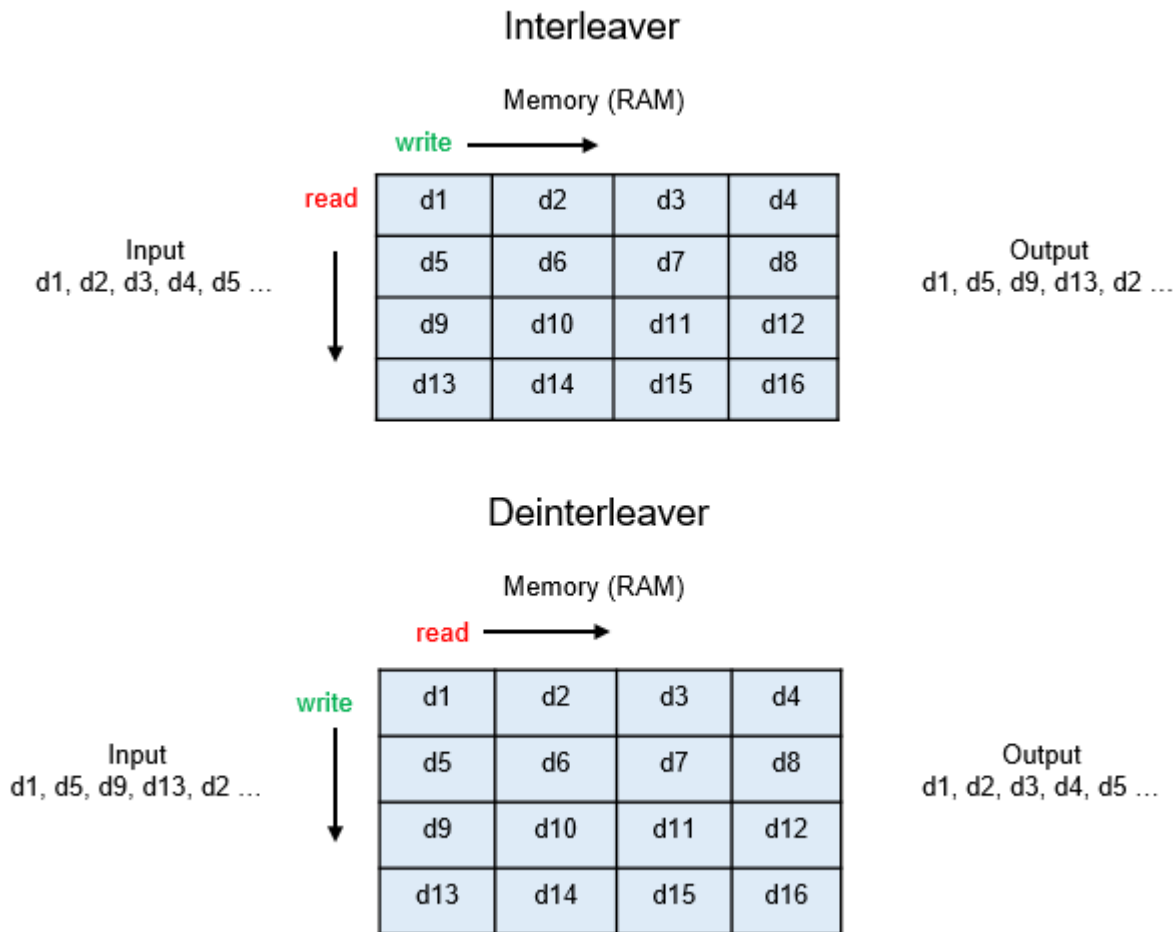errorRateWithoutInterleaving =

    0.1354
```

Enable interleaving and then run the script to simulate the model, validate the outputs, and display errors.

```
errorRateWithInterleaving =

    0.0125
```

When you enable interleaving, the error rate is less than the error rate when you disable interleaving. This result occurs because interleaving improves the performance of the Viterbi Decoder block by spreading out the burst errors.

HDL Code Generation and Implementation Results

To check and generate the HDL code referenced in this example, you must have the HDL Coder™ product. To generate the HDL code, enter this command at the MATLAB command prompt.

```
>> makehdl('WHDLInterleaverModel/Deinterleaver/Deinterleaver Block')
```

The resource utilization and frequency of operation values vary with the input data type, the maximum block size, and the number of columns. HDL code is synthesized for the Xilinx® Zynq®-7000 ZC706 evaluation board for the Deinterleaver Block in the Deinterleaver subsystem with `fixdt(1,16,14)` input, a maximum block size of 360, and 30 columns. This table shows the post place and route resource utilization. The maximum frequency of operation is 292 MHz. Similar results are obtained for the Interleaver Block in the Interleaver subsystem.

```
     Resources         Usage
  _____     _____

  Slice Registers       293
  Slice LUT             271
  RAMB18                1
```

HDL Implementation of Digital Predistorter with LMS Coefficient Estimation

This example shows how to implement a digital predistorter (DPD) with least mean squares (LMS) based coefficient estimation, which is optimized for HDL code generation and hardware implementation. This example expands on the "HDL Implementation of Digital Predistorter" on page 4-55 example for computing DPD coefficients on an FPGA instead of a processor.

This example contains two Simulink® models.

- `HDLDPDwithLMSCoeffExample` — Use this model to perform predistortion using an adaptive LMS coefficient estimator that estimates coefficients dynamically. This model replaces the C/C++ code generatable `RPEM Coeff Estimation` subsystem of the HDL Implementation of Digital Predistorter example with an HDL-compatible `LMS Coefficient Estimator` subsystem.
- `HDLDPDwithPreCalculatedCoeffExample` — Use this model to perform predistortion using pre-calculated coefficients.

This example supports the hardware-friendly interface for the Xilinx® Zynq® UltraScale™ RFSoC ZCU111 evaluation board, which uses RF data converter. These example models support `Normal` and `Accelerator` simulation modes. For more information about DPD, see Adaptive DPD Design.

Run this command to open the high-level architecture of the HDL DPD with LMS coefficient estimation.

```
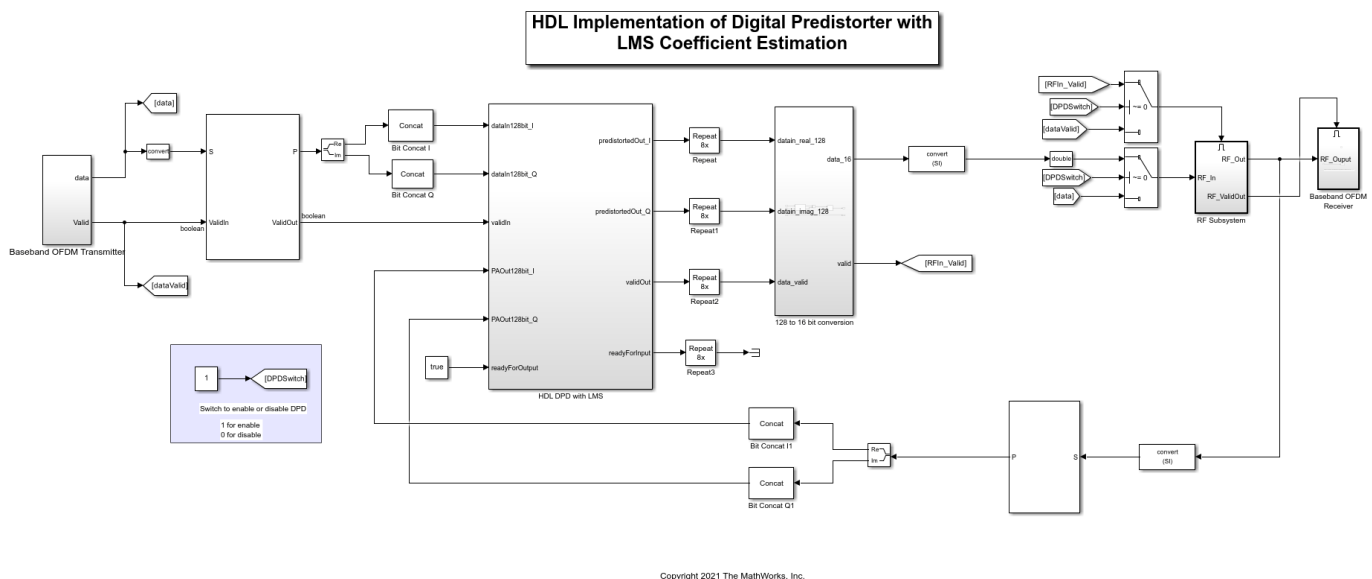modelname = 'HDLDPDwithLMSCoeffExample';
open_system(modelname);
```


Copyright 2021 The MathWorks, Inc.

Model Architecture

The `Baseband OFDM Transmitter` subsystem generates a 16 bit complex baseband orthogonal frequency division multiplexed (OFDM) signal at a sample rate of 15.36 MHz. A radio frequency system-on-chip (RFSoC) device has an RF data converter connected to the programmable logic. The

RF data converter supports a 128 bit in-phase (I) and quadrature-phase (Q) word. To generate a 128 bit I and Q word, the complex 16 bit baseband OFDM signal is grouped into 8 samples and converted into a 128 bit I and Q word. Deserializer1D, Complex to Real-Imag, and Bit Concat Simulink® blocks perform this operation. The output data sample rate of Bit Concat blocks is 1.92 MHz (15.36/8 MHz).

The `DPD_LMS` subsystem of the `HDL DPD with LMS` subsystem accepts the 128 bit I and Q word that is generated from the `Baseband OFDM Transmitter` subsystem and the 128 bit I and Q word that is generated from the `RF Subsystem` subsystem. The `HDL DPD with LMS` subsystem performs DPD using the LMS-based coefficient estimation and returns a 128 bit predistorted I and Q word. The Upsample block and the `128 to 16 bit converter` MATLAB® function convert the 128 bit predistorted I and Q word to 16 bit complex predistorted data. The output data sample rate of the `128 to 16 Bit Converter` function is 15.36 MHz (1.92 x 8 MHz).

When you enable the **DPDSwitch** in the model, the `RF Subsystem` subsystem accepts 16 bit complex predistorted data from the `128 to 16 Bit Converter` function. Otherwise, the subsystem accepts data from the `Baseband OFDM Transmitter` subsystem as input I and Q samples. The power amplifier (PA) accepts these I/Q samples that are upsampled to 2.4 GHz. The PA is preloaded with the coefficient matrix based on the standard-compliant LTE signal with a sample rate of 15.36 MHz. These PA coefficients are stored in a MAT-file, and these values are loaded while initializing the example.

In the other path, the data is passed through a low noise amplifier (LNA) and is down-converted before providing to the `DPD_LMS` subsystem in the `HDL DPD with LMS` subsystem. The `Baseband OFDM Receiver` subsystem collects the down-converted data and provides it as an input to the `OFDMRx` function.

In this example, the **readyForInput** output port is terminated. You can use this port if the previous subsystem driving the `HDL DPD with LMS` subsystem has the **readyForOutput** as the input control signal. Additionally, the **readyForOutput** input port is `true` because the subsequent `RF Subsystem` subsystem accepts input data at each time step. You can use this port if the subsequent subsystem after the `HDL DPD with LMS` subsystem has the **readyForInput** as the output control signal. The **validOut** of the `HDL DPD with LMS` is high (1) when the **readyForOutput** and the **validOut** of the `DPD_LMS` subsystem are high (1).

For more information about the `Baseband OFDM Transmitter`, `RF Subsystem`, and `Baseband OFDM Receiver` subsystems, see the "HDL Implementation of Digital Predistorter" on page 4-55 example.

Open the `HDL DPD with LMS` subsystem.

```
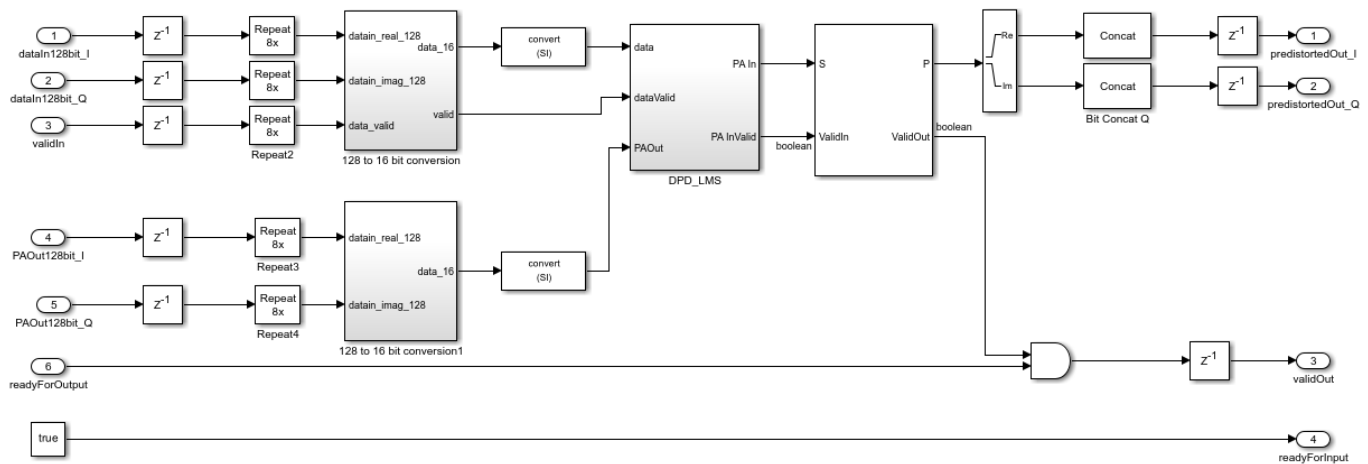load_system(modelname);
open_system([modelname '/HDL DPD with LMS']);
```

HDL DPD with LMS Coefficients Estimation

The `HDL DPD with LMS` subsystem performs digital predistortion in these stages.

1. Convert 128 Bit I and Q Input Word to 16 Bit Complex Data

The `HDL DPD with LMS` subsystem accepts 128 bit I and Q input words that are generated from the `Baseband OFDM Transmitter` and `RF Subsystem` subsystems. Because the `Digital Predistorter` and the `LMS Coefficient Estimator` subsystems operate on 16 bit complex inputs, the 128 bit I and Q input word data is reconverted to 16 bit complex data using the Repeat block and the `128 to 16 Bit Converter` MATLAB function. The output data sample rate of the `128 to 16 Bit Converter` function is 15.36 MHz (1.92 x 8 MHz).

2. Perform Digital Predistortion with LMS Coefficient Estimation on 16 Bit Complex Data

To perform digital predistortion with LMS coefficients, use the Digital Predistorter and Least Mean Square Coefficient Estimator subsystems. The next couple of sections give a detailed explanation about these subsystems.

3. Convert 16 Bit Complex Data to 128 Bit I and Q Output Word

To generate a 128 bit I and Q output word, a complex 16 bit predistorted signal is grouped into 8 samples and converted into a 128 bit I and Q output word. This operation is performed using the Deserializer1D, Complex to Real-Imag, and Bit Concat Simulink blocks. The output sample rate of data after using the Bit Concat blocks is 1.92 MHz (15.36/8 MHz).

Digital Predistorter

The `Digital Predistorter` subsystem distorts the 16 bit complex input data using the coefficients that are estimated by the `LMS Coefficient Estimator` subsystem. The DPD design in this example is similar to the HDL Implementation of Digital Predistorter example, which is optimized for memory depth 3 and polynomial degree 3. The input data is placed in a shift register and multiplexed to form a vector based on the memory depth. Then, the vector is concatenated with the nonlinear products of the data depending on the polynomial degree. This concatenation forms a vector of 9 elements, which equals the memory depth times the degree. The dot product of the obtained vector and estimated coefficients provides the predistorted input that is fed as input to the `RF Subsystem` subsystem when you enable the **DPDSwitch**. Open the `Digital Predistorter` subsystem.

```
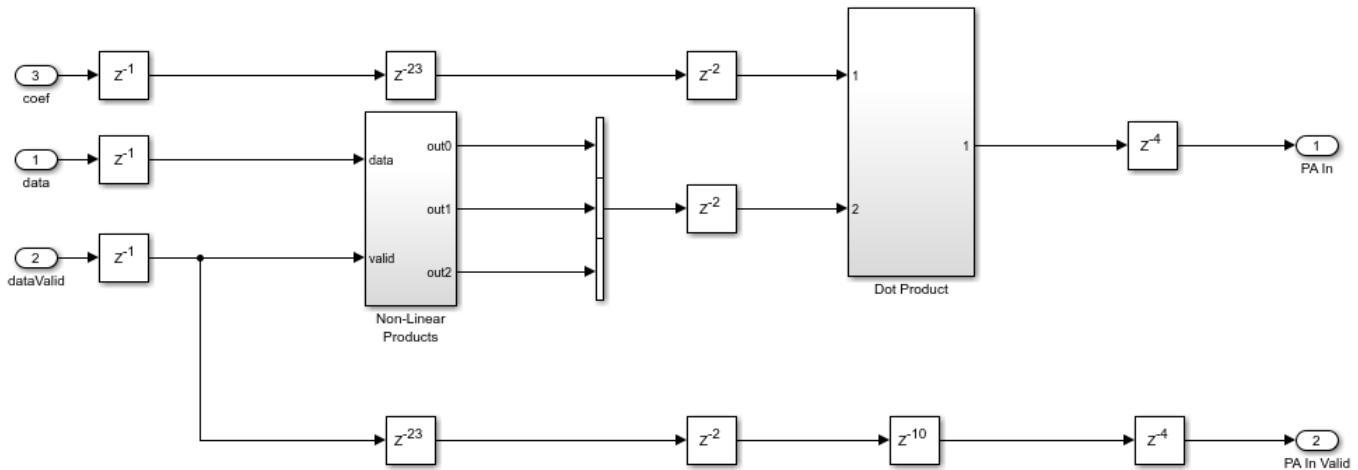load_system(modelname);
open_system([modelname '/HDL DPD with LMS/DPD_LMS/Digital Predistorter']);
```


LMS Coefficient Estimator

When PA characteristics vary over time and different operating conditions, using an adaptive estimation algorithm that runs on an FPGA to estimate the inverse of the PA is necessary. In this example, a hardware-friendly estimation algorithm based on the LMS method is considered due to its simple architecture and easier implementation on hardware with less resources compared to other estimation algorithms such as recursive least squares (RLS) and recursive prediction error method (RPEM). The `LMS Coefficient Estimator` subsystem estimates the DPD coefficients from the outputs of the `Digital Predistorter` subsystem (the PA input) and the PA output of the RF `Subsystem` subsystem. Similar to the `Digital Predistorter` subsystem, the `LMS Coefficient Estimator` subsystem also operates at 15.36 MHz. For memory depth 3 and polynomial degree 3, the `LMS Coefficient Estimator` subsystem estimates a total of 9 coefficients. Open the LMS `Coefficient Estimator` subsystem.

```
load_system(modelname);
open_system([modelname '/HDL DPD with LMS/DPD_LMS/LMS Coefficient Estimator']);
```
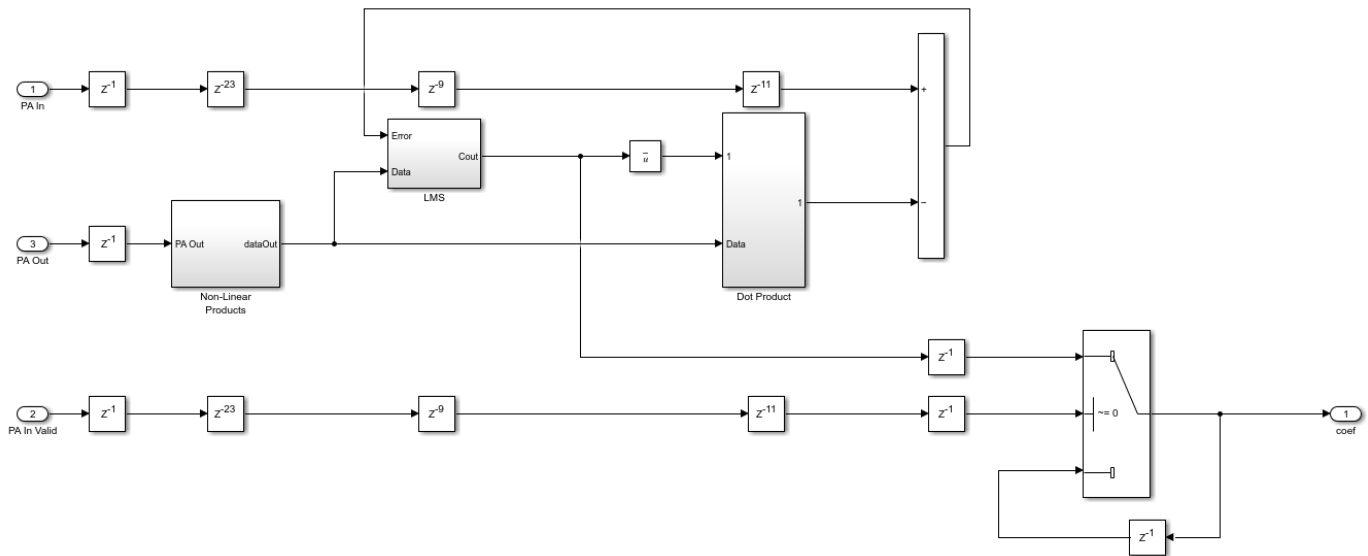
The PA output data from the `RF Subsystem` subsystem is placed in a shift register based on the memory depth, which is 3. Then, this vector is concatenated with the nonlinear products of the PA output data depending on the polynomial degree, which is 3. This concatenation forms a vector of 9 elements, which equals the memory depth times the degree. The `LMS` subsystem estimates the coefficients such that the error is minimal between the PA input data and the PA output data. The `Dot product` subsystem performs the dot product of the conjugate of the estimated coefficients and the concatenated PA output data. To send out the estimated coefficients based on **PA In Valid**, the example uses a switch. When the **PA In Valid** is high (1), this subsystem sends the current estimated coefficients. Otherwise, the subsystem sends the previously estimated coefficients.

Verification and Results

Run the `HDLDPDwithLMSCoeffExample` model. By default, the **DPDSwitch** is enabled. If you disable it, the error vector magnitude (EVM) and spectral regrowth in adjacent channels increase. The constellation and spectrum analyzer diagrams show the results of running the `HDLDPDwithLMSCoeffExample` model with the DPD enabled.

`sim(modelname);`

```
Estimating carrier frequency offset ...

First four frames are used for carrier frequency offset estimation.

Estimated carrier frequency offset is 1.291716e+00 Hz.

Detected and processing frame 5
-----------------------------------------

Header CRC passed

Modulation: 16QAM, codeRate=1/2 and FFT Length=128
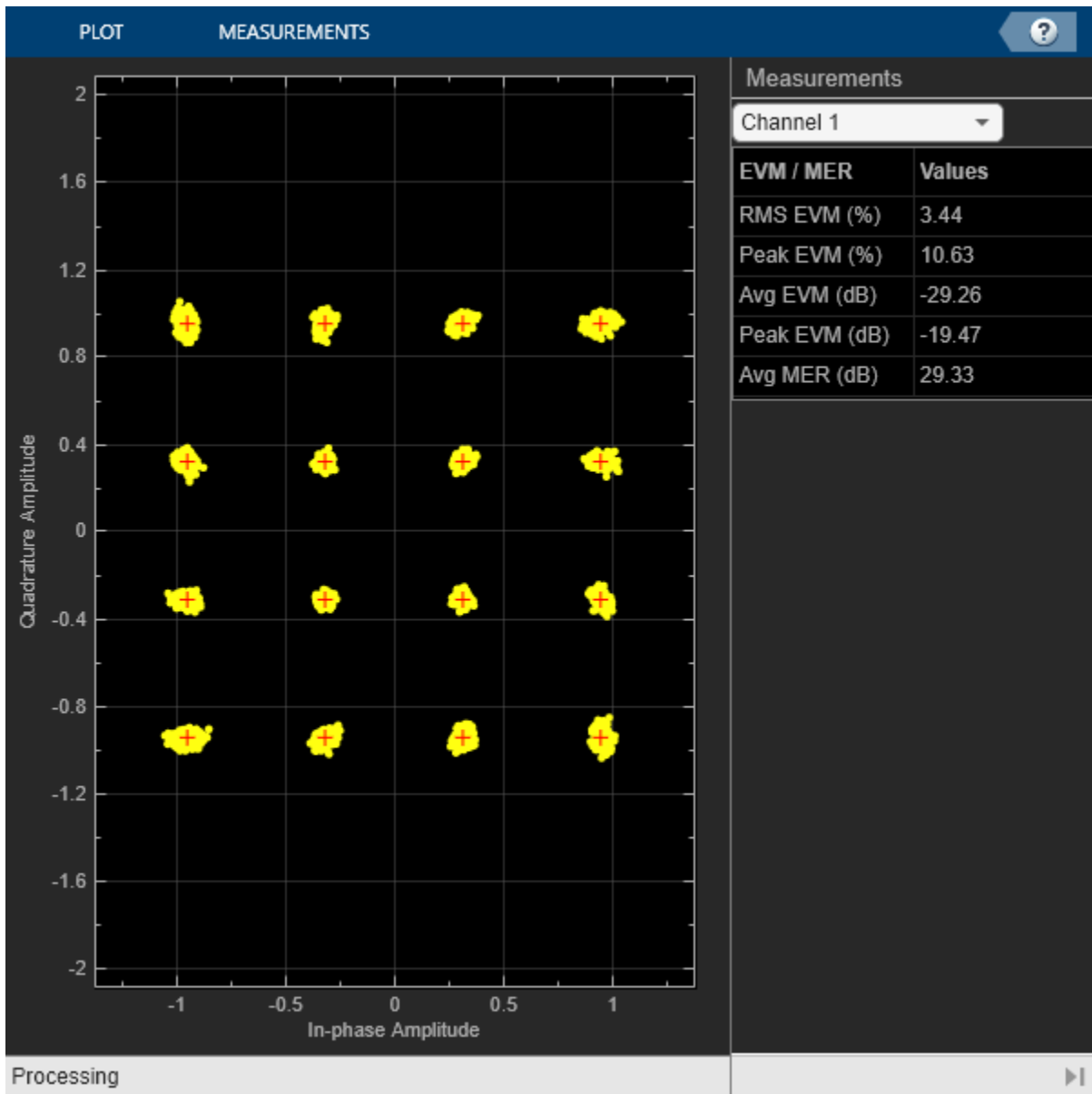
Data CRC passed

Data decoding completed
```

```
------------------------------------------

 Detected and processing frame 6
------------------------------------------

 Header CRC passed

 Modulation: 16QAM, codeRate=1/2 and FFT Length=128

 Data CRC passed

 Data decoding completed
------------------------------------------
```
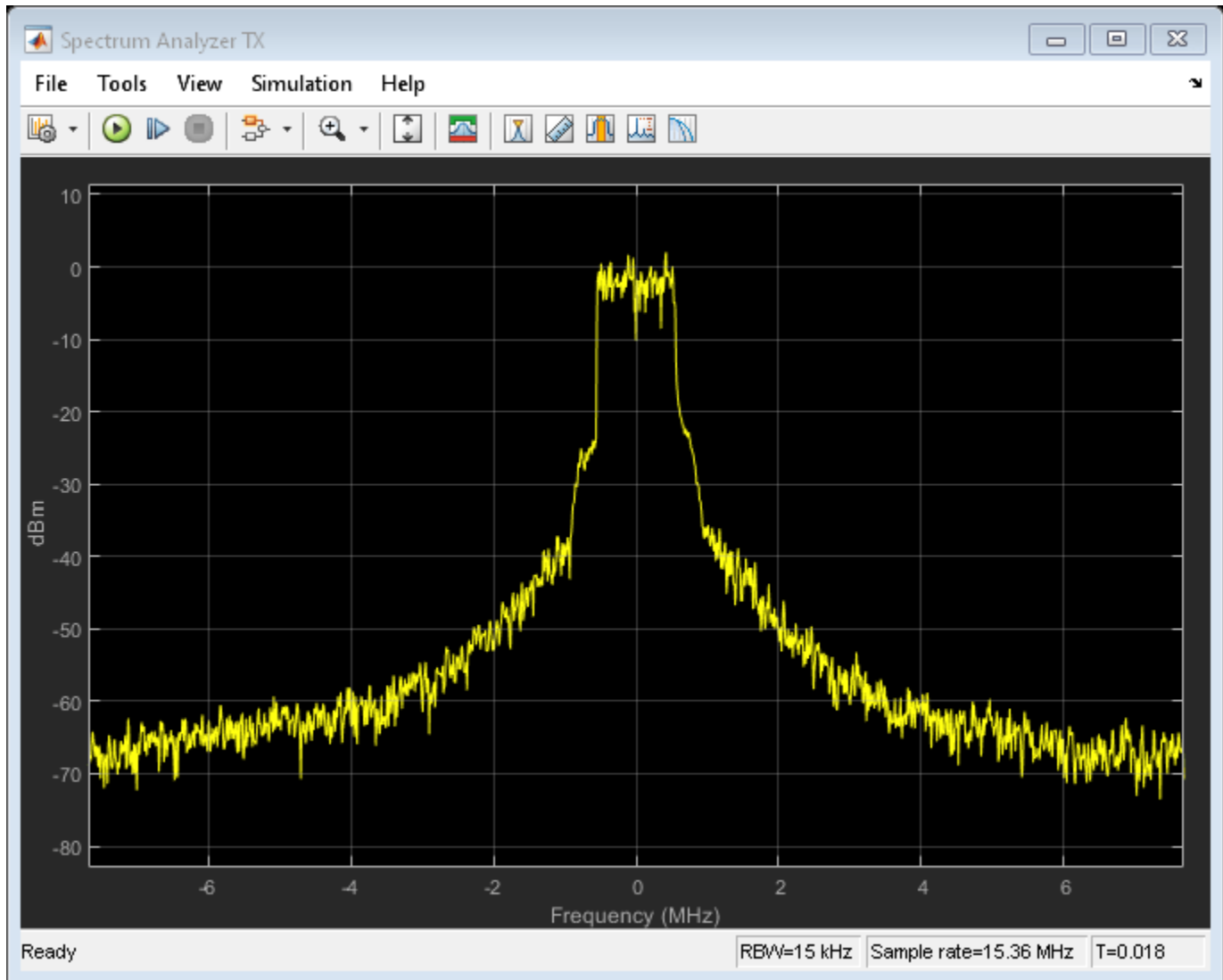
HDL Code Generation and Implementation Results

To check and generate HDL code for this model, you must have the HDL Coder™ product. To generate HDL code and a testbench for the `HDL DPD with LMS` subsystem, use the `makehdl` and `makehdltb` commands.

The `HDL DPD with LMS` subsystem is synthesized on the Xilinx® Zynq® Ultrascale RFSoC ZCU111 evaluation board. The frequency obtained after place and route is about 501 MHz. This table displays the post place and route resource utilization results for a 128 bit complex input.

```
F = table(...
    categorical({'Slice LUT'; 'Slice Registers';'DSP'}), ...
    categorical({'4023'; '10281'; '120'}), ...
    categorical({'425280'; '850560'; '4272'}), ...
    categorical({'0.95'; '1.21'; '2.81'}), ...
    'VariableNames', ...
    {'Resources','Utilized','Available','Utilization (%)'});
disp(F);
```

Resources	Utilized	Available	Utilization (%)
Slice LUT	4023	425280	0.95
Slice Registers	10281	850560	1.21
DSP	120	4272	2.81

DPD with Pre-calculated Coefficients

To perform digital predistortion with pre-calculated coefficients, use the `HDLDPDwithPreCalculatedCoeffExample` model and the `preCalculatedCoeff` MAT file containing pre-calculated coefficients. These coefficients are generated using the `HDLDPDwithLMSCoeffExample` model.

To create a new set of coefficients for any other PA settings, modify the parameters of the PA in the `HDLDPDwithLMSCoeffExample` model according to your requirement and run the model. If you already have a set of estimated coefficients to perform digital predistortion, replace the coefficients in `preCalculatedCoeff` MAT file and run the `HDLDPDwithPreCalculatedCoeffExample` model.

```
modelname = 'HDLDPDwithPreCalculatedCoeffExample';
open_system(modelname);
```
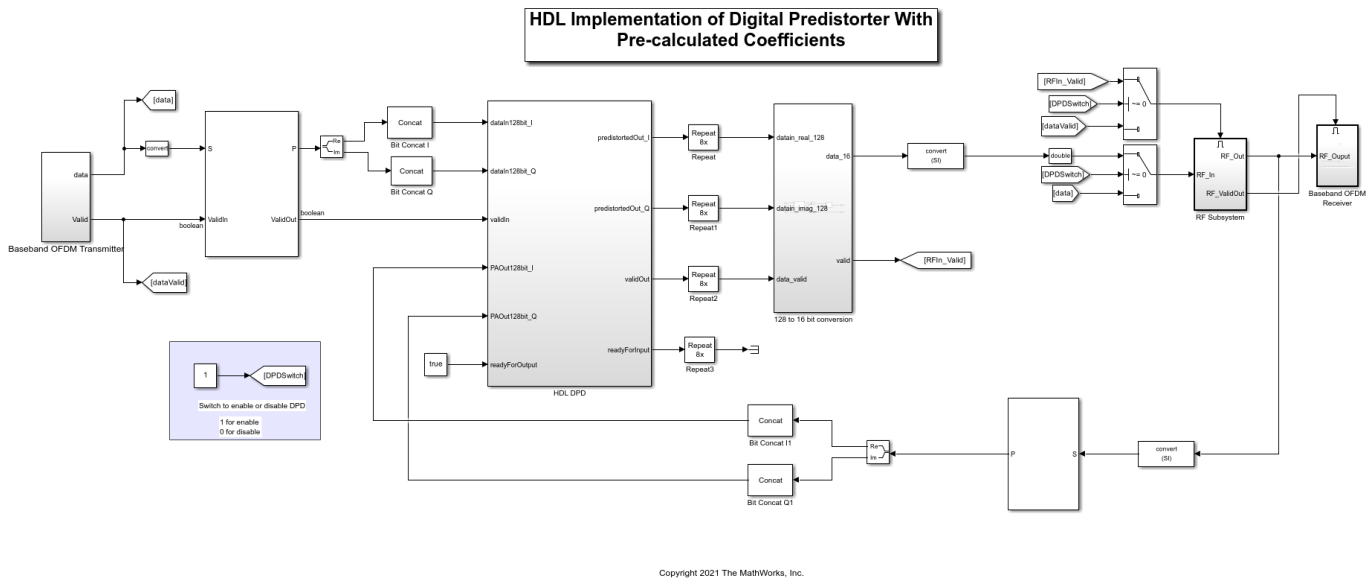

Verification and Results

Run the `HDLDPDwithPreCalculatedCoeffExample` model. By default, the **DPDSwitch** is enabled. If you disable it, the EVM and spectral regrowth in adjacent channels increase. The constellation and spectrum analyzer diagrams show the results of running the `HDLDPDwithPreCalculatedCoeffExample` model with the DPD enabled. You can see that when using DPD with pre-calculated LMS coefficients, the spectrum and constellation remain intact.

```
sim(modelname);
```

```
Estimating carrier frequency offset ...
```

```
First four frames are used for carrier frequency offset estimation.

Estimated carrier frequency offset is 1.269763e+00 Hz.

Detected and processing frame 5
-----------------------------------------

Header CRC passed

Modulation: 16QAM, codeRate=1/2 and FFT Length=128

Data CRC passed

Data decoding completed
-----------------------------------------

Detected and processing frame 6
-----------------------------------------

Header CRC passed

Modulation: 16QAM, codeRate=1/2 and FFT Length=128

Data CRC passed

Data decoding completed
-----------------------------------------
```
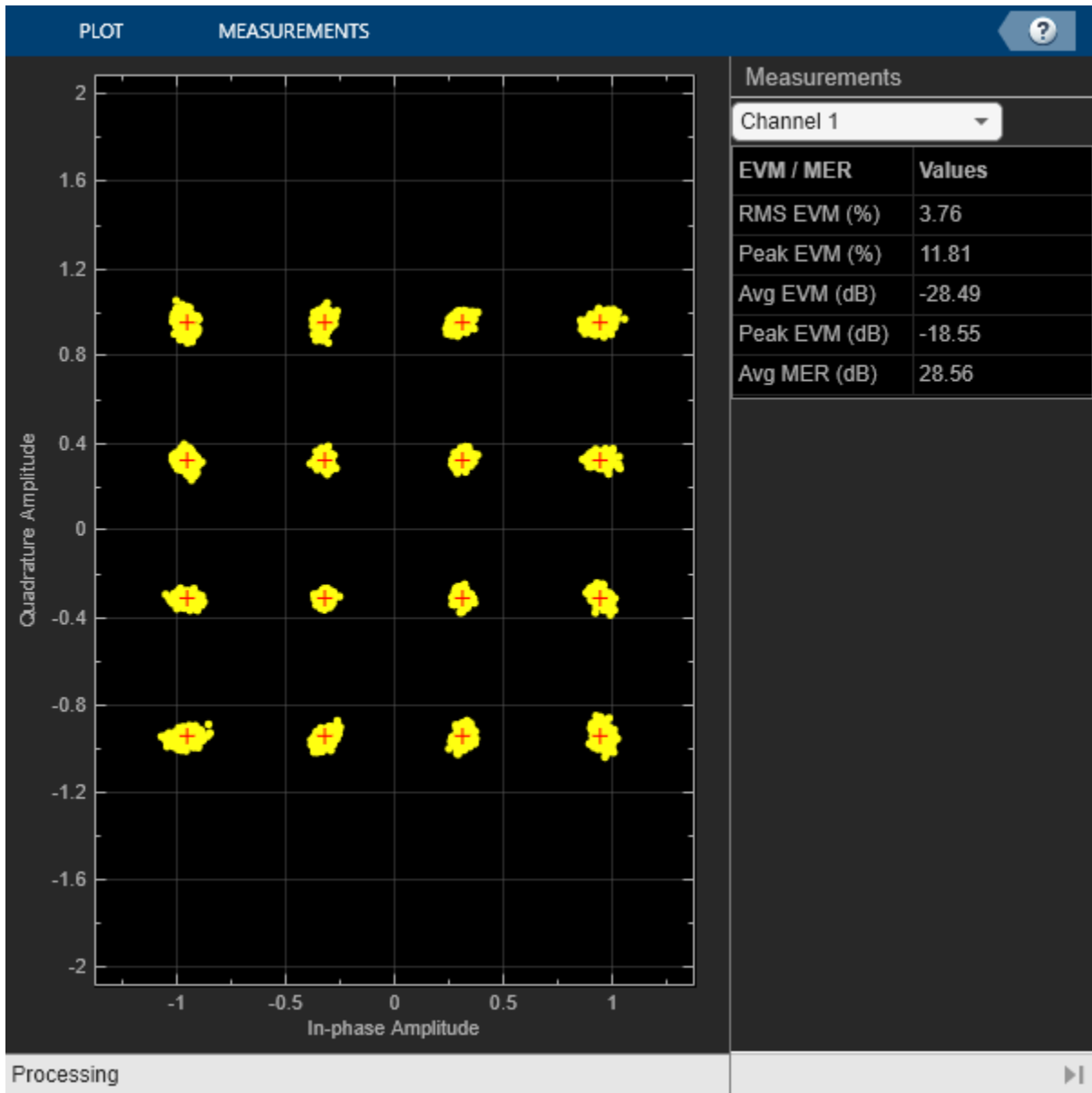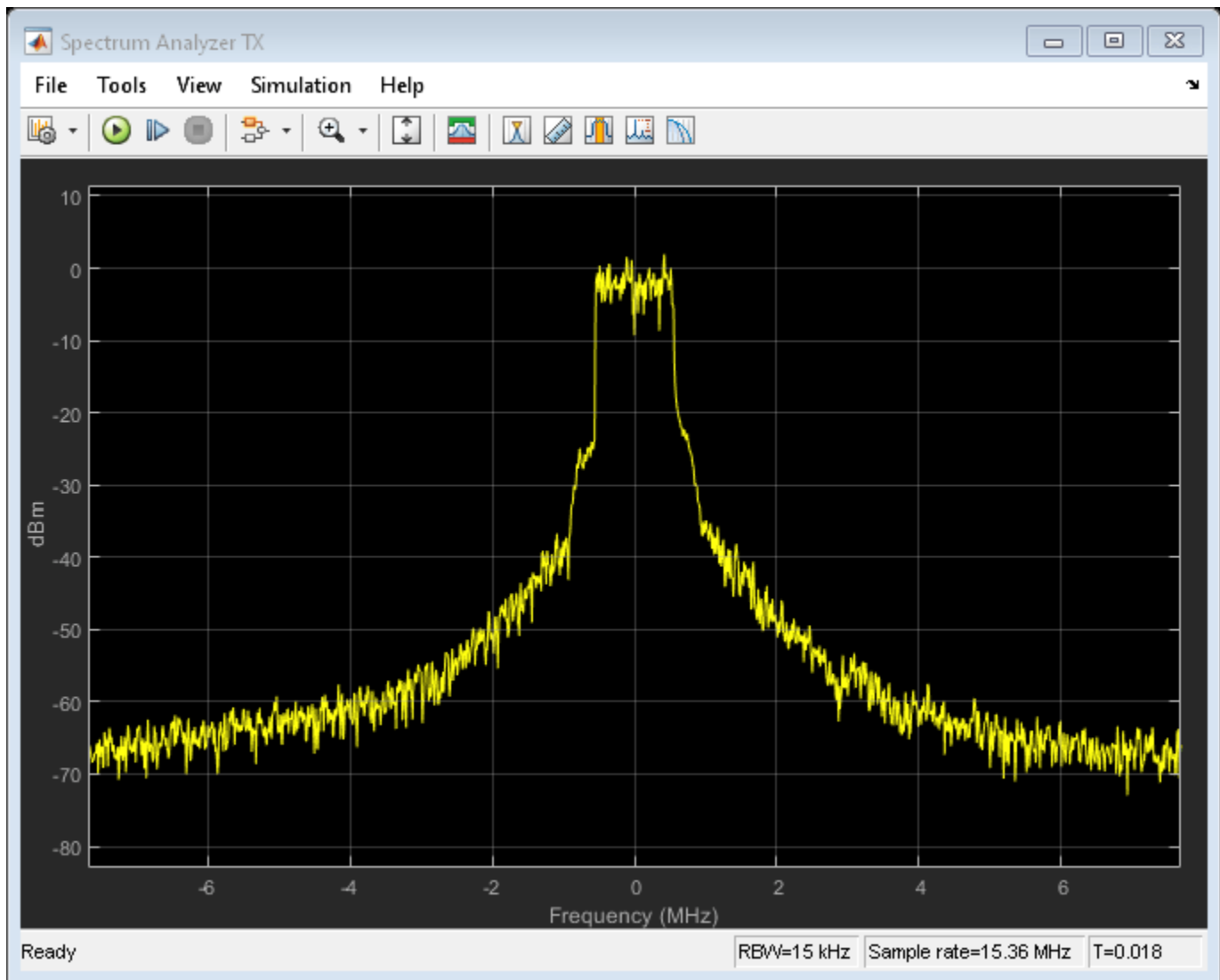
HDL Code Generation and Implementation Results

To check and generate HDL code for this model, you must have the HDL Coder™ product. To generate HDL code and a testbench for the `HDL DPD` subsystem, use the `makehdl` and `makehdltb` commands.

The Digital Predistorter with pre-calculated LMS coefficients subsystem is synthesized on a Xilinx® Zynq® Ultrascale RFSoC ZCU011 evaluation board. The frequency obtained after place and route is about 470 MHz. This table displays the post place and route resource utilization results for a 128 bit complex input.

```
F = table(...
    categorical({'Slice LUT'; 'Slice Registers';'DSP'}), ...
    categorical({'1714'; '4606'; '36'}), ...
    categorical({'425280'; '850560'; '4272'}), ...
    categorical({'0.4'; '0.54'; '0.84'}), ...
    'VariableNames', ...
```

```
    {'Resources','Utilized','Available','Utilization (%)'});
disp(F);

        Resources        Utilized      Available      Utilization (%)
    _____      _____      _____      _____

    Slice LUT               1714         425280             0.4
    Slice Registers         4606         850560             0.54
    DSP                       36           4272             0.84
```

See Also

Related Examples

- "HDL Implementation of Digital Predistorter" on page 4-55

DVB-S2 HDL PL Header Recovery

This example shows how to implement DVB-S2 time, frequency, and phase synchronization and PL header recovery using Simulink® blocks that are optimized for HDL code generation and hardware implementation.

Digital Video Broadcasting Satellite Second Generation (DVB-S2) modems operate in C (4-8 GHz), Ku (12-18 GHz) and Ka (26-40 GHz) frequency bands. According to the DVB-S2 standard, the satellite transponder bandwidth ranges from 1 MHz to 72 MHz. The model in this example operates at a symbol rate of 25 Mbaud with a root raised cosine (RRC) filter roll-off factor of 0.35. For a MATLAB® implementation of end-to-end DVB-S2 receiver, see the "End-to-End DVB-S2 Simulation with RF Impairments and Corrections" (Satellite Communications Toolbox) example.

This example shows how to design a DVB-S2 HDL receiver synchronization and physical layer (PL) header recovery system that can handle radio frequency (RF) impairments. The model in this example performs symbol timing synchronization, frame synchronization, coarse and fine frequency synchronization, phase offset estimation and correction, gain correction, and noise variance estimation. Then the model decodes the PL header information followed by fine phase synchronization.

Model Architecture

This section explains the high-level architecture of the model. The model receives the DVB-S2 transmitter waveform sequence that streams into the Coarse Frequency Compensator block. The Symbol Synchronizer block extracts the modulated symbol sequence from the Matched Filter block output and the Frame Synchronizer block locates the start of each frame in the modulated symbol sequence. The PL Descrambler block descrambles the scrambled data symbols and the Pilot Generator block indicates the pilot locations in the frame synchronized sequence. The Coarse Frequency Estimator block estimates the frequency offset, which is used to correct the frequency offset in the transmitter waveform sequence at the model input by conjugate multiplication of the estimate. The Fine Frequency Compensator block corrects the residual frequency left in the PL descrambled sequence. The Coarse Phase Error Compensator block corrects the coarse phase deviation in the Fine Frequency Compensator block output sequence. The phase error compensated sequence is magnitude corrected in the Gain Control block and the gain-corrected sequence is used to estimate noise variance. The Demultiplexer divides the gain-corrected sequence into physical layer signaling code (PLSC) symbols and descrambled data symbols in each frame. The PL Header PLSC Decoder block decodes header parameters **MODCOD** and **FECFrame**. The Fine Phase Compensator block uses the **MODCOD** parameter and corrects the residual phase in the descrambled data symbols that stream into the symbol demodulator.

This block diagram shows the high-level architecture of the model.

DVB-S2 Synchronization and PLHeader Recovery
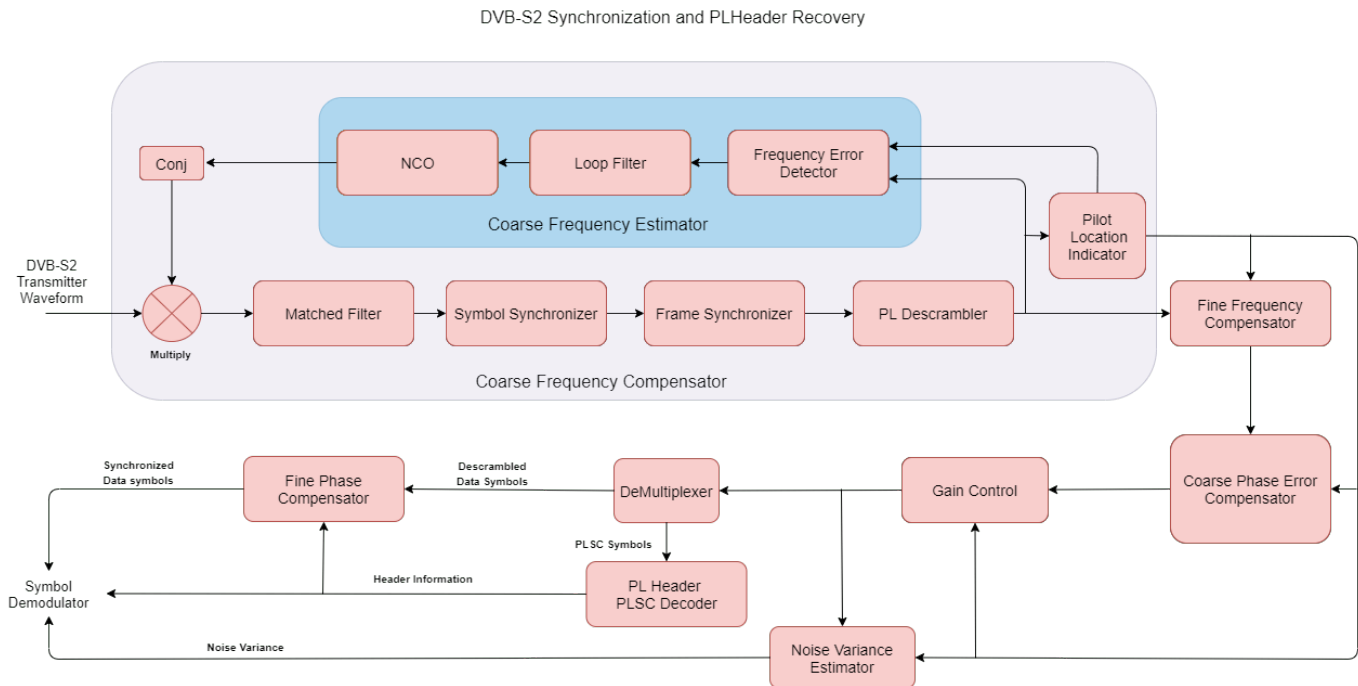
File Structure

This example uses two Simulink models, six MATLAB files, and one Simulink data dictionary.

- `dvbs2hdlPLHeaderRecovery.slx` — Top-level Simulink model.
- `dvbs2hdlSyncPLHeaderRecoveryCore.slx` — Model reference that synchronizes time, frequency, and phase, and decode PL Header.
- `getdvbs2LDPCParityMatrices.m` — Download the LDPC matrices `.mat` file.
- `dvbs2hdlRxParameters.m` — Generate parameters for the `dvbs2hdlSyncPLHeaderRecoveryCore.slx` model reference.
- `dvbs2hdlPhaseNoise.m` — Introduce phase noise to the input sequence.
- `dvbs2hdlRxInit.m` — Generate the transmitter waveform and initialize the `dvbs2hdlSyncPLHeaderRecoveryCore.slx` model reference.
- `dvbs2hdlStreamRecovery.m` — Decode baseband frame (BBFRAME).
- `dvbs2hdlPLHeaderRecoveryVerify.m` — Gather PL header parameters and XFECFRAME symbols, demodulate symbols, decode FEC (LDPC and BCH), recover baseband frame, and compute bit errors using Satellite Communications Toolbox functions.
- `dvbs2hdlReceiverData.sldd` — Store bus signal configurations that come out of the model reference.

System Interface

This figure shows the top-level overview of the `dvbs2hdlPLHeaderRecovery.slx` model.

DVB-S2 HDL PL Header Recovery
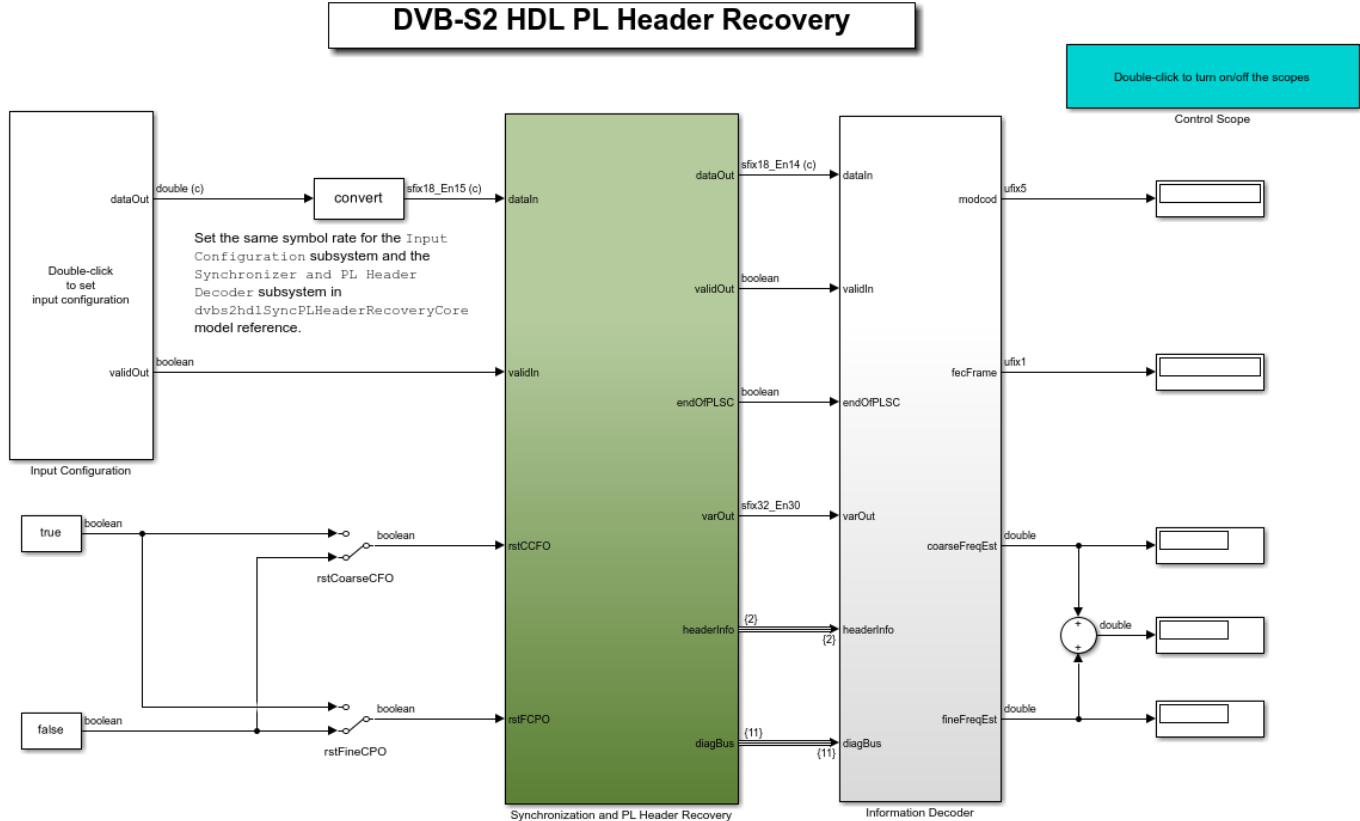
Copyright 2021 The MathWorks, Inc.

Model Inputs

- **dataIn** — Input data, specified as an 18 bit complex data with a sample rate that is four times the symbol rate.

- **validIn** — Control signal to validate the **dataIn** input port, specified as a Boolean scalar.

- **rstCCFO** — Control signal to reset the coarse frequency compensation loops, specified as a Boolean scalar.

- **rstFCPO** — Control signal to reset the fine phase compensation loops, specified as a Boolean scalar.

Model Outputs:

- **dataOut** — Decoded output symbols, returned as an 18 bit complex scalar.

- **validOut** — Control signal to validate the **dataOut** output port, specified as a Boolean scalar.

- **endOfPLSC** — Control signal to indicate the end of PLSC symbols in each synchronized frame, specified as a Boolean scalar.

- **nVar** — Estimated noise variance, returned as a 32 bit complex scalar.

- **headerInfo** — Bus signal to provide the parameters **MODCOD** and **FECFrame** of the PL header in each synchronized frame.

- **diagBus** — Bus signal to provide the coarse frequency normalized with the sample rate, the fine frequency normalized with the symbol rate, the symbol synchronized output, the PLSC symbols, and the pilot symbols.

Model Structure

This figure shows the top-level model of the `Synchronization and PL Header Recovery` subsystem. It comprises `Time Frequency and Coarse Phase Synchronizer`, `PL Header Recovery`, and `Fine Phase Synchronizer` subsystems.
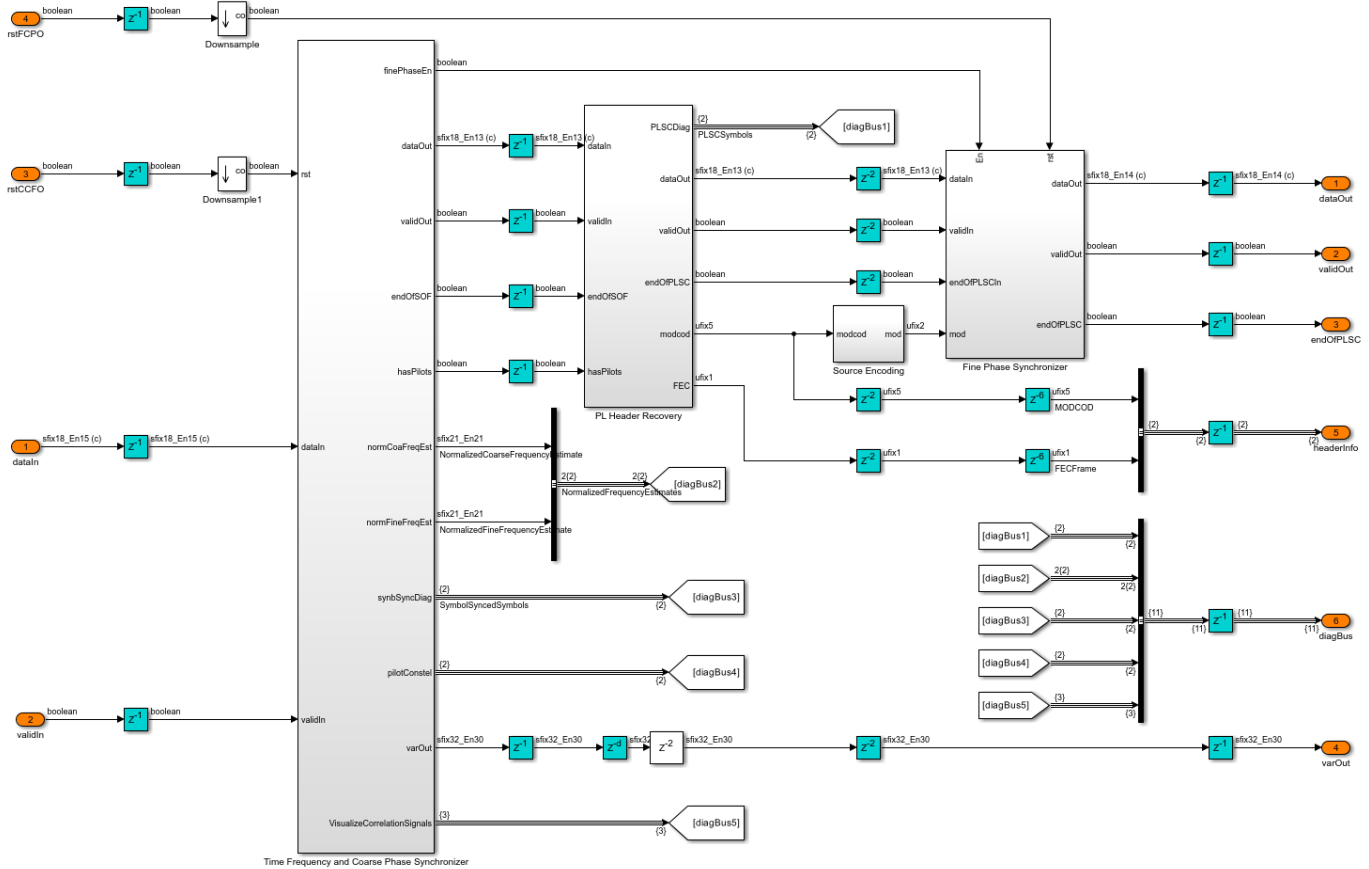
Time Frequency and Coarse Phase Synchronizer

The `Time Frequency and Coarse Phase Synchronizer` subsystem comprises `Time and Coarse Frequency Synchronizer`, `Fine Frequency Synchronizer`, and `Coarse Phase Synchronizer` subsystems.
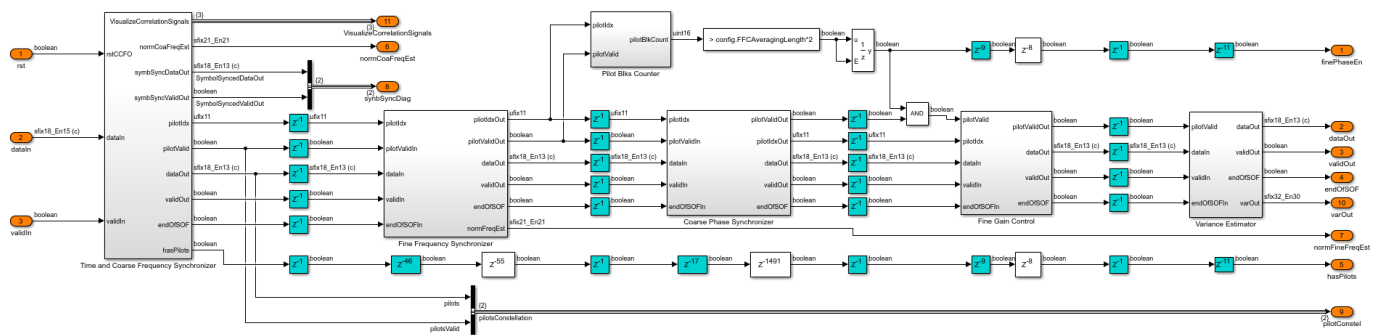
Time and Coarse Frequency Synchronizer

The `Time and Coarse Frequency Synchronizer` subsystem compensates coarse frequency in a frequency-locked loop (FLL) system. The normalized loop bandwidth of the FLL system is set to 1e-4. The loop involves RRC matched filtering, symbol synchronization, frame synchronization, PL descrambling, pilot extraction, and coarse frequency estimation.
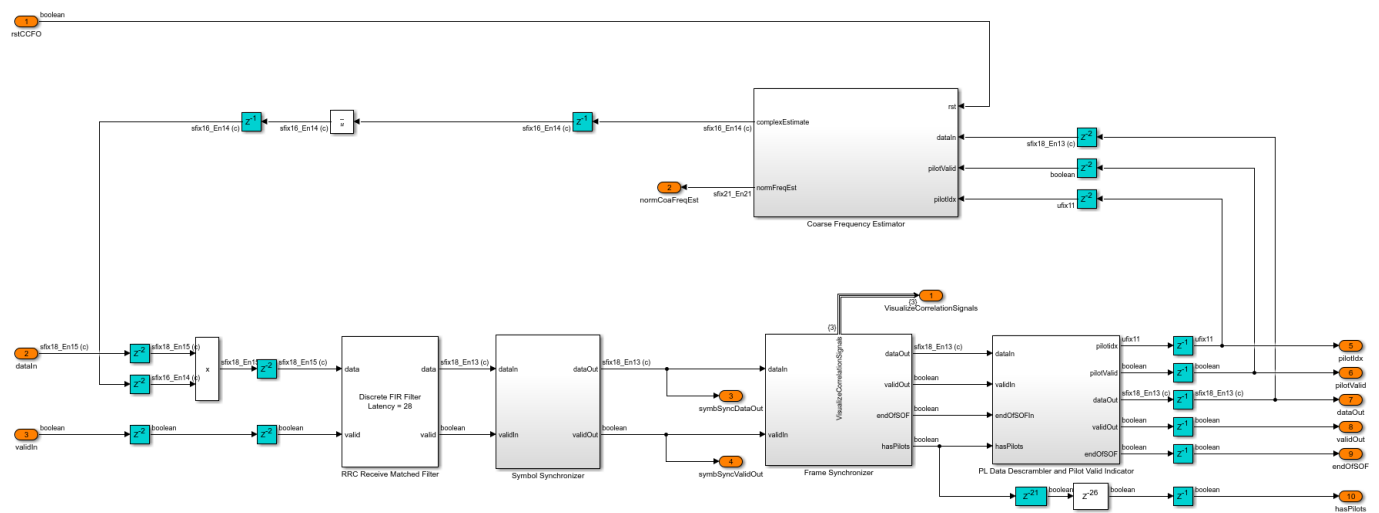
Coarse Frequency Estimator

The `Coarse Frequency Estimator` subsystem performs frequency error detection, loop filtering, and direct digital synthesis. The frequency error detection is described with equation C.2 in the Annex C.4 of [2]. The coarse frequency estimator is a pilot-aided frequency estimator. The `Frequency Error Detector` subsystem outputs frequency error at the pilot locations. The frequency error is passed through the loop filter and the output of the loop filter drives the NCO block to generate the complex exponential sinusoidal samples. These samples are conjugated and multiplied by the input sequence to correct the frequency offset. The loop filter is disabled for frequency error filtering after 32 pilot blocks so that the estimated frequency remains stable. A reset signal **rstCCFO** resets the loop filter and restarts the estimation process.

RRC Receive Matched Filter

The RRC Receive Matched Filter is a Discrete FIR Filter block with matched filter coefficients with four samples per symbol, and a roll-off factor of 0.35. The RRC matched filtered output is an RC pulse-shaped waveform that has zero inter symbol interference (ISI) characteristics at the maximum eye opening in the eye diagram of the waveform. Also, the matched filtering process maximizes the signal-to-noise power ratio (SNR) of the filter output.

Symbol Synchronizer

The `Symbol Synchronizer` subsystem is a phase locked loop (PLL) based implementation as described in the chapter 8.4 of [4]. The subsystem generates one output sample for every four input samples. The PLL loop is set with a normalized loop bandwidth of 8e-3. The `Interpolation Filter` subsystem implements a piecewise parabolic interpolator with a hardware resource efficient farrow structure. This filter introduces fractional delays in the input waveform. As specified in Annex C.2 of [2], the `Gardner TED` subsystem implements a Gardner timing error detector. The loop filter filters the timing error and pass it on to the `Interpolation Control` MATLAB function block. This block implements a mod-1 decrementing counter to calculate fractional delays based on the loop-filtered timing error to generate interpolants at optimum sampling instants. The `Rate Handle` subsystem selects the required interpolant indicated by the strobe.

Frame Synchronizer

The frame synchronizer implementation is described in the Annex C.3.1 of [2]. The `Correlator` subsystem in the `Frame Synchronizer` subsystem generates the start of frame (SOF), PLSC correlation values and a threshold. The SOF correlated sequence is delayed by a length of PLSC sequence so that the correlation peaks of SOF and PLSC are aligned. The `Pilot and Sync Pulse Indicator` subsystem detects the threshold exceeded correlation value and also detects the existence of pilots in the current frame.
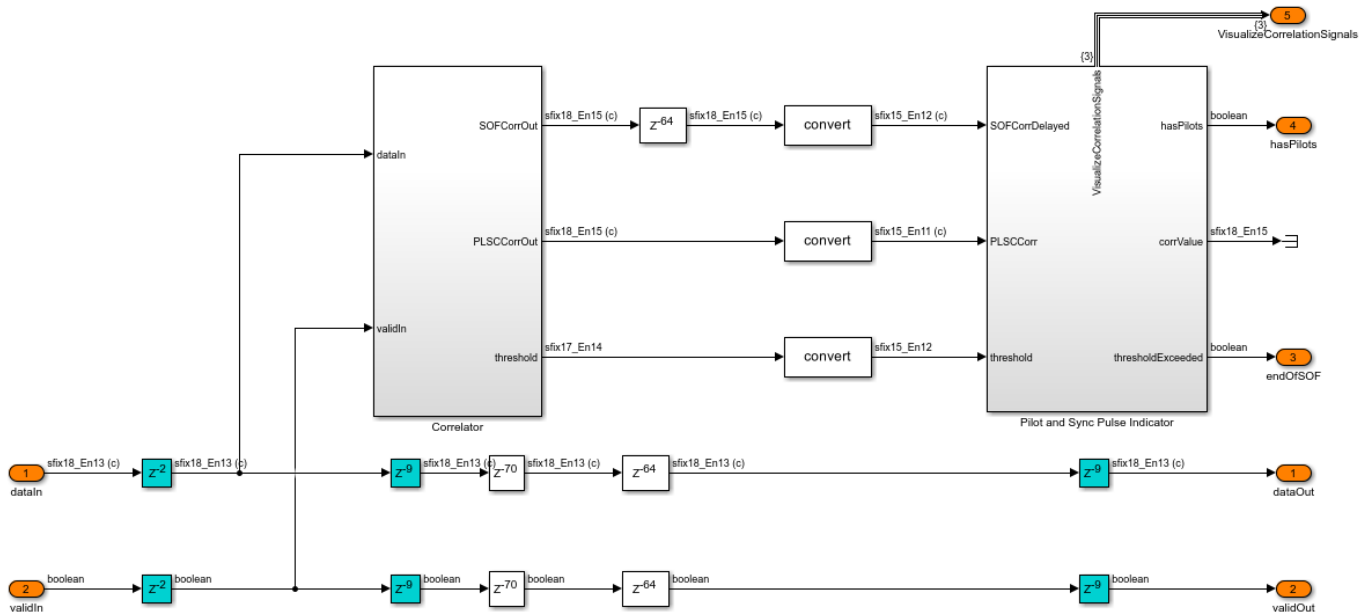
The `Correlator` subsystem implements differential detection and removes the frequency offset dependency in the input sequence. The output sequence is continuously cross-correlated with SOF and PLSC correlators. In addition, the energy of the signal is computed on each time step and then scaled and summed up in the span of each correlator filter length in the `Moving Sum` subsystem. The scaling factors used before the `Moving Sum` subsystem are derived from each of the correlation sequences respectively in the `dvbs2hdlRxParameters.m` file. The two scaled energy values are added to generate a threshold.
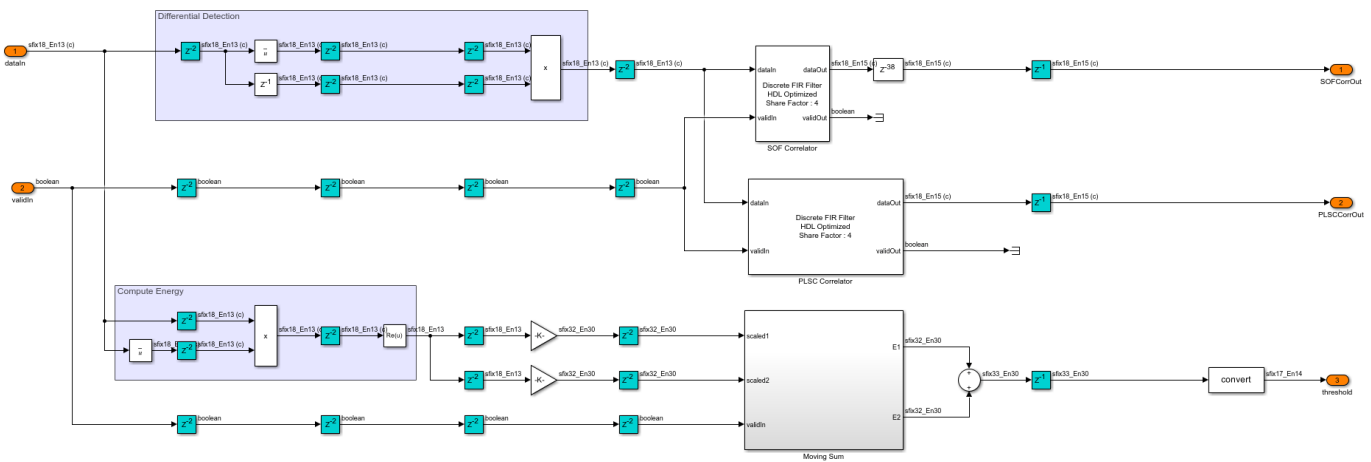
The `Pilot and Sync Pulse Indicator` subsystem adds and subtracts the SOF and PLSC correlation values and computes energy at each time step to generate two correlation metrics. The threshold is downscaled with a precomputed value in the `dvbs2hdlRxParameters.m` file, and a lower limit is applied to saturate the threshold with a lower bound. Both of the correlation metrics are compared with the downscaled threshold value. The existence of pilots in the current frame is confirmed if the correlation metric obtained by adding SOF and PLSC correlation values exceeds the downscaled threshold. For a given frame, only one of the two correlation metrics exceeds the threshold based on the existence of pilots.
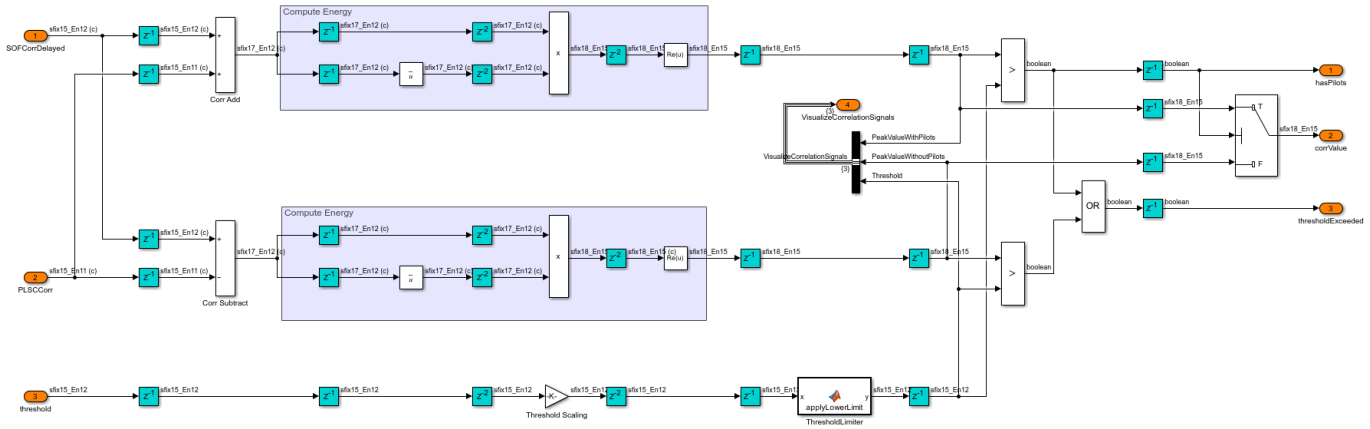
PL Data Descrambler

The `PL Data Descrambler` subsystem uses `PL Scrambler Gold sequence Generator` subsystem, which is described in section 5.5.4 of [1]. The `PL Scrambler Gold Sequence Generator` subsystem resets for every frame. The gold sequence is used as an address to the `PL Scramble Sequence Mapper LUT` block to generate the PL scrambling sequence. The scrambling sequence is conjugated to generate the PL descrambling sequence, and the descrambling is performed by multiplying PL descrambling sequence with the input sequence. A switch is used to multiplex the PLSC symbols and the descrambled data symbols.
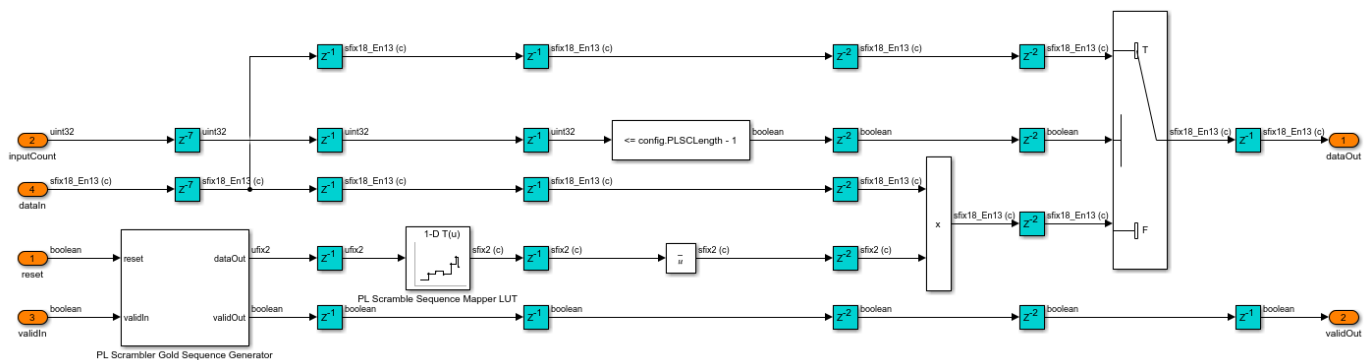
Pilot Valid Indicator

The `Pilot Valid Indicator` subsystem counts the input sequence and assigns a pilot index for each symbol. As specified in section 5.5.3 of [1], the pilots of length 36 symbols exist after 16 slots (1440 symbols) in a pilot-active XFECFRAME (pilot activeness is confirmed in the frame synchronizer). The subsystem generates a pilot valid signal for 36 symbols to indicate the location of

the pilot block. The counter resets after the pilot block. This process continues for the rest of the XFECFRAME. The signal, which indicates the end of the PLSC symbols of the next XFECFRAME, determines the end of the current XFECFRAME. The pilot valid signal is not generated after 16 slots if the next XFECFRAME is detected, as the PLSC symbols take the place of pilots.
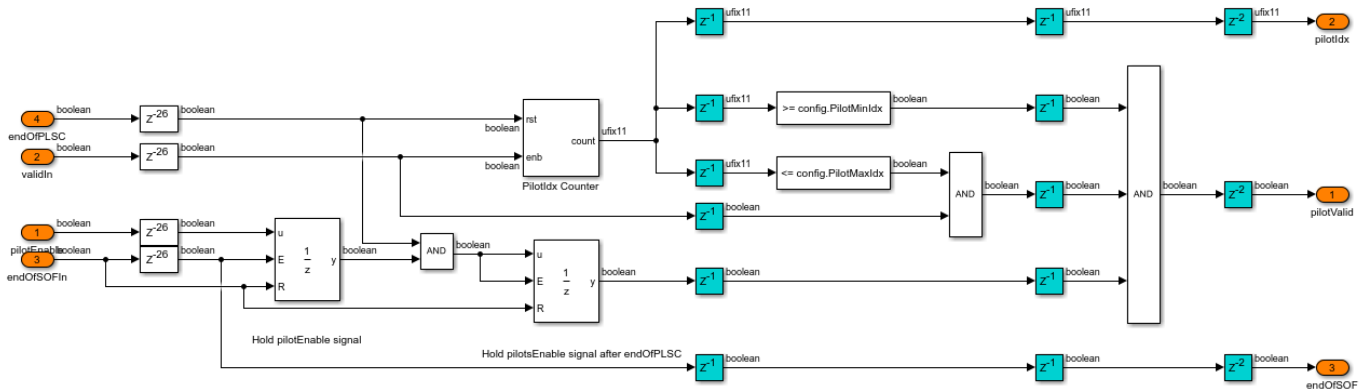
Fine Frequency Synchronizer

The `Fine Frequency Synchronizer` subsystem uses the Modified L and R algorithm, as described by the equation C.3 in Annex C.4 of [2]. The L and R algorithm is described in [3]. The subsystem implements an 18 point autocorrelation function of the L and R algorithm followed by a 512 length moving average filter. The frequency estimate from the `Modified L and R Algorithm` subsystem drives the NCO block to generate the complex exponential sinusoidal samples, which are conjugated and used to correct the frequency offset in the input.
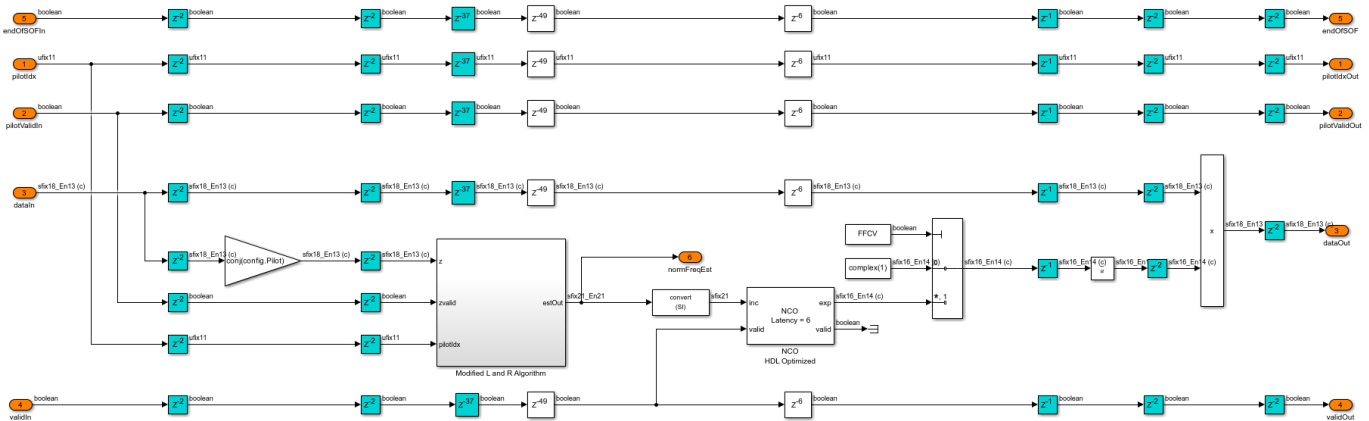
Coarse Phase Synchronizer

The `Coarse Phase Synchronizer` subsystem uses the pilot aided linear interpolation technique. The `Coarse Phase Estimator` subsystem estimates the complex phase from each of the 36 pilot symbols and performs averaging, which results in one estimate from each pilot block. The `Unwrapping Algorithm` subsystem implements the equation C.7 in section C.6.1 of [2] and interpolates the complex phase estimate from two consecutive pilot blocks. This interpolated estimate is used to compensate the phase of the symbols in between these two consecutive pilot blocks.
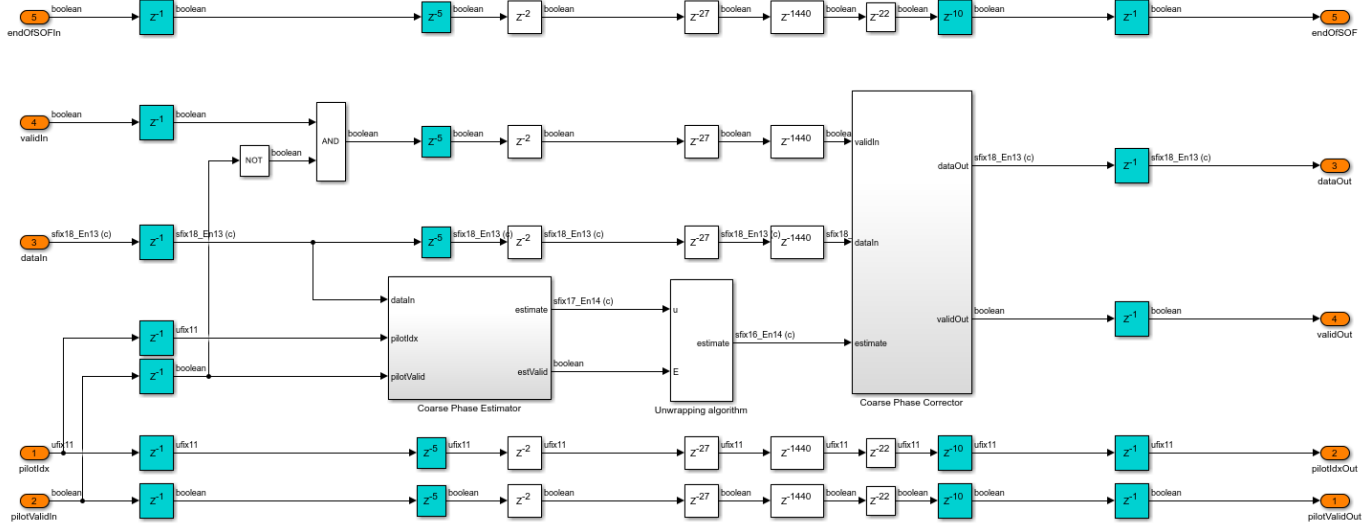
Fine Gain Control

The `Fine Gain Control` subsystem performs magnitude correction using the estimates derived from the pilot symbols. The input sequence is time and frequency synchronized before gain control. Each estimate is derived by multiplying the pilot symbol in the input sequence with reference pilot. The divide (1/32 gain block) and the integrator averages 32 estimates and stores in a register. The input is divided with the averaged estimate to correct the input magnitude.
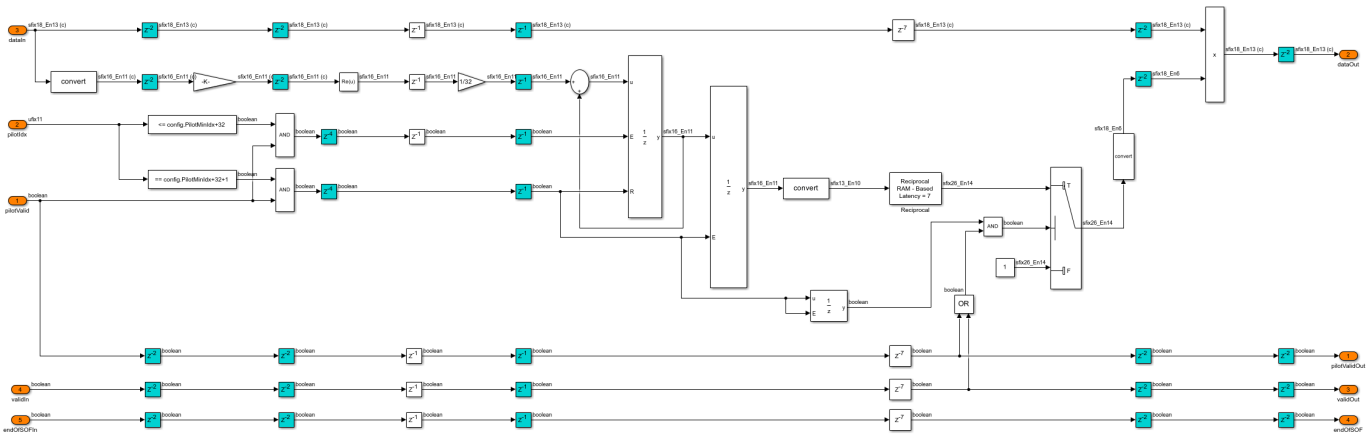
Variance Estimator

The `Variance Estimator` subsystem computes the noise variance of the input signal. The input sequence is time and frequency synchronized and gain-corrected before estimating the variance. The reference pilot symbols are subtracted from the noise affected pilot symbols in the input sequence to generate zero-mean noisy symbols. The variance is computed by absolute squaring these symbols and averaging previous 2048 symbols using a moving average filter of length 2048.
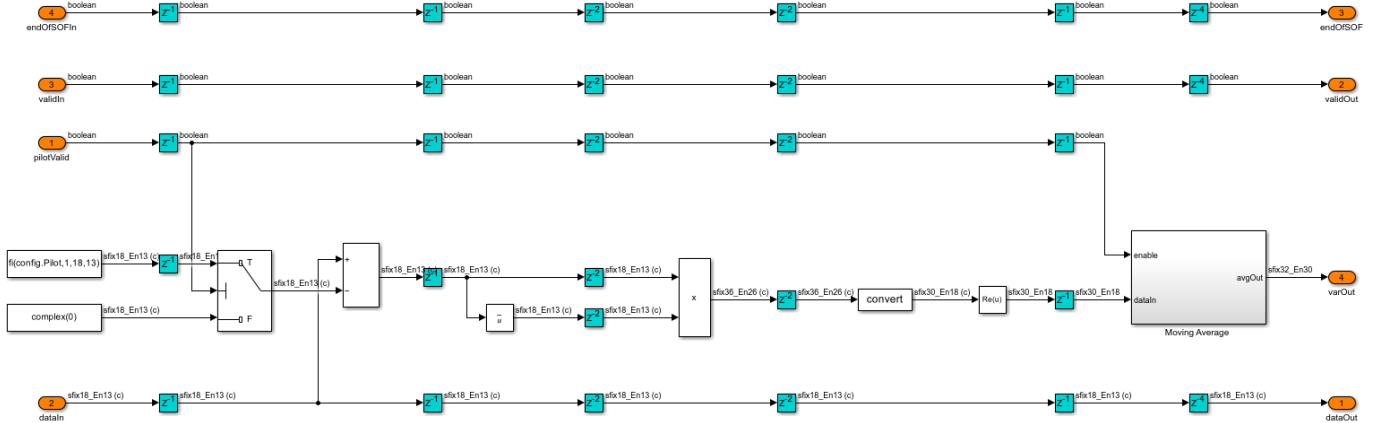
PL Header Recovery

The `PL Header Decoder` subsystem in the `PL Header Recovery` subsystem decodes the PLSC symbols of the PL header on a frame-by-frame basis to get the **MODCOD** and **FECFrame** type of a frame.

MODCOD indicates the modulation and coding scheme of the frame, and **FECFrame** indicates the type of the frame (short or normal) as described in sections 5.5.2.2 and 5.5.2.3 of [1].

The PLSC symbols contain seven information bits that are bi-orthogonally encoded with a (64,7) code. The construction of the 64 bit code is such that each odd bit in the code is either always a flipped bit or equal to the even consecutive bit based on whether the pilots exist or does not exist in the frame respectively, as described in section 5.5.2.4 of [1]. The 64 encoded bits are pi/2-BPSK modulated as described in section 5.5.2 of [1].

The `PL Frame Demultiplexer` subsystem demultiplexes the PLSC symbols and the PL data frame of the input. The signals from the **hvalid** and **dvalid** ports indicate the locations of PLSC symbols and PL data frame symbols respectively. The PLSC symbols are streamed into the `PL Header Decoder` subsystem.
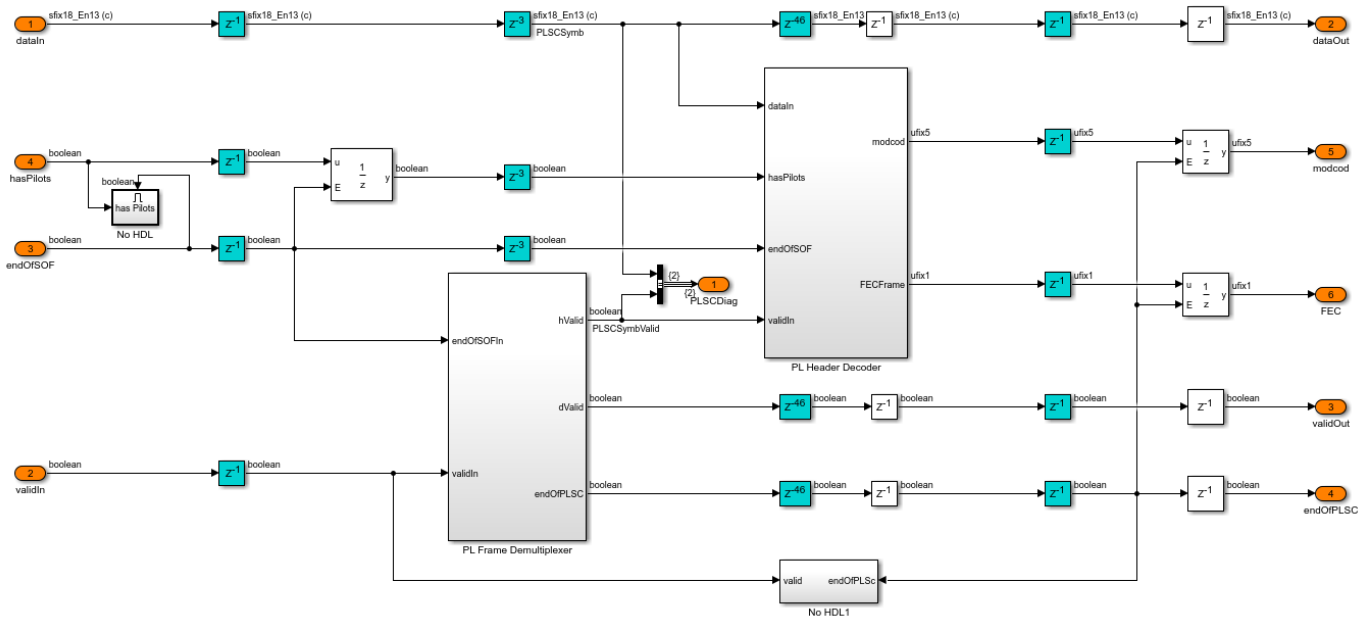
PL Header Decoder

The `PLSC Descrambler` subsystem in the `PL Header Decoder` subsystem descrambles the PLSC symbols. The signal from the **indexIndicator** port of the `PLSC Descrambler` subsystem distinguishes the even and odd locations of the PLSC symbols. The `pi/2 BSPK Soft Bit Demodulator` subsystem demodulates the PLSC symbols. If the pilots exists in the current PLFRAME (which is decided in the frame synchronization), the `Bit Flipping and Averaging` subsystem multiplies the odd soft bits by –1 in the PLSC symbols. A bit flip for a hard bit is same as multiplying by –1 for a soft bit. The subsystem averages the soft bits in even and odd locations to get one estimate. Likewise, 32 soft bits are generated from 64 soft bits. A maximum likelihood (ML) decoder is used to decode the (32,6) bi-orthogonal encoded bits. The 6 decoded bits are used to construct the **MODCOD** and **FECFrame** type.
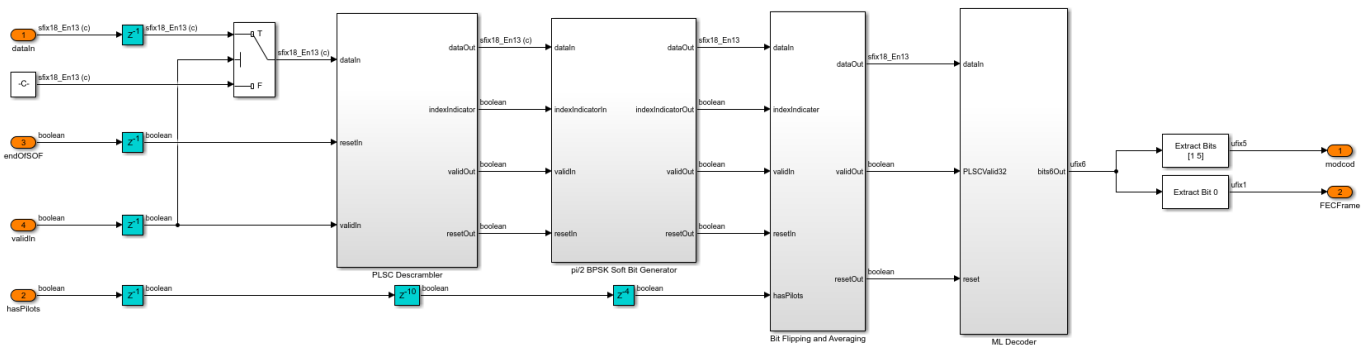
ML Decoder

The `ML Decoder` subsystem decodes the (32,6) bi-orthogonal code by choosing the maximum likelihood codeword. A total of $2^6 = 64$ codeword combinations, each 32 bits wide, are precomputed in the `dvbs2hdlRxParameters.m` file. The codewords are stored as integers in the `uint32` format, with the first 32 codewords in one look-up table (LUT) and the next 32 codewords in another LUT. The

LUT storing is such that the most significant bit of all of the codewords is called first, followed by the next significant bits, and so on. A bit level Euclidean distance is computed in the `Compute Euclidean Distance` subsystem with –1 and 1 as reference values for bit 0 and bit 1, respectively. The `Euclidean Distance Sum Up` subsystem adds all of the 32 bit level Euclidean metrics over time and generates a codeword Euclidean metric for each codeword. This subsystem uses a *for each* iterator to repeat the execution for all of the codewords and generates 64 codeword Euclidean metrics. The minimum Euclidean metric of 64 combinations maps to the maximum likelihood codeword. The maximum likelihood code word is used to construct the 6 bit input, and the **MODCOD** and **FECFrame** type values.

For a hardware-friendly implementation, the Euclidean metric is computed (computing involves multipliers) outside the `ML Decoder` subsystem as it uses a *for each* iterator.
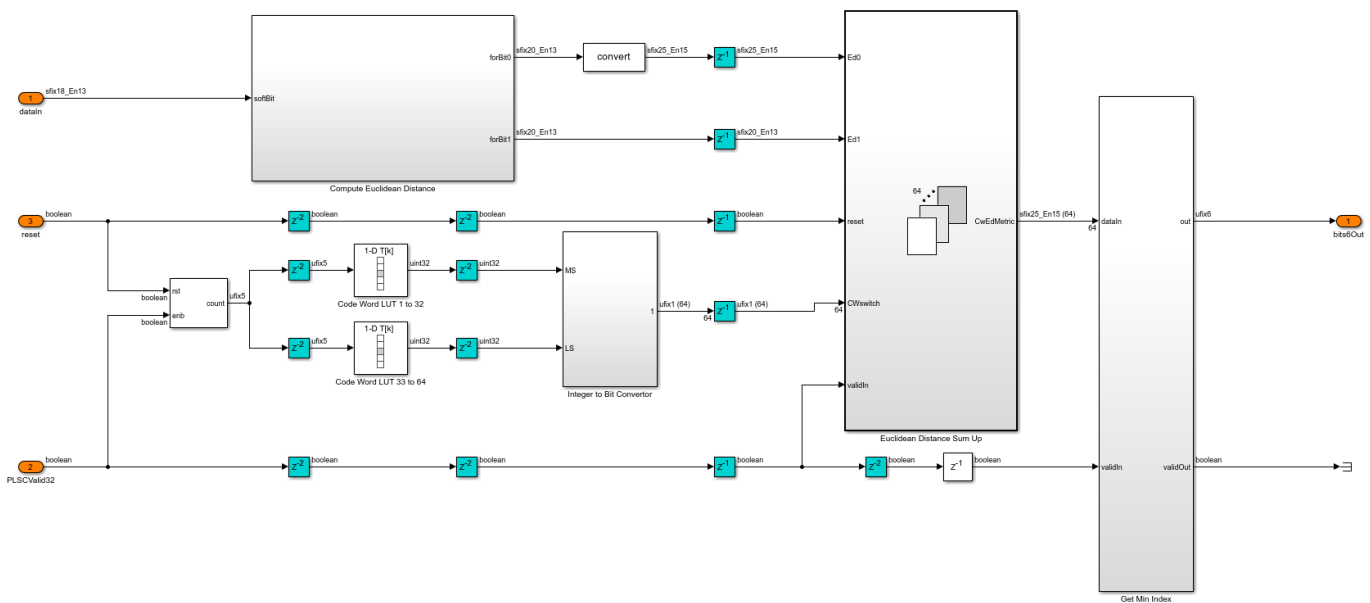
Fine Phase Synchronizer

The `Fine Phase Synchronizer` subsystem is a PLL implementation. Its normalized loop bandwidth is set to 20e-6. The **MODCOD** value decoded from the PL header specifies the modulation type of the symbols in the frame. The `Power Q and Beta Offset Rotation` subsystem raises the QPSK, 8-PSK, 16-APSK, and 32-APSK symbols to a power of Q and rotates the constellation by an angle of beta. The `Phase Error Detector` subsystem computes the phase error from the output of the `Power Q and Beta Offset Rotation` subsystem, as described by equation C.10 in annex C.6.2 of [2]. The phase error is filtered by the loop filter. The filtered output drives the NCO block in the `Direct Digital Synthesis` subsystem to generate the complex exponential sinusoidal samples, which are conjugated and used to correct the phase of the input samples. A reset signal **rstCCFO** resets the loop filter and restarts the estimation process.

This table shows the Q and beta values for the modulated symbols

```
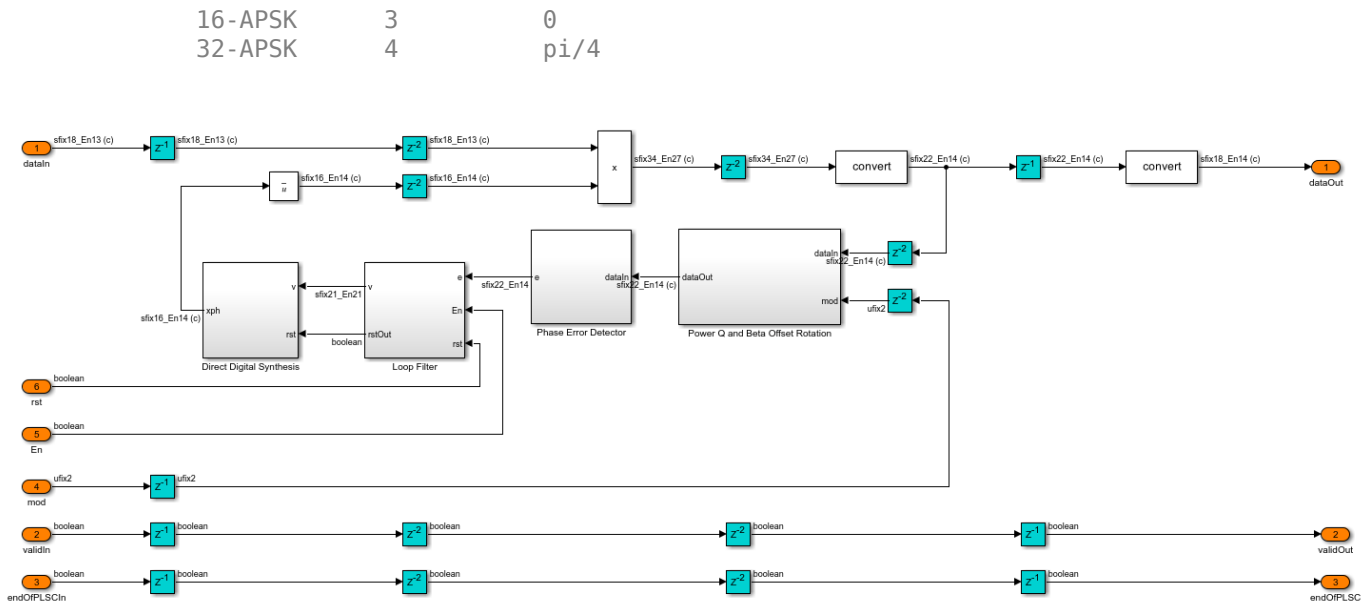Modulation    Q    Beta (in radians)
_____    _    _____

QPSK          1         0
8-PSK         2         pi/4
```

```
16-APSK      3          0
32-APSK      4          pi/4
```


Channel

The `Channel` subsystem introduces the impairments in this table.

Impairment	Description
Fading Factor	Specified in the interval [0.9,1.1]
Additive white Gaussian noise (AWGN)	Specified in Es/N0 in dB
Carrier frequency offset (CFO)	Specified in Hz
Carrier frequency drift (CFD)	Specified in Hz/second
Carrier phase offset (CPO)	Specified in degrees
Sampling clock offset (SCO)	Specified in the interval [0,1)
Phase noise	Specified as Low, Medium, High

This table defines the phase noise mask level in dBc/Hz that the phase noise generator in the `dvbs2hdlPhaseNoise.m` file uses to generate the phase noise and introduce in the transmitter output signal.

Frequency	Low	Medium	High
100 Hz	-73	-59	-25
1 KHz	-83	-77	-50
10 KHz	-93	-88	-73
100 KHz	-112	-94	-85
1 MHz	-128	-104	-103
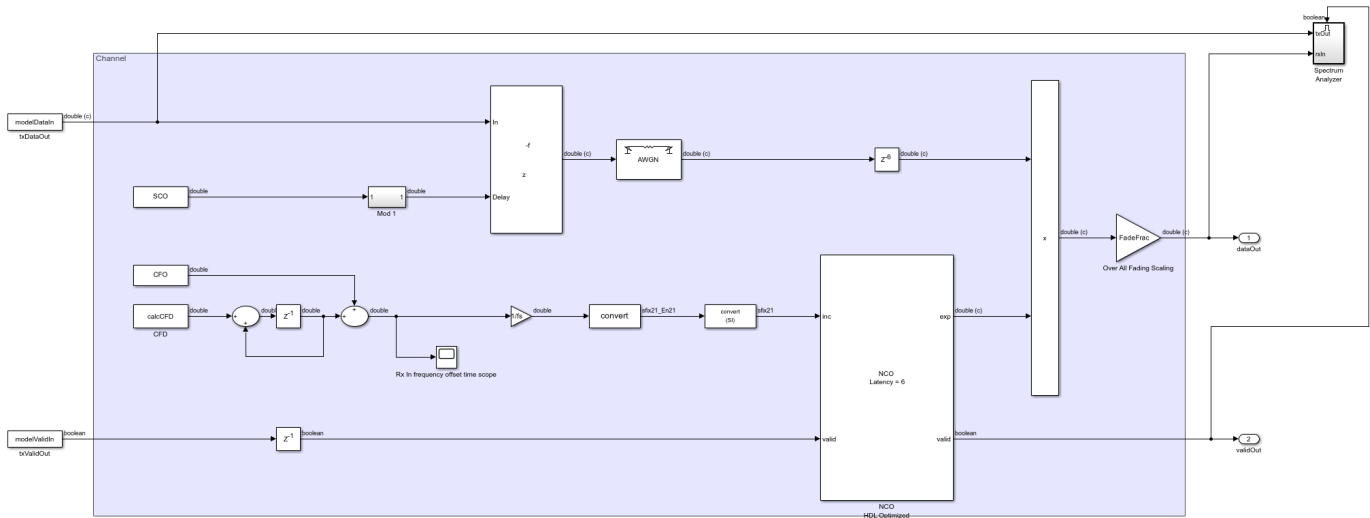
Run the Model

Set the symbol rate, **MODCOD**, **FECFrame** type values, input stream format, user packet length and channel impairments on the `Input Configuration` subsystem mask and run the `dvbs2hdlPLHeaderRecovery.slx` model. Alternatively, to run the model, execute this command at the MATLAB command prompt.

```
sim dvbs2hdlPLHeaderRecovery
```

The **MODCOD** and **FECFrame** type values must be row vectors. Each element of the row vector corresponds to a frame.

Verification and Results

Run the `dvbs2hdlPLHeaderRecovery.slx` model.

```
### Starting serial model reference simulation build.
### Model reference simulation target for dvbs2hdlSyncPLHeaderRecoveryCore is up to date.
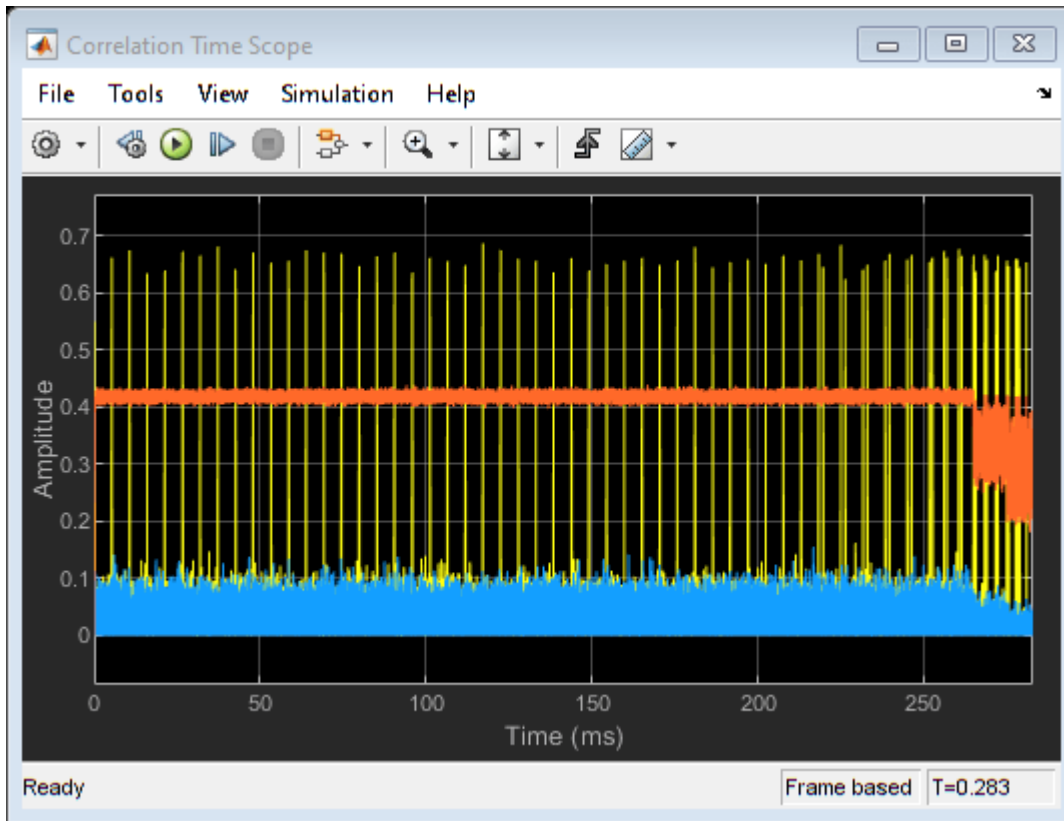
Build Summary

0 of 1 models built (1 models already up to date)
Build duration: 0h 1m 15.237s


Number of frames synced = 68 out of 68
Initial frames not compared = 35
Number of frames lost due to PL header mismatch = 0 out of 33
Number of frames lost due to BB header CRC failure = 0 out of 33
Number of packets errored = 0 out of 641
Number of bits errored = 0 out of 964064
```
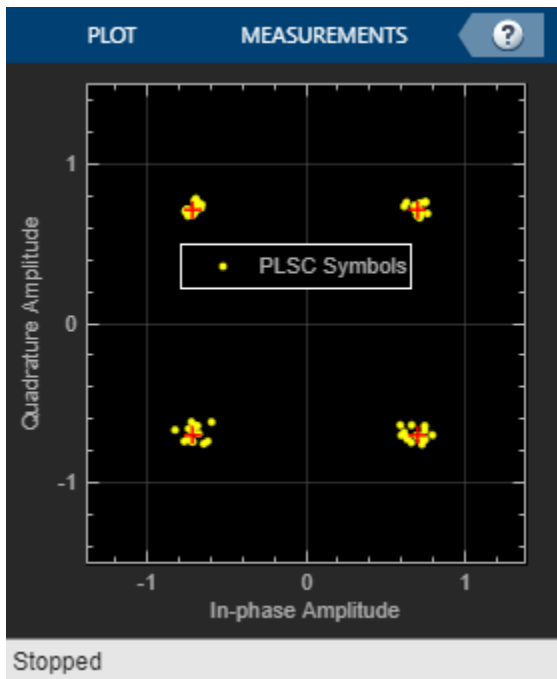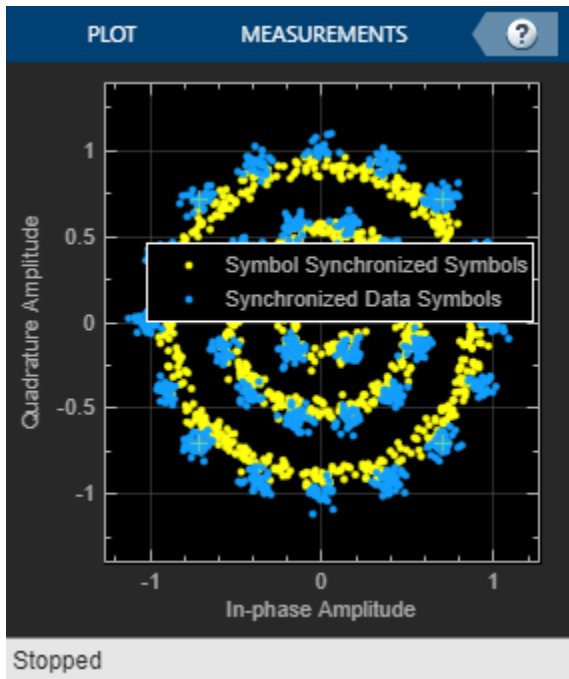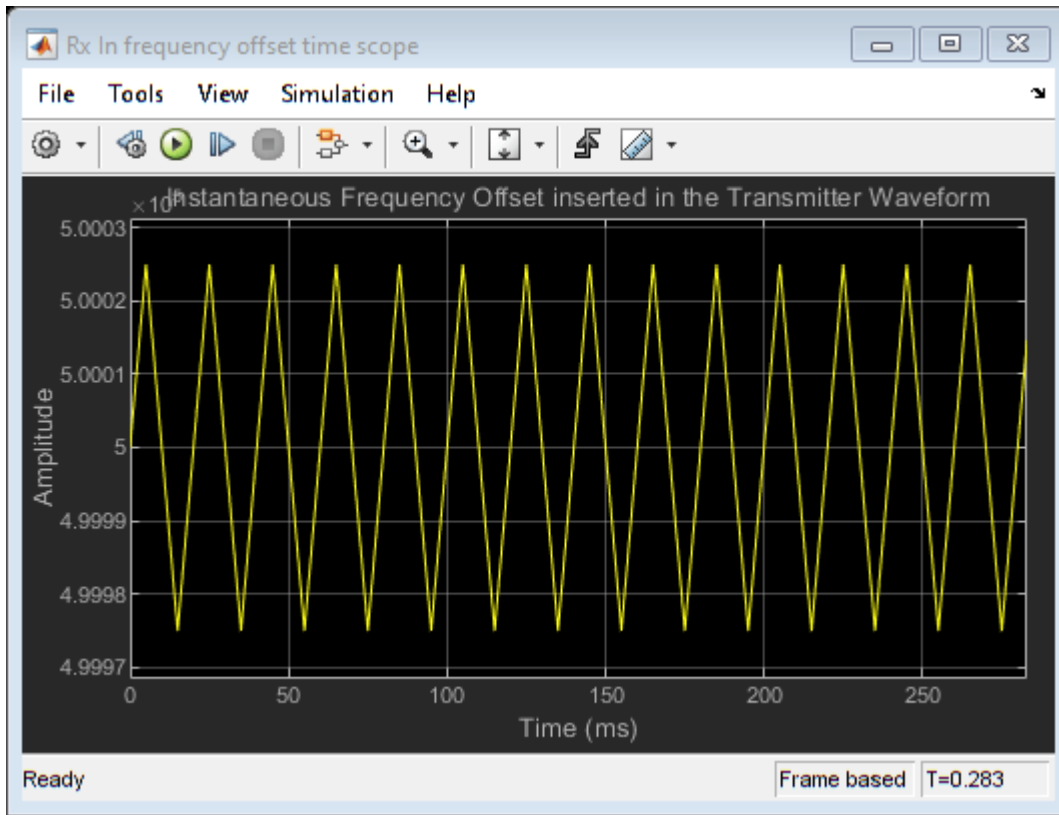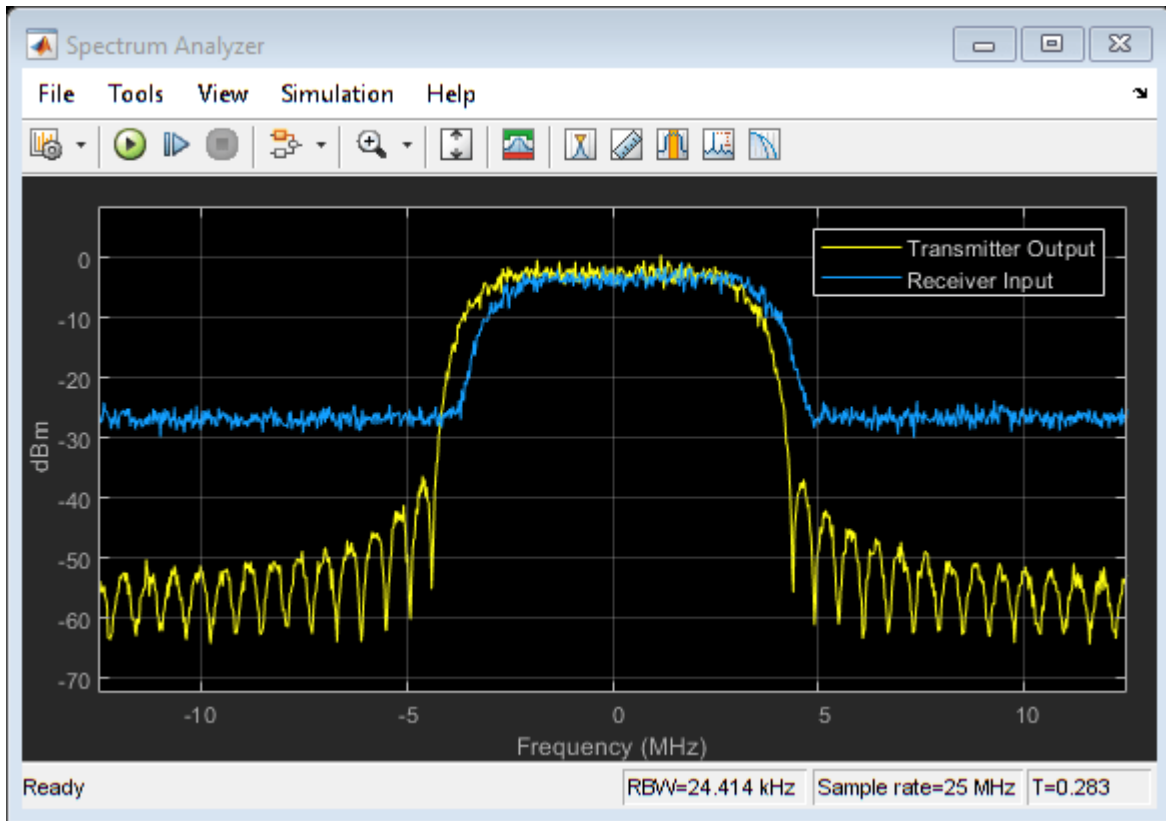
HDL Code Generation

To generate the HDL code for this example, you must have HDL Coder™. Use `makehdl` and `makehdltb` commands to generate HDL code and HDL testbench for the `Synchronization and PL Header Recovery` subsystem. The testbench generation time depends on the simulation time.

The resulting HDL code is synthesized for a Xilinx® Zynq®-7000 ZC706 evaluation board. The post place and route resource utilization are shown in this table. The maximum frequency of operation is 205 MHz.

```
    Resources        Usage
    _____     _____

    Slice LUT        42134
    Slice Registers  63436
    RAMB36           20
    RAMB18           1
    DSP48            248
```

References

1 ETSI Standard EN 302 307-1 V1.4.1(2014-11). *Digital Video Broadcasting (DVB); Second Generation Framing Structure, Channel Coding and Modulation Systems for Broadcasting, Interactive Services, News Gathering and other Broadband Satellite Applications (DVB-S2)*.

2 ETSI Standard TR 102 376-1 V1.2.1(2015-11). *Digital Video Broadcasting (DVB); Implementation Guidelines for the Second Generation System for Broadcasting, Interactive Services, News Gathering and other Broadband Satellite Applications (DVB-S2).*

3 Marco Luise and Ruggero Reggiannini, *Carrier Frequency Recovery in All-Digital Modems for Burst-Mode Transmissions.*

4 Michael Rice, *Digital Communications - A Discrete-Time Approach*, Prentice Hall, April 2008.

See Also

Blocks
Discrete FIR Filter | NCO

Related Examples

- "DVB-S2 HDL Receiver" on page 5-280

DVB-S2 HDL Receiver

This example shows how to implement DVB-S2 receiver using Simulink® blocks that are optimized for HDL code generation and hardware implementation.

This example shows how to model a digital video broadcast satellite second generation (DVB-S2) HDL receiver system by using the "DVB-S2 HDL PL Header Recovery" on page 5-260 example to demodulate, deinterleave, decode using low density parity check (LDPC) and Bose-Chaudhuri-Hocquenghem (BCH) codes, and recover the stream bits.

Model Architecture

This section explains the high-level architecture of the DVB-S2 receiver model. The Synchronization and PLHeader Recovery block extracts the data symbols, estimates noise variance, and decodes physical layer (PL) header information from the *Rx Input Waveform* signal. The Symbol Demodulator block demodulates the data symbols and computes soft bits. The Deinterleaver block deinterleaves and FEC Decoder block decodes the soft bits to extract a *DVB-S2 baseband frame* signal. The Stream Recovery block extracts the BB header information and the output stream bits from the baseband frame.

This block diagram shows the high-level architecture of the model.

File Structure

This example uses four Simulink models, five MATLAB files, and one Simulink data dictionary.

- `dvbs2hdlReceiver.slx` — Top-level Simulink model.
- `dvbs2hdlSyncPLHeaderRecoveryCore.slx` — Model reference that synchronizes time, frequency, and phase, and decode PL header.
- `dvbs2hdlDemodDeinterleaveFecdecodeCore.slx` — Model reference that demodulates the symbols, deinterleaves the demodulated soft bits, and decodes the deinterleaved soft bits using forward error correction (FEC). It discards the frames that does not support the configuration according to [1].
- `dvbs2hdlStreamRecoveryCore.slx` — Model reference that recovers the stream of data bits.
- `getdvbs2LDPCParityMatrices.m` — Download the MAT file that stores LDPC parity check matrices that are used to generate the receiver input waveform.
- `dvbs2hdlRxParameters.m` — Generate parameters for the `dvbs2hdlSyncPLHeaderRecoveryCore.slx` model reference.
- `dvbs2hdlPhaseNoise.m` — Introduce phase noise to the input sequence.

- `dvbs2hdlRxInit.m` — Generate the transmitter waveform and initialize the `dvbs2hdlSyncPLHeaderRecoveryCore.slx` model reference.

- `dvbs2hdlReceiverVerify.m` — Gather PL header parameters and stream recovered bits.

- `dvbs2hdlReceiverData.sldd` — Simulink data dictionary to store bus signal configurations.

System Interface

This figure shows the top-level overview of the `dvbs2hdlReceiver.slx` model.
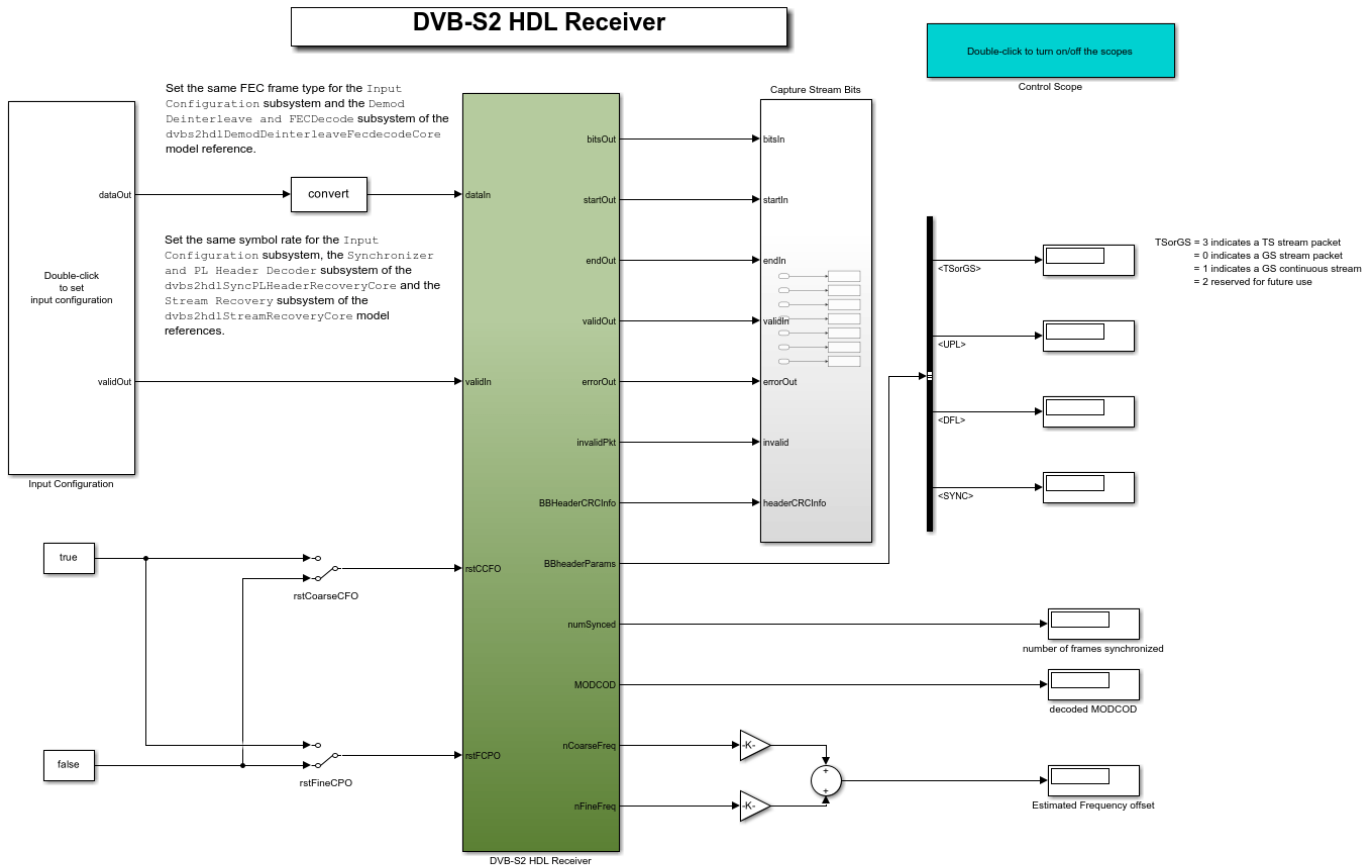
Copyright 2021 The MathWorks, Inc.

Model Inputs

- **dataIn** — Input data, specified as an 18 bit complex data with a sample rate that is four times the symbol rate.

- **validIn** — Control signal to validate the **dataIn**, specified as a Boolean scalar.

- **rstCCFO** — Control signal to reset the coarse frequency compensation loops, specified as a Boolean scalar.

- **rstFCPO** — Control signal to reset the fine phase compensation loops, specified as a Boolean scalar.

Model Outputs:

- **diagBus** — Bus signal with diagnosis information.
- **bitsOut** — Decoded stream bits, returned as a Boolean scalar.
- **startOut** — Control signal for start of **bitsOut** stream bits, returned as a Boolean scalar.
- **endOut** — Control signal for end of **bitsOut** stream bits, returned as a Boolean scalar.
- **validOut** — Control signal to validate the **bitsOut**, returned as a Boolean scalar.
- **errorOut** — Control signal to indicate packet CRC failures. It can be ignored for non-packetized continuous streams.
- **invalidPkt** — Control signal to indicate invalid packets that can be discarded. It can be ignored for non-packetized continuous streams.
- **headerCRCInfo** — Header CRC status, returned as a 2 bit real data. MSB bit high indicates a CRC error, and LSB high indicates when the CRC is considered.
- **numSynced** — Number of frames synchronized, returned as a 32 bit scalar integer
- **MODCOD** — Decoded **MODCOD**, returned as a 5 bit scalar integer.
- **nCoarseFreq** — Estimated normalized (with sample rate) coarse frequency offset, returned as a 21 bit scalar.
- **nFineFreq** — Estimated normalized (with symbol rate) fine frequency offset, returned as a 21 bit scalar.
- **BBHeaderParams** — The following are the list of BB header parameters:
- **TSorGS** — Input stream format, returned as a 2 bit real data.
- **SISorMIS** — Single or multiple input stream input, returned as a Boolean scalar.
- **CCMorACM** — Constant coding modulation (CCM), or adaptive coding modulation (ACM) or variable coding modulation (VCM), returned as a Boolean scalar.
- **RO** — Roll-off factor, returned as a 2 bit real data.
- **UPL** — User packet length (UPL), returned as a 16 bit real data.
- **DFL** — Data field length (DFL), returned as a 16 bit real data.
- **SYNC** — SYNC word, returned as an 8 bit real data.
- **ISSYI** — Input stream synchronization indicator (ISSYI), returned as a Boolean scalar.
- **NPD** — Null packet detection (NPD), returned as a Boolean scalar.
- **MIS_ISI** — Input stream identifier (ISI) for multiple input stream input, returned as an 8 bit real data. For single stream, this is reserved by the standard.
- **SYNCD** — Start location of SYNC word in number of bits from start of data field, returned as a 16 bit real data.

Model Structure

This figure shows the top-level model of the DVB-S2 HDL Receiver subsystem. The subsystem comprises three model references, dvbs2hdlSyncPLHeaderRecoveryCore, dvbs2hdlDemodDeinterleaveFecdecodeCore, and dvbs2hdlStreamRecoveryCore.

dvbs2hdlSyncPLHeaderRecoveryCore — Synchronizes the input receiver waveform, estimates noise variance, and decodes PL header information. For more information, see "DVB-S2 HDL PL Header Recovery" on page 5-260 example.
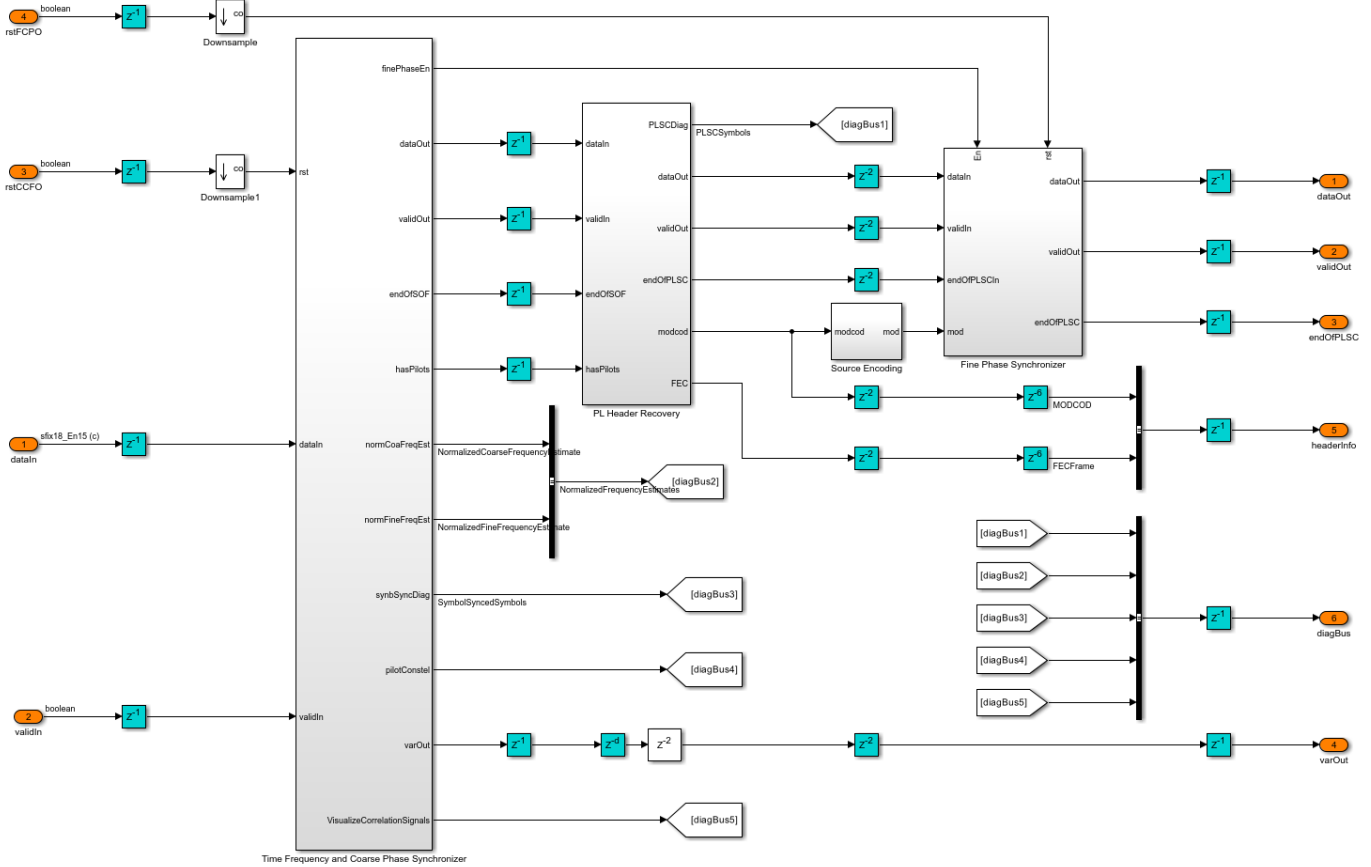
dvbs2hdlDemodDeinterleaveFecdecodeCore — Demodulates the input symbols using DVB-S2 Symbol Demodulator block to extract soft bits, deinterleave the soft bits using `DVB-S2 DeinterLeaver` subsystem, and FEC decodes (LDPC and BCH decodes) the soft bits using the DVB-S2 LDPC Decoder block and the DVB-S2 BCH Decoder block to extract baseband frame (BBFrame). The baseband frame streams into the `dvbs2hdlStreamRecoveryCore` model reference.
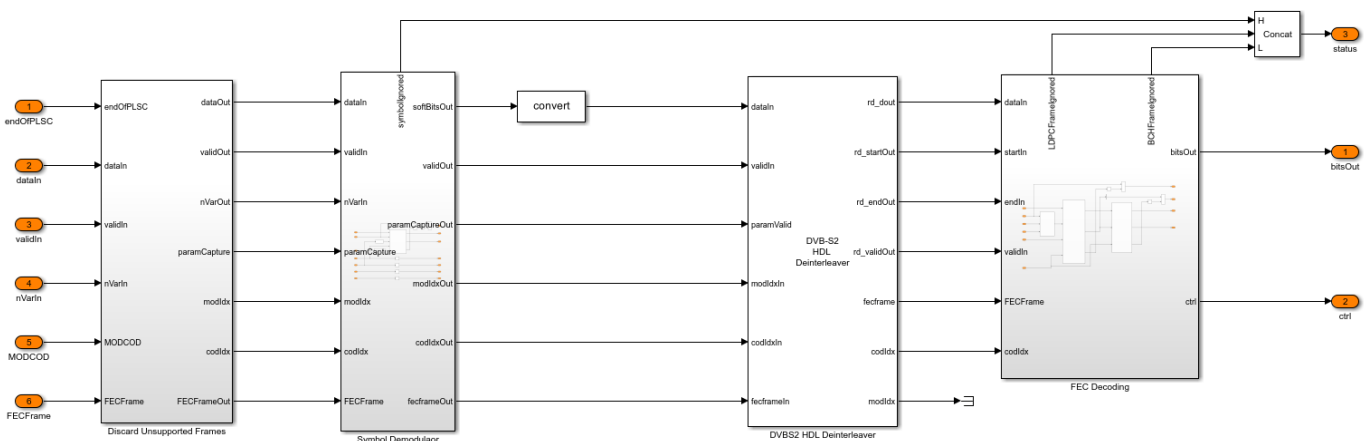
The `DVB-S2 Deinterleaver` subsystem in the `dvbs2hdlDemodDeinterleaveFecdecodeCore` model reference continuously stores the soft bits received from the DVB-S2 Symbol Demodulator

inside the RAM subsystem. The `RAM Address Generator` subsystem generates the read and write logic to the RAM for deinterleaving.
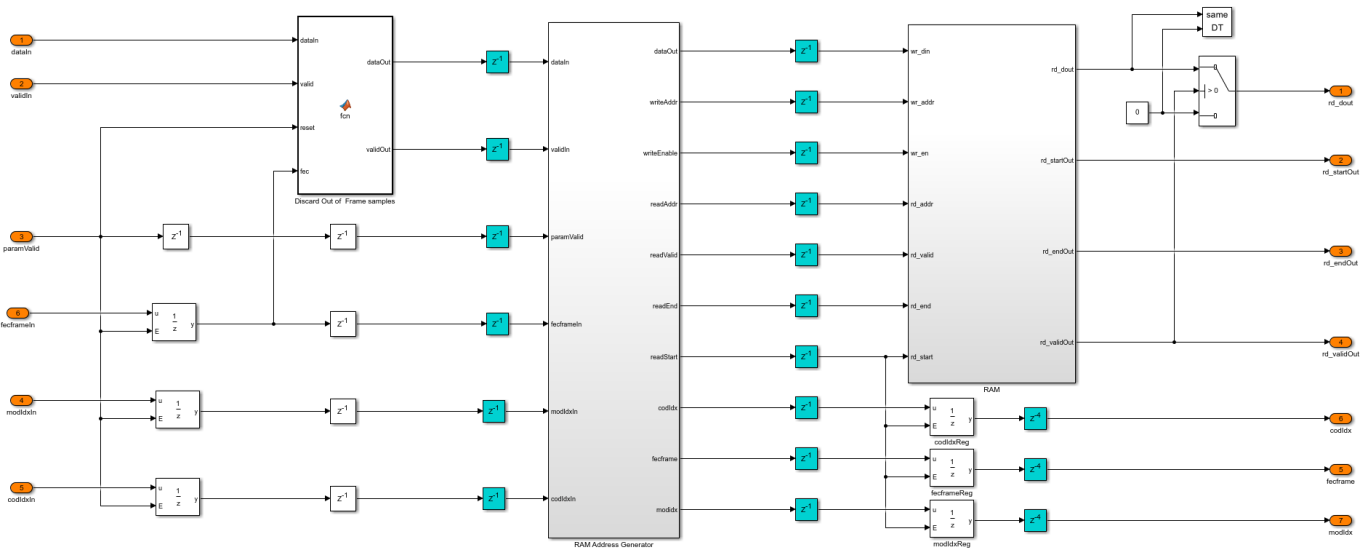
In the `RAM Address Generator` subsystem, the `nRows and nColumns` subsystem stores the number of rows and columns of each of the possible configuration in look-up tables (LUT). Based on the PL header parameters, the number of rows and columns is decided for deinterleaving. The `Read Offset Address` subsystem generates the deinterleaver indices of each frame as an offset address. The `Generate and Add Read Base Address` subsystem adds the offset address with a base read address to get the actual address of the soft-bit stored in the RAM. The `Parameter Store FIFO` subsystem stores the PL header parameters and reads these parameters in synchronous with start of each frame.
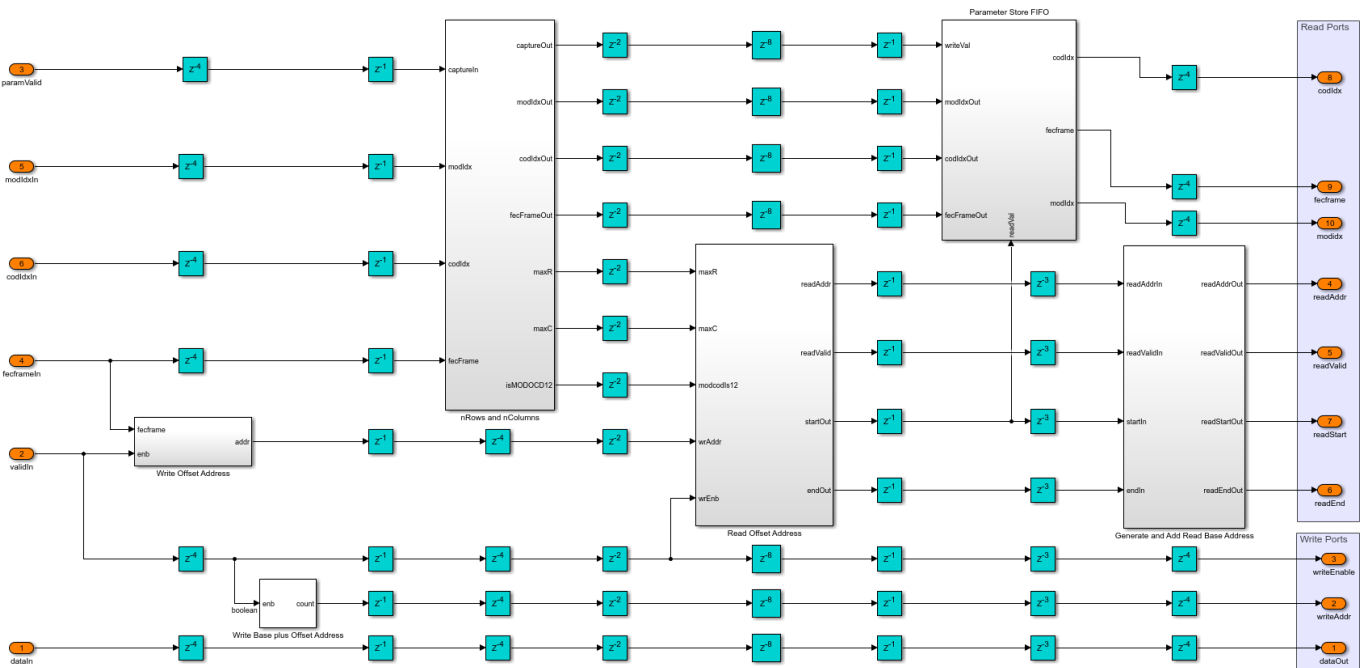
This table shows the rows and columns considered for deinterleaving for each of the configurations.

Modulation	Rows (Normal)	Rows (Short)	Columns
QPSK	64800	16200	1
8-PSK	21600	5400	3
16-APSK	16200	4050	4
32-APSK	12960	3240	5

dvbs2hdlStreamRecoveryCore — Decodes the BB header and recovers the stream bits. The BB Descrambler subsystem descrambles the baseband frame. The BB Demultiplexer subsystem demultiplexes the descrambled frame into BB header and data bits. The BB Header CRC Check subsystem uses the General CRC Syndrome Detector HDL Optimized block to check the CRC status and discards the baseband frames that fails CRC check. The BB Decoder subsystem extracts the BB header information and the data field. For packetized stream of bits, the control signals are generated to indicate the start, end, and validity of bits for each packet. The General CRC Syndrome Detector HDL Optimized block checks the CRC status of each packet. For continuous stream of bits, the data field is passed on to the output.

Run the Model

Set the symbol rate, MODCOD, FECFrame type values, input stream format, user packet length and channel impairments on the mask of the Input Configuration subsystem and run the dvbs2hdlReceiver model. Alternatively, to run the model, execute this command at the MATLAB command window.

```
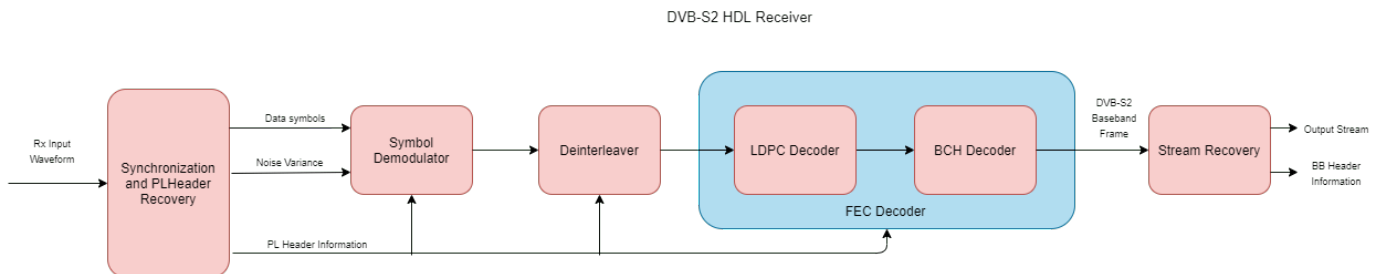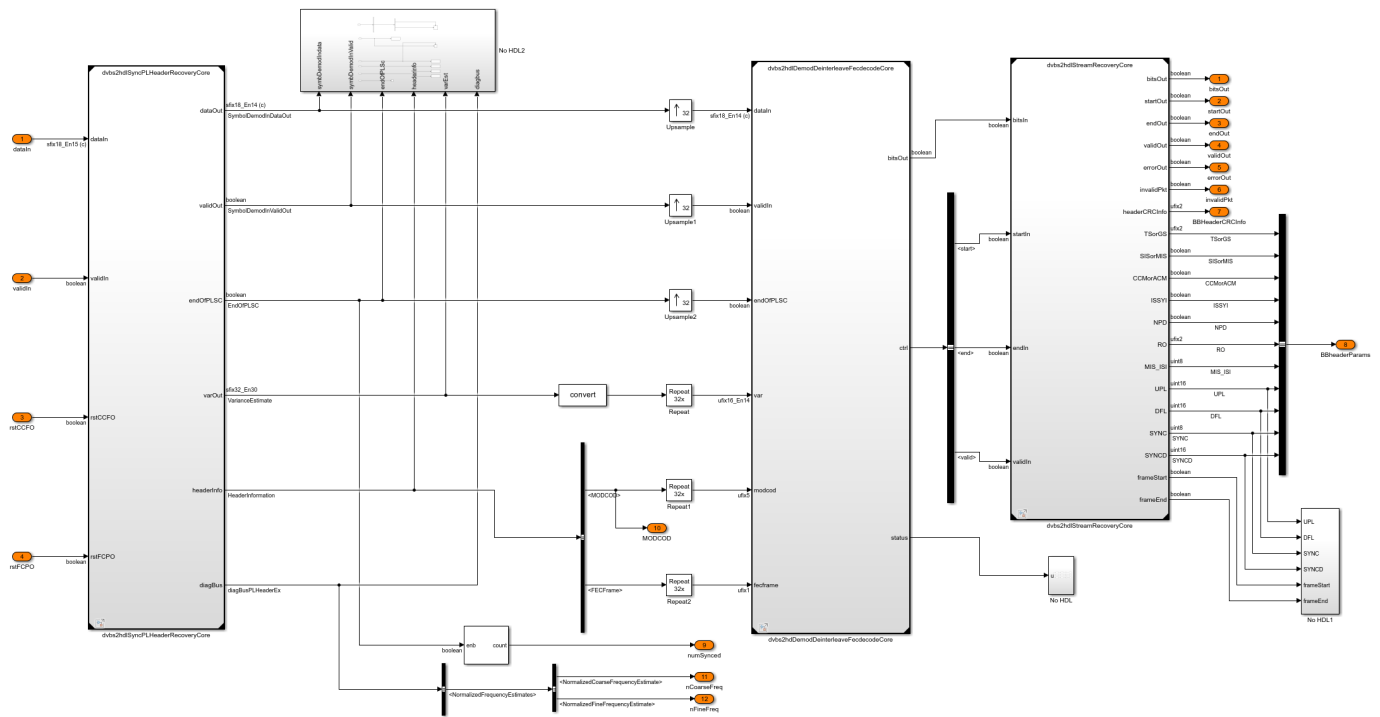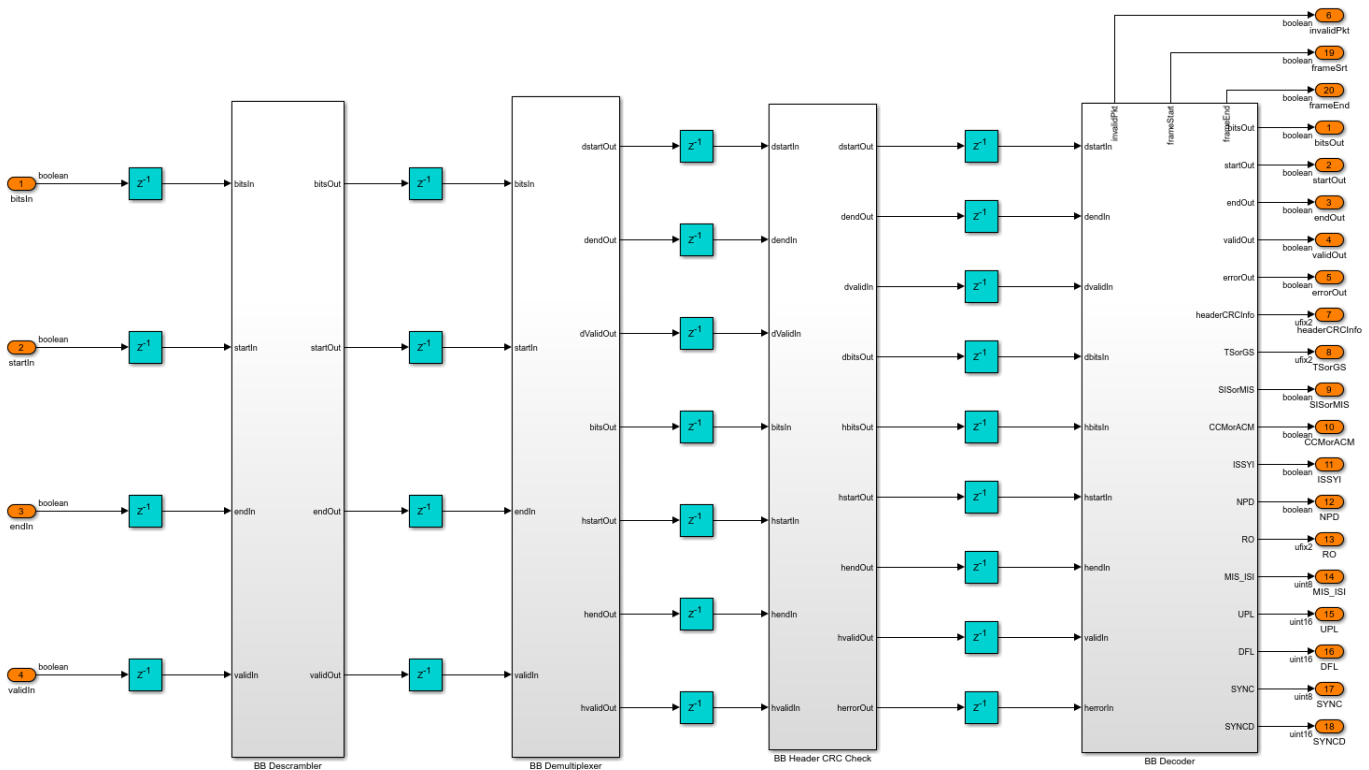sim dvbs2hdlReceiver
```

The MODCOD value must be a row vector. Each element of the row vector corresponds to a frame.

Note: Use QPSK modulated frames initially to achieve time frequency and phase synchronization.

Verification and Results

Run the `dvbs2hdlReceiver.slx` model. The model utilizes 120 short QPSK frames for synchronization.

```
### Starting serial model reference simulation build.
### Successfully updated the model reference simulation target for: dvbs2hdlDemodDeinterleaveFec
### Successfully updated the model reference simulation target for: dvbs2hdlStreamRecoveryCore
### Successfully updated the model reference simulation target for: dvbs2hdlSyncPLHeaderRecoveryC

Build Summary

Simulation targets built:

Model                                   Action                     Rebuild Reason
============================================================================================
dvbs2hdlDemodDeinterleaveFecdecodeCore  Code generated and compiled.  dvbs2hdlDemodDeinterleaveFe
dvbs2hdlStreamRecoveryCore              Code generated and compiled.  dvbs2hdlStreamRecoveryCore_
dvbs2hdlSyncPLHeaderRecoveryCore        Code generated and compiled.  dvbs2hdlSyncPLHeaderRecover

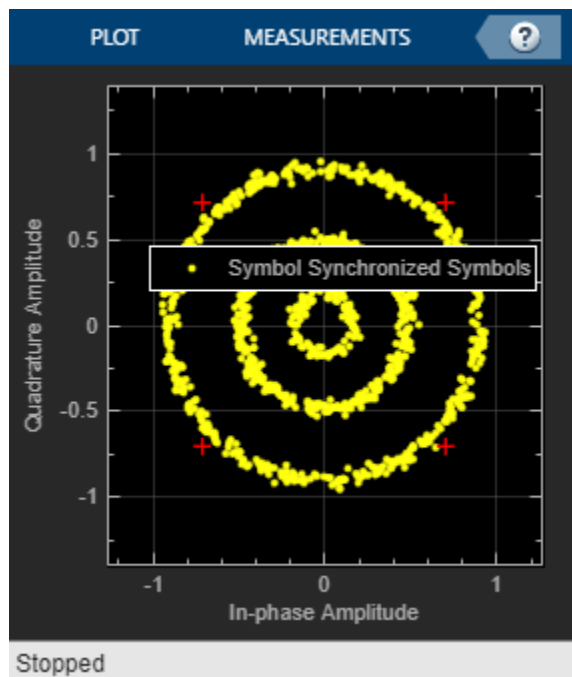3 of 3 models built (0 models already up to date)
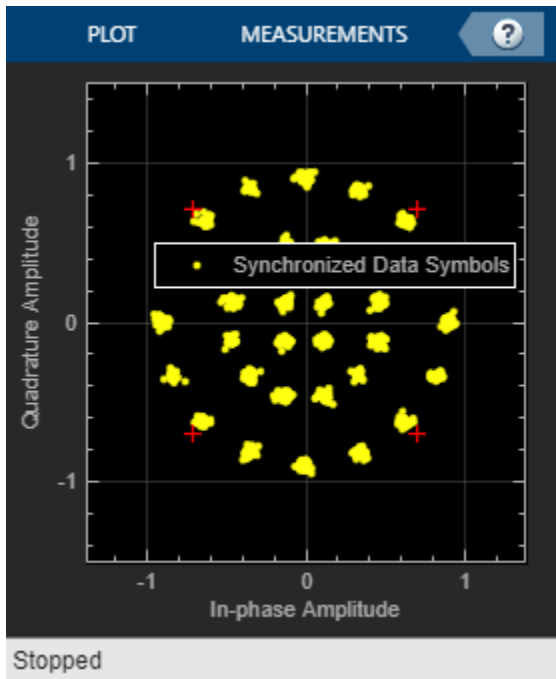Build duration: 0h 7m 7.8594s


Number of frames synced = 124 out of 124
Initial frames not compared = 120
Number of frames lost due to BB Header CRC failure = 0 out of 4
Number of packets lost due to packet CRC failure = 0 out of 27
```

HDL Code Generation

To generate the HDL code for this example, you must have HDL Coder™. Use `makehdl` and `makehdltb` commands to generate HDL code and HDL testbench for the `DVB-S2 HDL Receiver` subsystem. The testbench generation time depends on the simulation time.

The resulting HDL code is synthesized for a Xilinx® Zynq® UltraScale+ RFSoC ZCU111 board. The post place and route resource utilization are shown in this table. The maximum frequency of operation is 181 MHz.

```
    Resources       Usage
  _____     _____

  CLB LUT           84262
  CLB Registers     98936
  RAMB36            821
  RAMB18            1
  DSP48             302
```

References

1 ETSI Standard EN 302 307-1 V1.4.1(2014-11). *Digital Video Broadcasting (DVB); Second Generation Framing Structure, Channel Coding and Modulation Systems for Broadcasting, Interactive Services, News Gathering and other Broadband Satellite Applications (DVB-S2)*.

2 ETSI Standard TR 102 376-1 V1.2.1(2015-11). *Digital Video Broadcasting (DVB); Implementation Guidelines for the Second Generation System for Broadcasting, Interactive Services, News Gathering and other Broadband Satellite Applications (DVB-S2).*

See Also

Blocks
DVB-S2 BCH Decoder | DVB-S2 LDPC Decoder | DVB-S2 Symbol Demodulator | General CRC Generator HDL Optimized | General CRC Syndrome Detector HDL Optimized

Related Examples

DVB-S2 HDL Transmitter

This example shows how to implement a digital video broadcast satellite second generation (DVB-S2) transmitter using Simulink® blocks that are optimized for HDL code generation and hardware implementation.

From this example, you can generate a DVB-S2 transmitter waveform using these steps:

1. Generate a baseband frame (BBFRAME).
2. Encode using Bose-Chaudhuri-Hocquenghem (BCH) and low-density parity-check (LDPC) codes.
3. Interleave, modulate, and generate a physical layer frame (PLFRAME).
4. Insert dummy frames.
5. Pulse shape the symbols in frames using a root raised cosine (RRC) filter.

Model Architecture

The BBFRAME Generator block generates BBFRAME from the user packets. The FECFRAME Generator block encodes the BBFRAME using BCH and LDPC coding, and interleaves the encoded frame to generate an FECFRAME. The Symbol Modulator block generates the xFECFRAME by mapping the FECFRAME bits to modulation symbols. The PLFRAME Generator block generates the PLHEADER, pilot symbols, and dummy frames, and multiplexes them with the xFECFRAME bits. The PL Data Scrambler block scrambles the multiplexer output to generate a PLFRAME. The RRC Filtering block pulse shapes the PLFRAME symbols to generate the transmitter waveform.

DVB-S2 HDL Transmitter
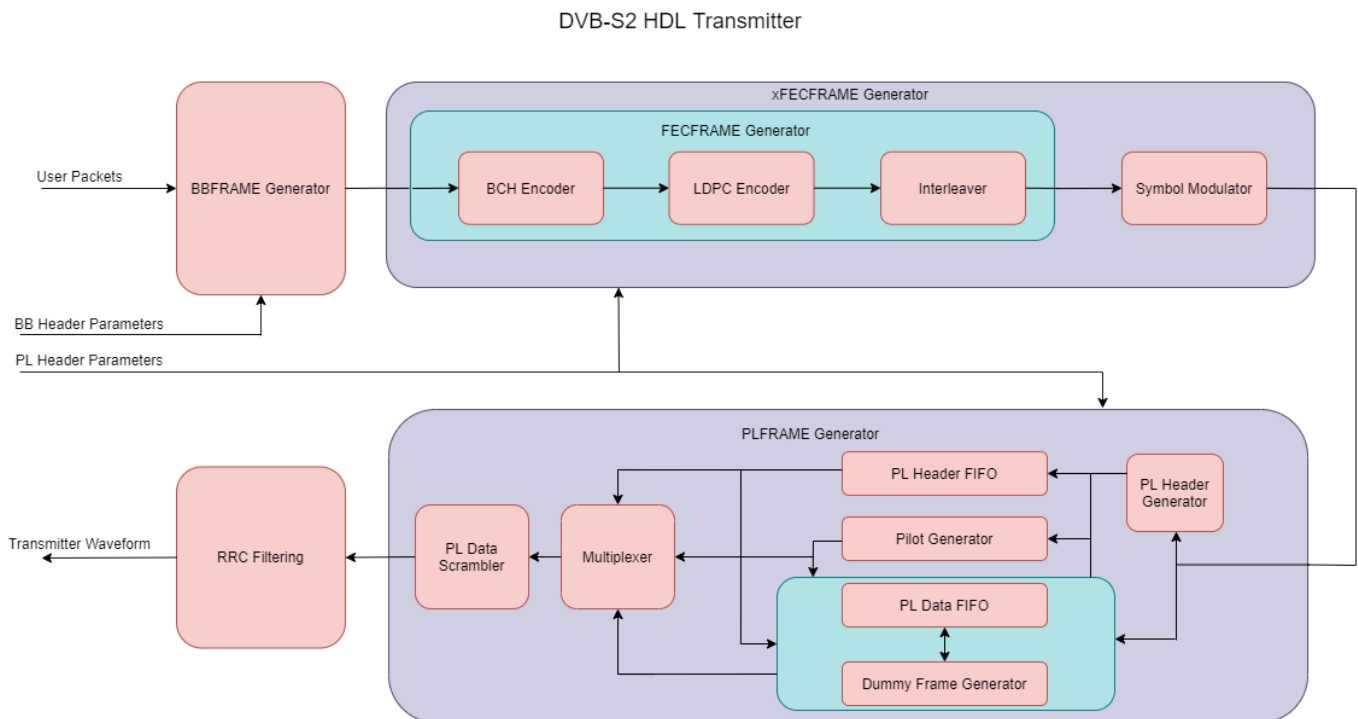
This figure shows the transmitter frame structure.
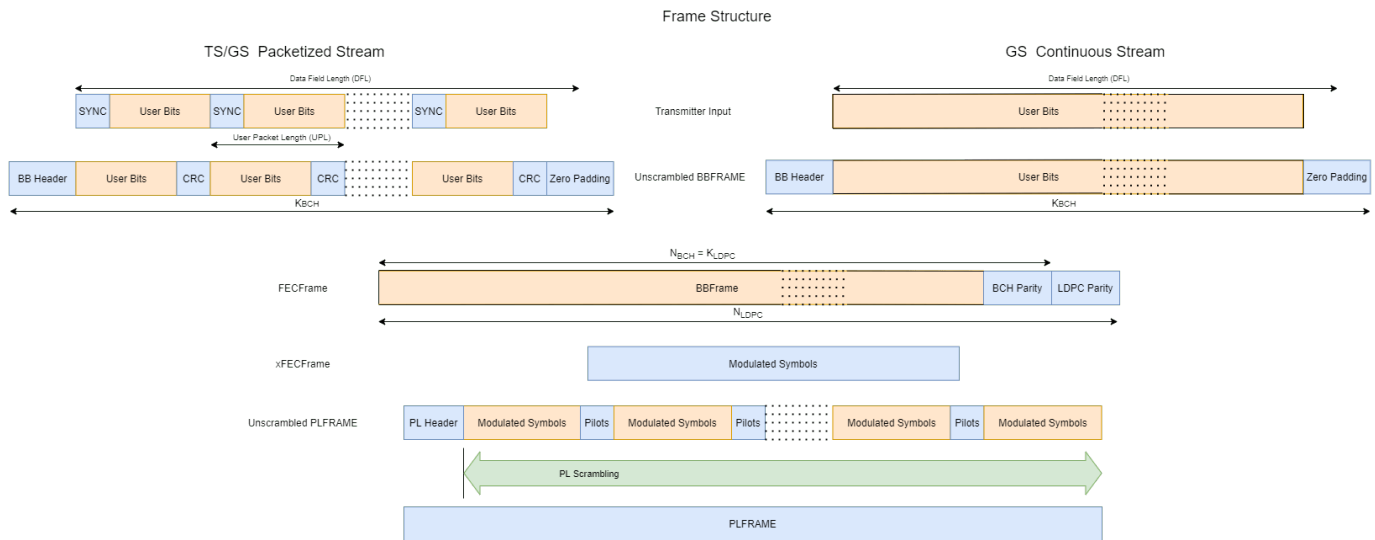
Frame Structure

File Structure

This example uses supporting files.

- `dvbs2hdlTransmitter` — Model for DVB-S2 HDL transmitter.
- `dvbs2hdlTransmitterCore` — Model reference for the transmitter design.
- `dvbs2hdlTxParameters` — Functions that generates parameters for the `dvbs2hdlTransmitterCore` model.
- `dvbs2hdlTxInit` — Script that initializes the `dvbs2hdlTransmitter` model.
- `dvbs2hdlTransmitterVerify` — Script that generates reference transmitter waveform using `dvbs2WaveformGenerator` (Satellite Communications Toolbox) function and compares the reference waveform with the simulated transmitter output.

System Interface

The figure shows the top-level overview of the `dvbs2hdlTransmitter` model.
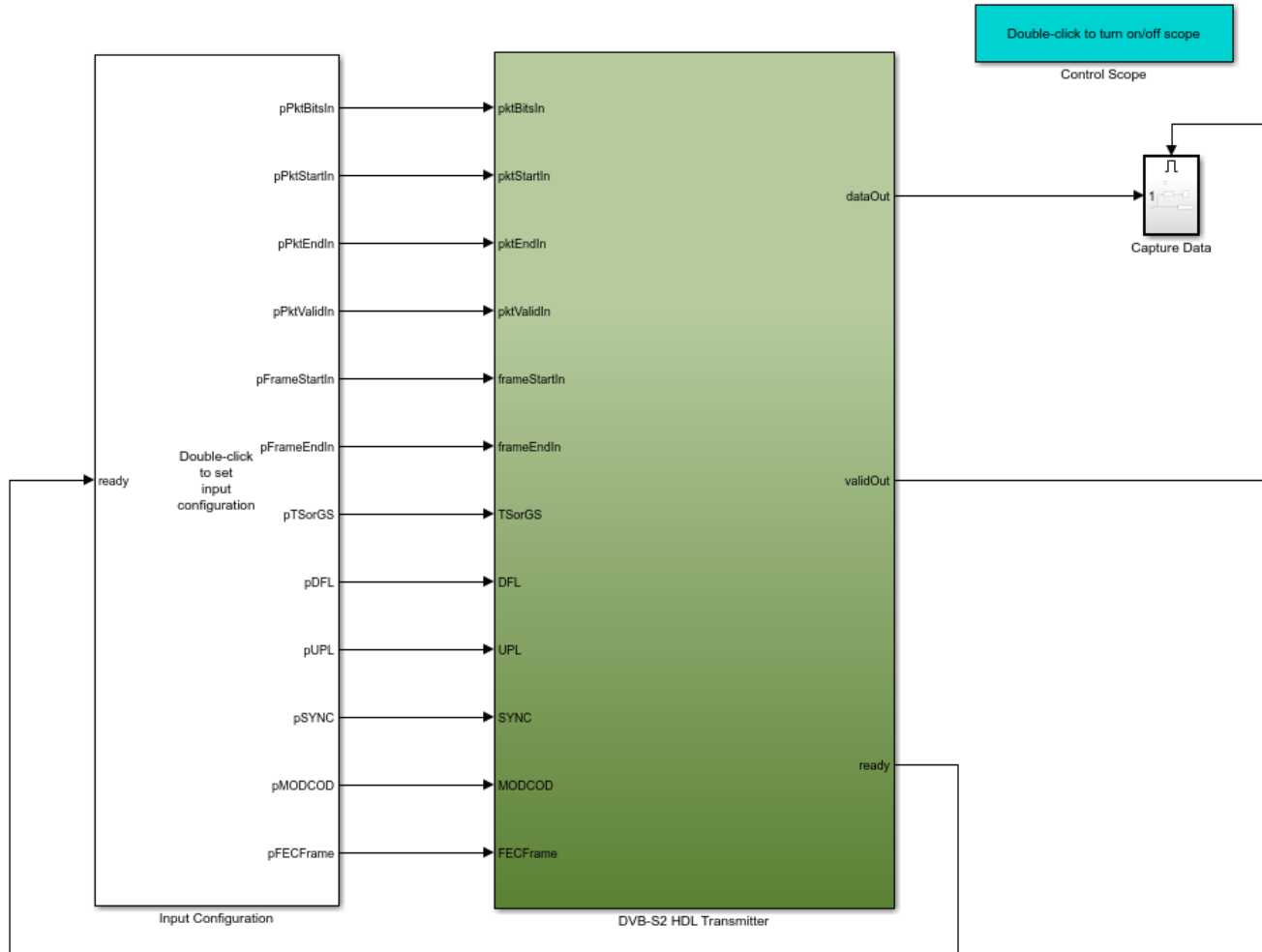
Copyright 2022 The Mathworks, Inc.

Model Inputs

- **pktBitsIn** — Input packet bits, specified as a Boolean scalar.
- **pktStartIn** — Control signal indicating the start of each packet, specified as a Boolean scalar.
- **pktEndIn** — Control signal indicating the end of each packet, specified as a Boolean scalar.
- **pktValidIn** — Control signal indicating whether the **pktBitsIn** is valid, specified as a Boolean scalar.
- **frameStartIn** — Control signal indicating the start of each frame, specified as a Boolean scalar.
- **frameEndIn** — Control signal indicating the end of each frame, specified as a Boolean scalar.
- **TSorGS** — Input stream format, specified as a 2 bit unsigned real integer.

- **DFL** — Data field length (DFL), specified as a 16 bit unsigned real integer.
- **UPL** — User packet length (UPL), specified as a 16 bit unsigned real integer.
- **SYNC** — SYNC word, specified as a 8 bit unsigned real integer.
- **MODCOD** — MODCOD, specified as a 5 bit unsigned real integer.
- **FECFrame** — FECFrame type, specified as a Boolean scalar.

Model Outputs:

- **dataOut** — Transmitter output, returned as a 18 bit complex scalar.
- **validOut** — Control signal indicating whether the **dataOut** is valid, returned as a Boolean scalar.
- **ready** — Control signal indicating whether the transmitter is ready for the input, returned as a Boolean scalar.

Model Structure

This figure shows the structure of the DVB-S2 HDL transmitter subsystem. The subsystem comprises `BB Frame Generator`, `FEC Encoder`, `DVB-S2 HDL Interleaver`, `Symbol Modulator`, `PL Frame Generator`, `PL Data Scrambler`, and `RRC Transmit Filter` subsystems.
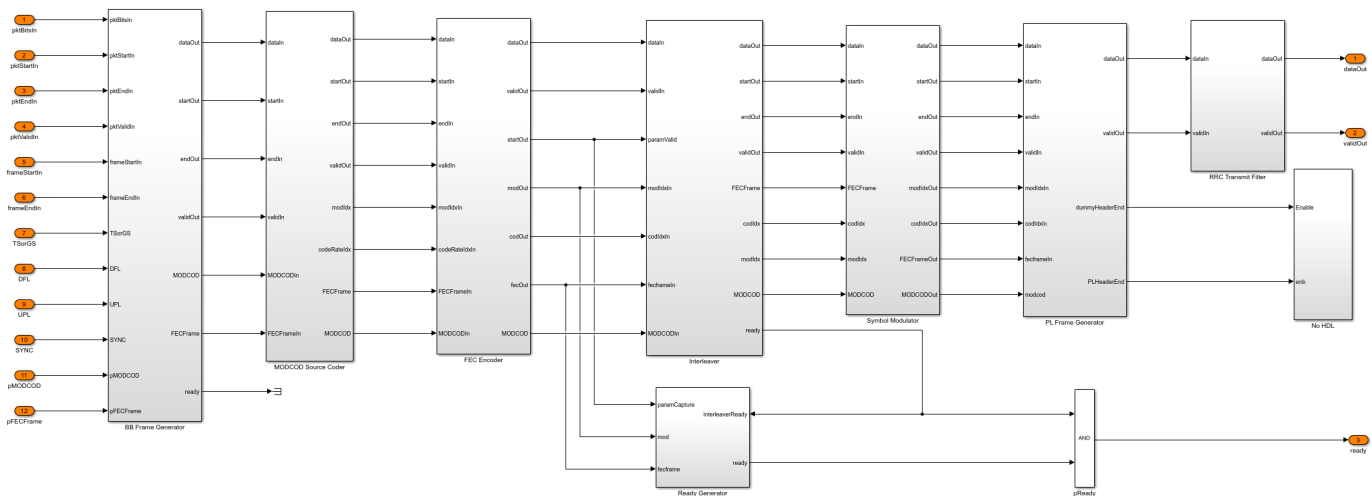
BB Frame Generator

The `BB Frame Generator` subsystem comprises `BB Header and Data CRC Generator`, `Data Store FIFO`, `Multiplexer`, and `BB Scrambler` subsystems. The `BB Header and Data CRC Generator` subsystem generates BB header, discards the SYNC bits, appends CRC bits to each packet using the General CRC Generator HDL Optimized block, and appends padding bits to the data field for each frame. The `Data Store FIFO` subsystem stores the data field of each frame in a RAM buffer and reads it out after the BB header. The `Multiplexer` subsystem multiplexes the BB header and data field. The `BB Scrambler` subsystem scrambles the BB header and data field to generate a BBFRAME.
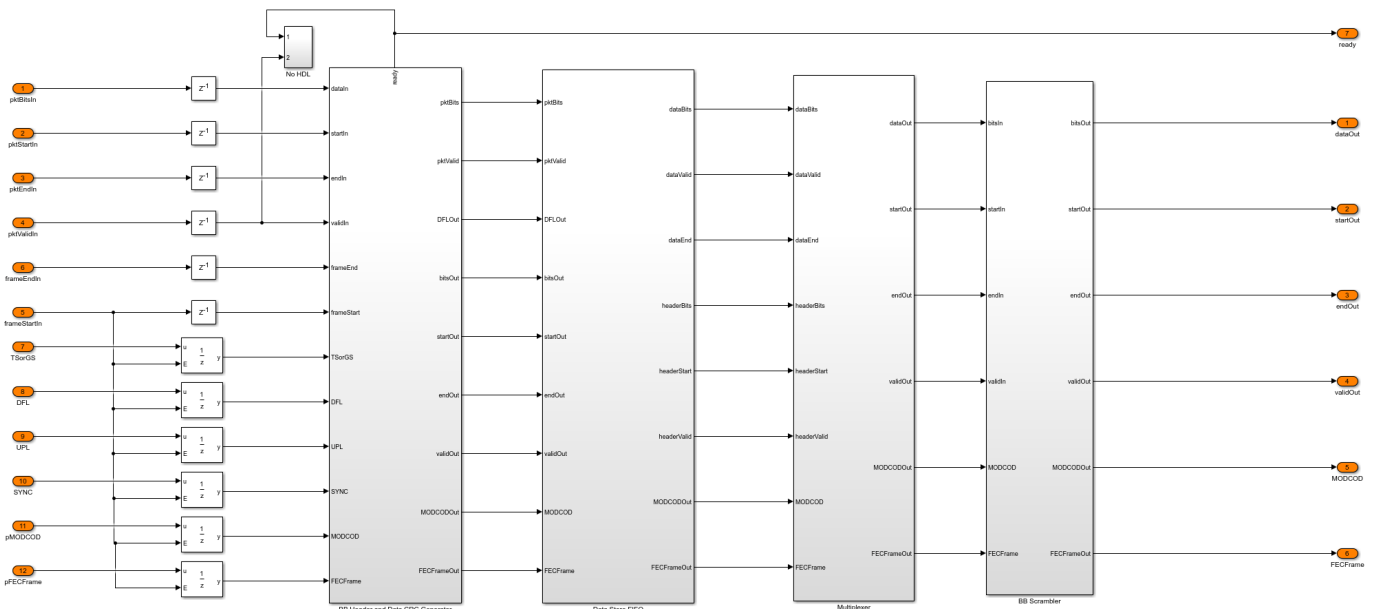
FEC Encoder

The `FEC Encoder` subsystem encodes the input bits with `BCH Encoder` subsystem followed by `LDPC Encoder` subsystem. For more information, see "DVB-S2 HDL BCH Encoder" on page 4-83 and "DVB-S2 HDL LDPC Encoder" on page 4-64 examples.
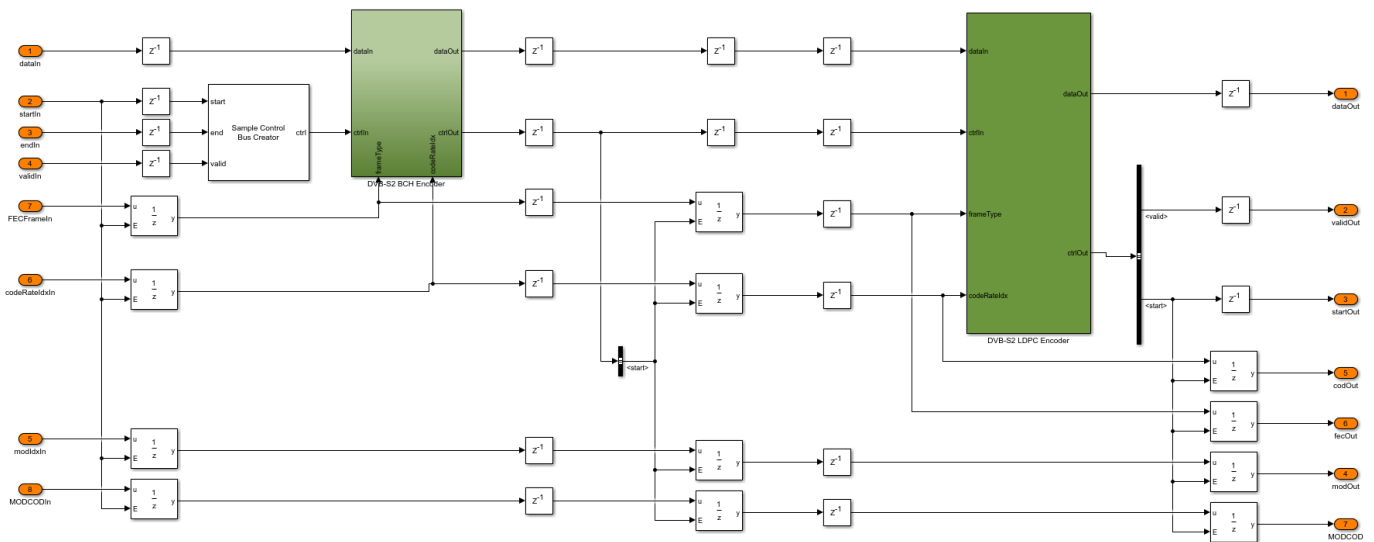
DVB-S2 HDL Interleaver

The `DVB-S2 HDL Interleaver` subsystem stores encoded bits from the `LDPC Encoder` subsystem inside the `RAM` subsystem. The `RAM Address Generator` subsystem generates read and write addresses for the RAM for interleaving.
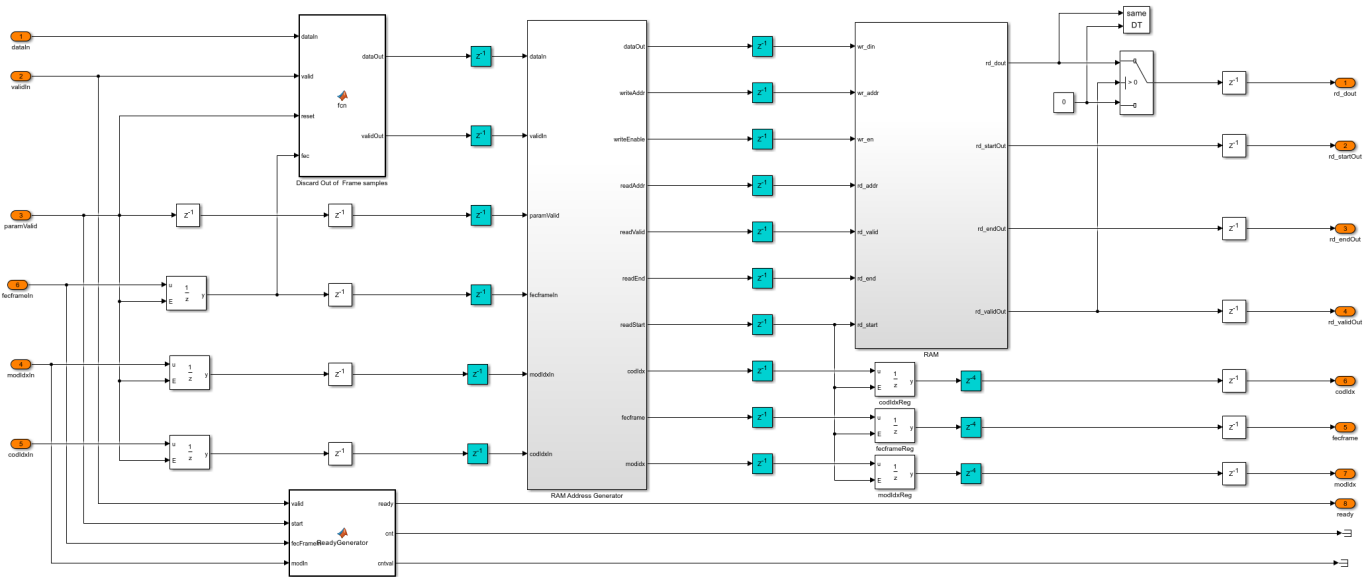
In the `RAM Address Generator` subsystem, the `nRows and nColumns` subsystem stores the number of rows and columns of each possible configuration in lookup tables (LUT). Based on the **MODCOD** and **FECFrame** parameters, the subsystem determines the number of rows and columns for interleaving. The `Read Offset Address` subsystem generates the interleaver indices of each frame as an offset address. The subsystem reads only two bits are read for QPSK, three bits for 8-PSK, four bits for 16-APSK, and five bits for 32-APSK every eight time steps. This process ensures that each symbol covers eight time steps of the interleaver, which is equal to one symbol duration. The `Generate and Add Read Base Address` subsystem adds the offset address with a base read address to get the address of the bits that are stored in the RAM. The `Parameter Store FIFO` subsystem stores the parameters and reads them at the start of each frame.
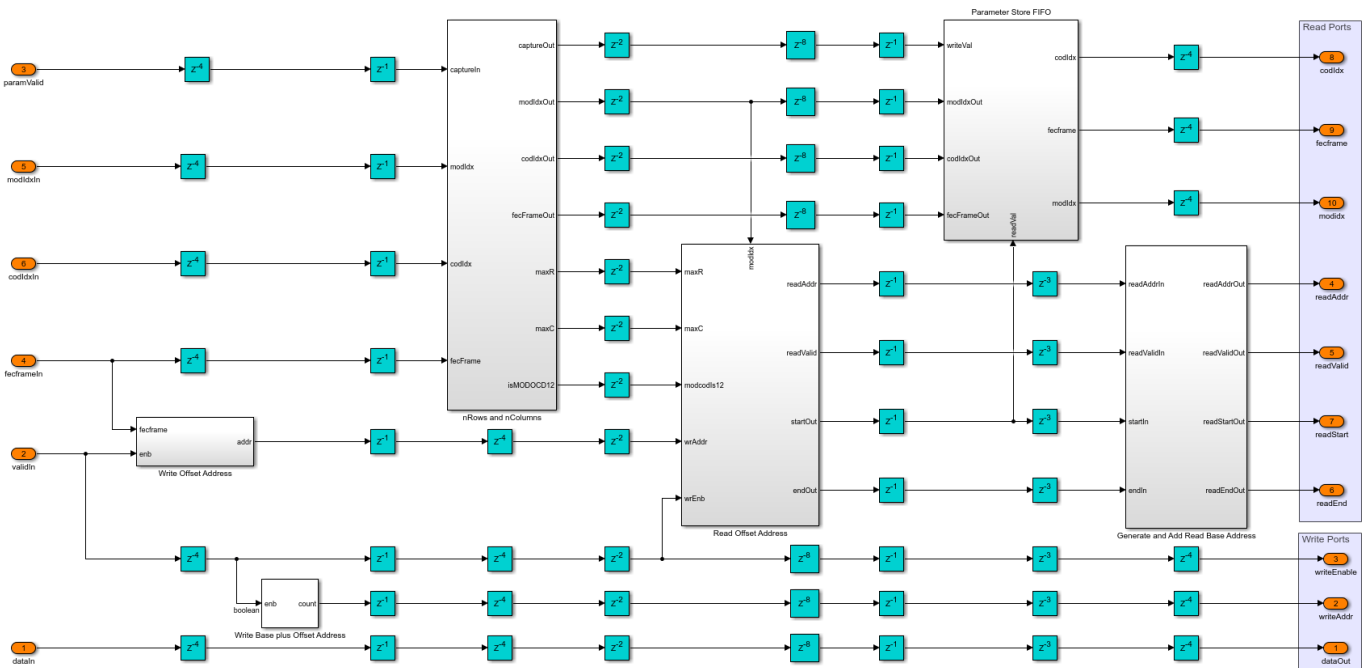
This table shows the rows and columns for deinterleaving in each of the configurations.

Modulation	Rows (Normal)	Rows (Short)	Columns
QPSK	64800	16200	1
8-PSK	21600	5400	3
16-APSK	16200	4050	4
32-APSK	12960	3240	5

Symbol Modulator

The `Symbol Modulator` subsystem comprises DVB-S2 Symbol Modulator block that maps the input bits to the corresponding modulation symbols.
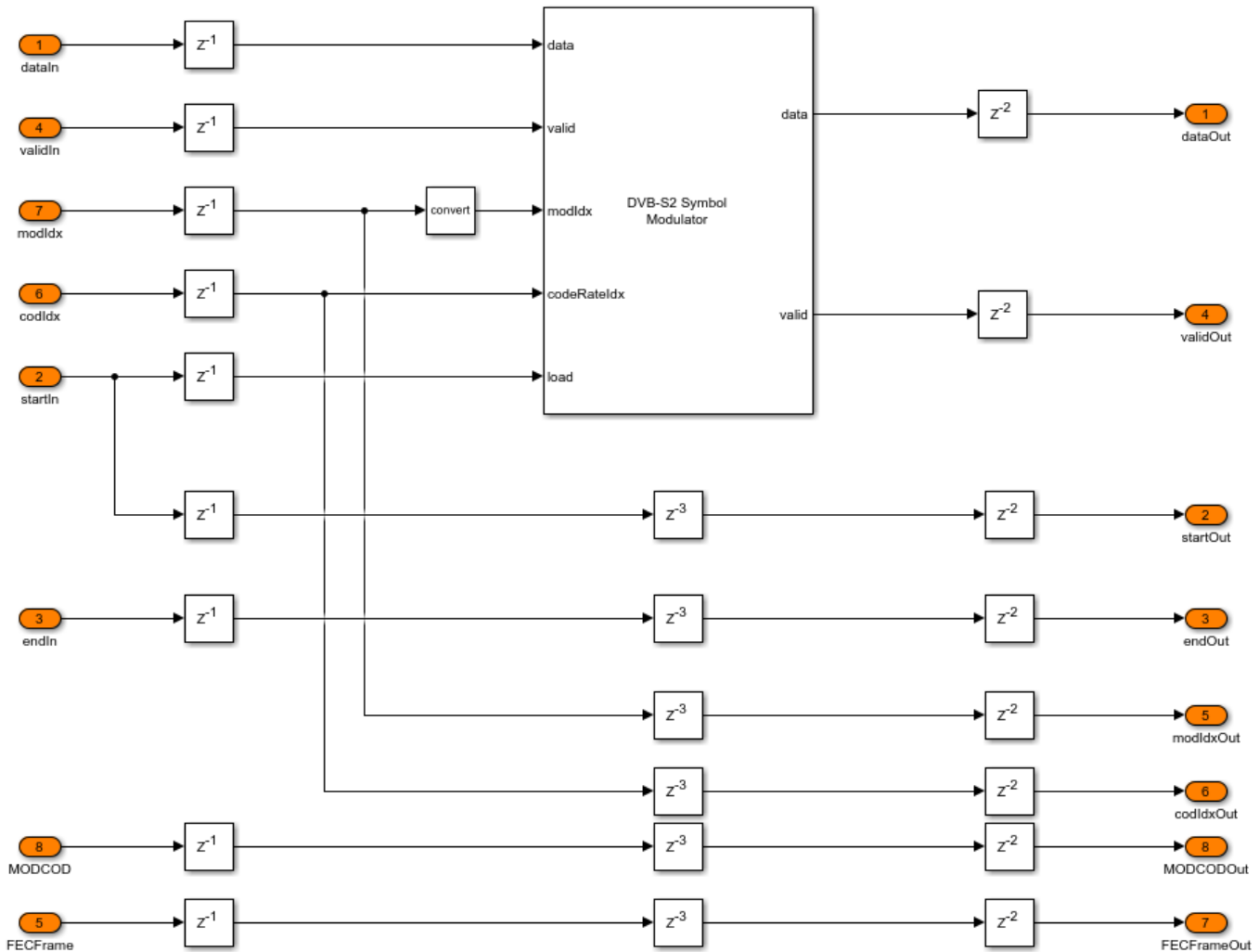
PL Frame Generator

The `PL Frame Generator` subsystem stores the modulated symbols in the `RAM FIFO` subsystem. The `PL Header Generator` subsystem generates the PLHEADER for each frame and stores it in the

RAM FIFO subsystem. When symbols corresponding to a frame are ready in the RAM, the PL Frame Generator subsystem outputs the PLHEADER, pilot symbols, and data symbols according to the frame structure specified in [1]. When the symbols corresponding to a frame are not ready, the Dummy Frame Generator subsystem inside the RAM FIFO subsystem outputs dummy frames. The Multiplexer subsystem multiplexes the PLHEADER, pilot symbols, data symbols, and dummy frames to generate a unscrambled PLFRAME. The PL Data Scrambler subsystem scrambles the data field excluding the PLHEADER. The scrambling starts after the PLHEADER of each frame and continues until the end of the frame.
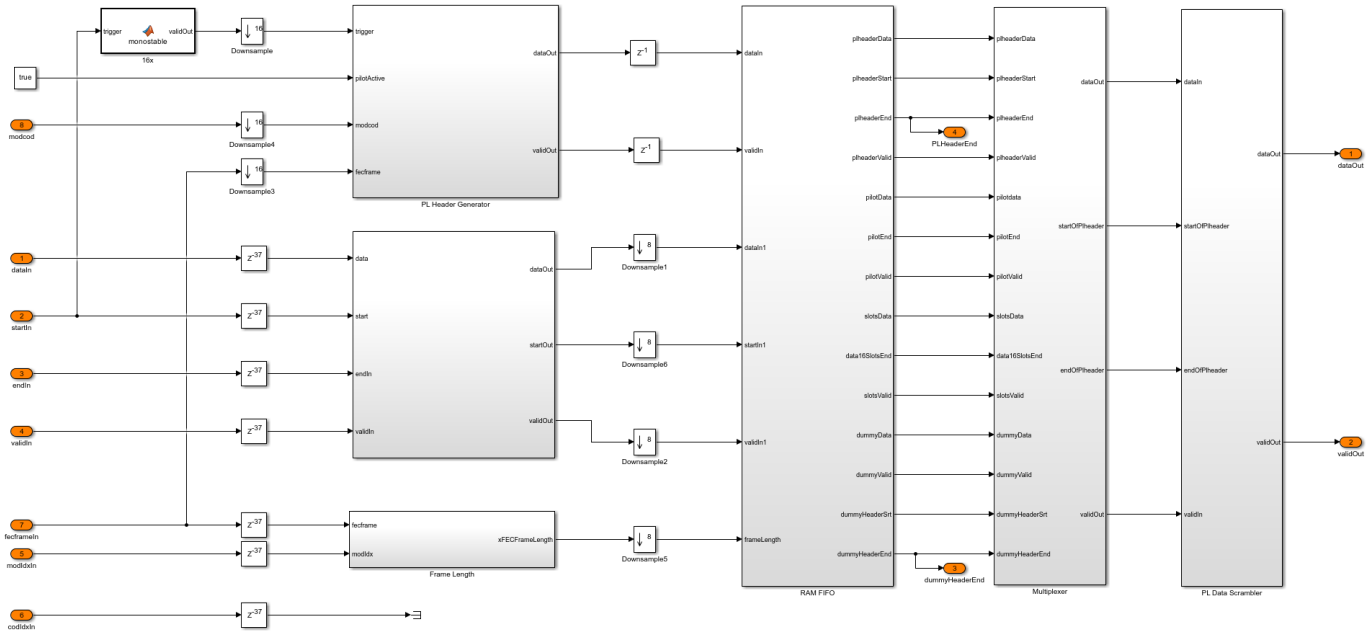
RRC Transmit Filter

The RRC Transmit Filter subsystem upsamples the input by a factor of four and uses the Discrete FIR Filter block with an RRC impulse response to pulse shape the PLFRAME symbols.
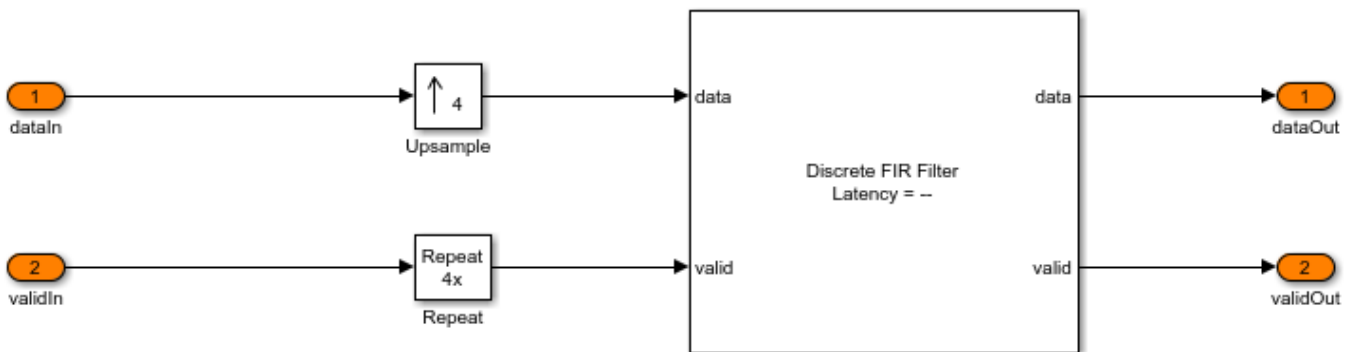
Run Model

Set the symbol rate, **MODCOD**, **FECFrame** type, input stream format, and user packet length on the mask of the Input Configuration subsystem and run the dvbs2hdlTransmitter model. Set the same symbol rate for the Input Configuration subsystem in the dvbs2hdlTransmitter model,

the DVB-S2 Tx subsystem of the `dvbs2hdlTransmitterCore` model reference. Alternatively, execute this command at the MATLAB Command Window to run the model.

`sim dvbs2hdlTransmitter`

The **MODCOD** and **FECFrame** must be row vectors. Each element of the row vectors corresponds to the values of **MODCOD** and **FECFrame** in a frame.

Verification and Results

Run the model to display the transmitter spectrum, error plot between the reference waveform and simulation output, and relative mean squared error of the simulation output.

```
### Starting serial model reference simulation build.
### Successfully updated the model reference simulation target for: dvbs2hdlTransmitterCore

Build Summary

Simulation targets built:

Model                    Action                          Rebuild Reason
=================================================================================================
dvbs2hdlTransmitterCore  Code generated and compiled.  dvbs2hdlTransmitterCore_msf.mexw64 does n

1 of 1 models built (0 models already up to date)
Build duration: 0h 2m 38.59s


Simulation Completed. Running verification script...
Relative mean squared error (dB) between the simulink output and reference = -79.6677
```
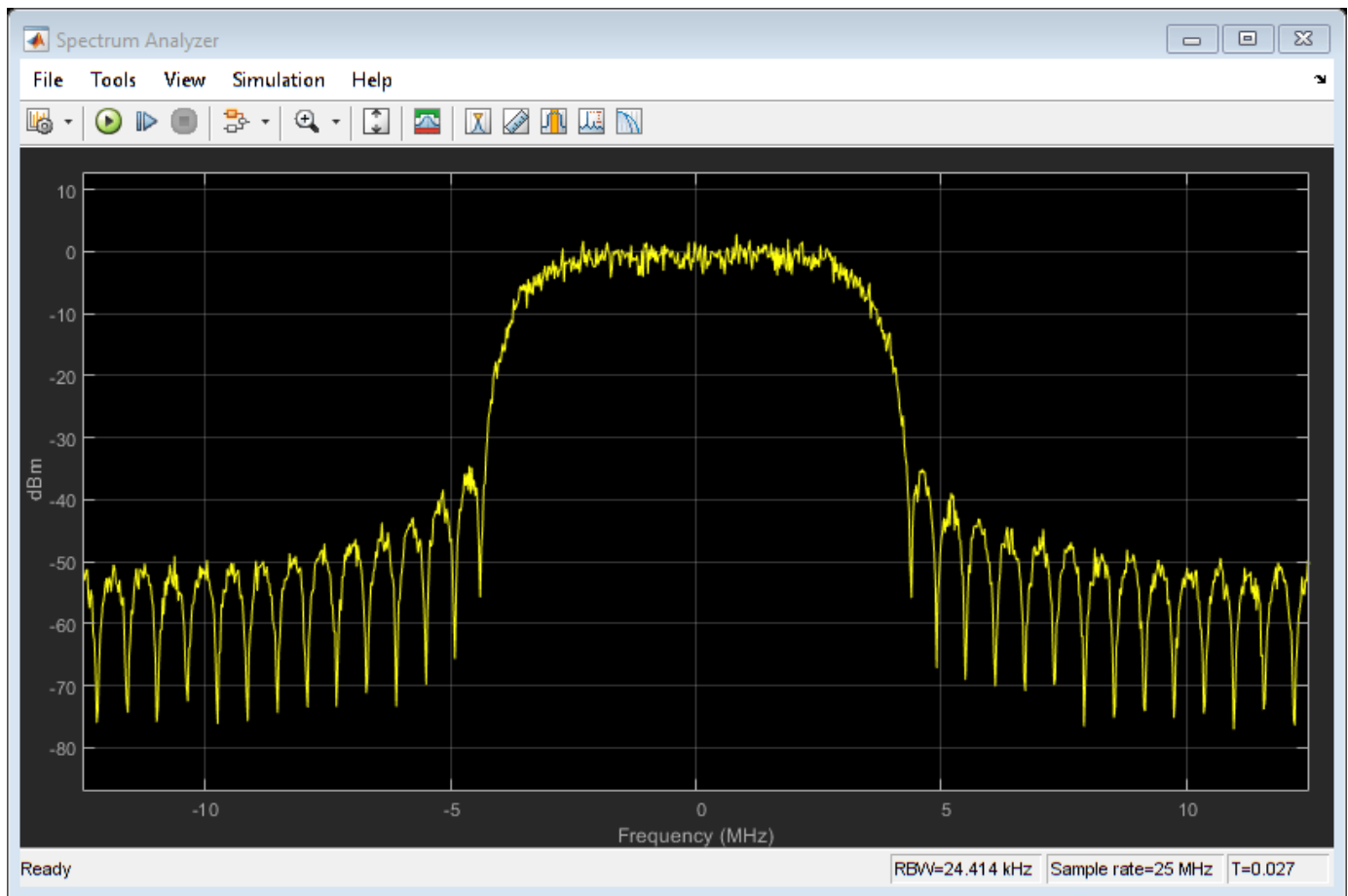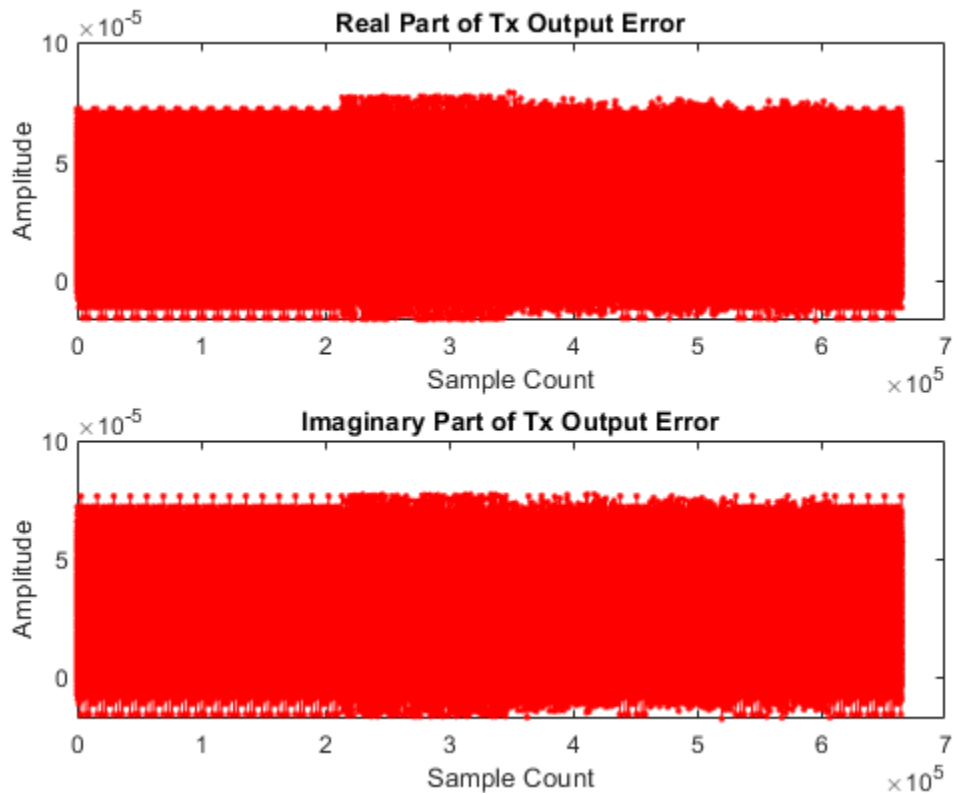
HDL Code Generation

To generate the HDL code for this example, you must have HDL Coder™. Use `makehdl` and `makehdltb` functions to generate HDL code and HDL test bench for the `DVB-S2 HDL transmitter` subsystem. The test bench generation time depends on the simulation time.

The resulting HDL code is synthesized for a Xilinx® Zynq® UltraScale+ RFSoC ZCU111 board. This table shows the post place and route resource utilization. The maximum frequency of operation is 202 MHz.

```
     Resources        Usage

    _____      _____

  CLB LUT            14717
  CLB Registers      8348
  RAMB36             151
  RAMB18             1
  DSP48              42
```

References

1 ETSI EN 302 307-1. *Digital Video Broadcasting (DVB); Second Generation Framing Structure, Channel Coding and Modulation Systems for Broadcasting, Interactive Services, News Gathering and other Broadband Satellite Applications (DVB-S2).*

2 ETSI TR 102 376-1. *Digital Video Broadcasting (DVB); Implementation Guidelines for the Second Generation System for Broadcasting, Interactive Services, News Gathering and other Broadband Satellite Applications (DVB-S2).*

See Also

Blocks
DVB-S2 Symbol Modulator | General CRC Generator HDL Optimized | Discrete FIR Filter

Functions
`dvbs2WaveformGenerator`

Related Examples

- "DVB-S2 HDL BCH Encoder" on page 4-83
- "DVB-S2 HDL LDPC Encoder" on page 4-64
- "DVB-S2 HDL Receiver" on page 5-280

GPS HDL Acquisition and Tracking Using C/A Code

This example shows how to acquire and track multiple Global Positioning System (GPS) satellite signals from a GPS baseband waveform using Simulink® blocks that are optimized for HDL code generation and hardware implementation. You use the L1 Coarse/Acquisition (C/A) code in the input waveform to perform signal acquisition and track the satellites. In acquisition, you detect the satellite signals in the waveform and estimate the coarse Doppler and C/A code phase offsets for the detected satellites. In tracking, you fine-tune the estimates and correct them to recover the legacy navigation (LNAV) symbols. The LNAV symbols can be used for GPS position estimation.

Model Overview

The model that you use in this example mainly consists of acquisition and tracking modules. The input to the model is a GPS baseband waveform with a sample rate of 32.768 Msps. This signal contains multiple satellite waveforms with Doppler offsets, code phase offsets, and additive white gaussian noise (AWGN) noise in it. The model upsamples the GPS waveform by a factor of six and increases the clock from 32.768 MHz to 196.6 MHz to achieve faster acquisition.

The Acquisition block decimates the GPS waveform from 32.768 Msps to 4.096 Msps and operates at this lower sampling rate. This block detects the satellites and generates coarse estimates for them.

The Tracking block uses the GPS waveform at 32.768 Msps, the pseudorandom noise identifier (PRNID) of the detected satellites, and the coarse estimates to track the satellites. Atleast four satellites are required for GPS position estimation, so tracking does not start until the acquisition process detects four satellites. The Tracking block contains eight instances of tracking logic to simultaneously track eight satellites. The Tracking block uses phase-locked loop, frequency-locked loop, and delay-locked loop to recover the pi/2-BPSK modulated LNAV symbols of each detected satellite. This figure shows an overview of the example model.
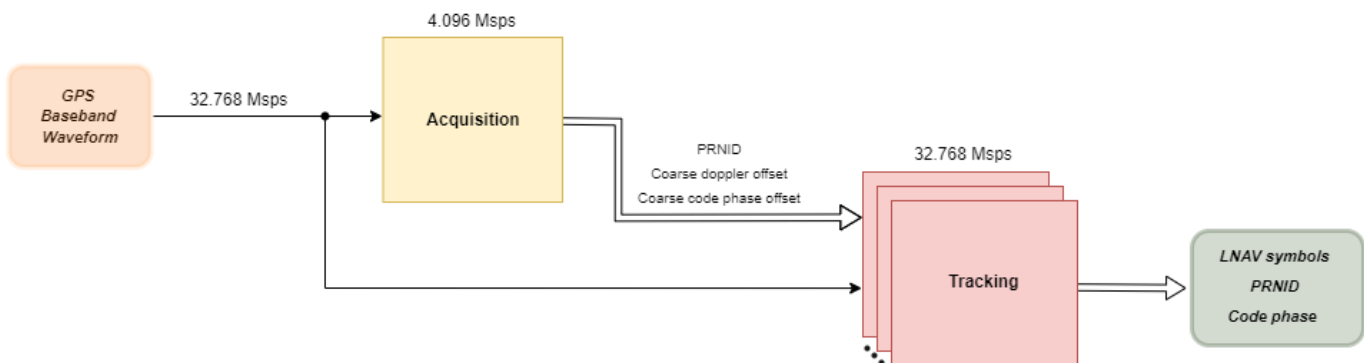
File Structure

This example uses one Simulink model, two functions and two scripts.

- `gpshdlAcquisitionTracking` — Model to acquire signals and track satellites.
- `gpshdlAcquisitionAndTrackingUsingCACodeInit` — Generates all the parameters and inputs required to run the `gpshdlAcquisitionTracking` model. The model calls this script in the `InitFcn` callback.

- `gpshdlAcquisitionAndTrackingUsingCACodeParameters` — Generates the parameters required to run the model. The model calls this function in the `gpshdlAcquisitionAndTrackingUsingCACodeInit` script.

- `gpshdlGenerateRxInput` — Generates the input GPS waveform for the receiver. The model calls this function in the `gpshdlAcquisitionAndTrackingUsingCACodeInit` script.

- `gpshdlAcquisitionTrackingValidate` — Validates the outputs of the model and plots them for visualization. The model calls this function in the `StopFcn` callback.

Model Interface

This figure shows the structure of the `gpshdlAcquisitionTracking` model.
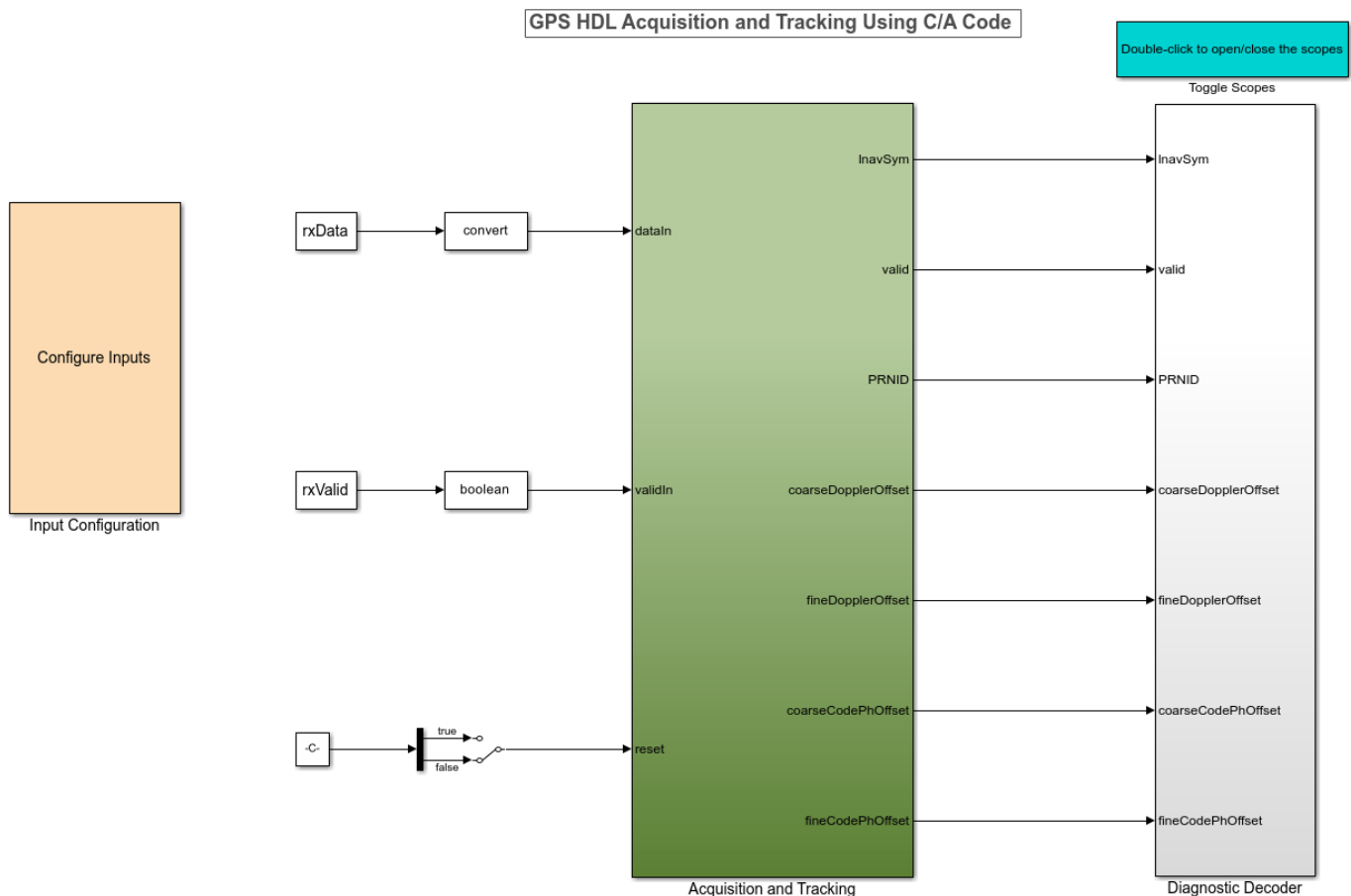
Copyright 2022 The MathWorks, Inc.

Model Inputs

- **dataIn** — Input data, specified as 18-bit complex data.
- **validIn** — Control signal to validate the **dataIn** signal, specified as a Boolean scalar.
- **reset** — Control signal to reset the receiver. The receiver restarts from acquisition.

Model Outputs:

The output ports are vectors of length 8 because the example tracks eight satellites simultaneously.

- **lnavSym** — Legacy navigation symbols in the input waveform, specified as 16-bit complex data.
- **validOut** — Control signal to validate all the output ports, specified as a Boolean signal.
- **PRNID** — PRNIDs of detected satellites, specified as a 6-bit unsigned integer.
- **coarseDopplerOffset** — Estimated coarse Doppler offset of the detected satellites, specified as a 16-bit integer.
- **fineDopplerOffset** — Estimated fine Doppler offset of the detected satellites, specified as 25-bit real data.
- **coarseCodePhOffset** — Estimated coarse C/A code phase offset of the detected satellites, specified as 20-bit real data.
- **fineCodePhOffset** — Estimated fine C/A code phase offset of the detected satellites, specified as 20-bit real data.

Input Configuration

Double-click the `Input Configuration` subsystem to configure the transmitter parameters and channel impairments.

- **Number of LNAV data bits** — Number of LNAV data bits to generate transmitter waveform. Use a minimum of eight bits to ensure successful tracking convergence.
- **Number of satellites** — Number of satellites to include in the transmitter waveform, specified as an integer in the range [1,8].
- **Satellite PRNIDs** — PRNIDs of satellites. This value must be a column vector of size equal to number of satellites. Each PRNID must be an integer in the range [1,32].
- **SNR (in dB)** — Signal-to-noise ratio (SNR) of individual satellites in the transmitter waveform, in dB, specified as a column vector of size equal to the number of satellites. The minimum SNR value must be -20 dB.
- **Peak Doppler offset** — Maximum Doppler offset to introduce in the satellite waveform, in Hz, specified as a column vector of size equal to the number of satellites. Each entry of this vector must be in the range [–10000, 10000].
- **Doppler rate** — Rate of change of the Doppler offset, specified in Hz/sec, specified as a column vector of size equal to the number of satellites. Each entry of this vector must be no greater than 1000.
- **C/A code phase offset** — C/A code delay to introduce in the waveform, specified as a column vector of size equal to the number of satellites. Each entry of this vector must be in the range (–1023, 1023).

Model Structure

This figure shows the `Acquisition and Tracking` subsystem. This subsystem consists of the `Acquisition`, `Time Synchronization`, and `Tracking` subsystems.
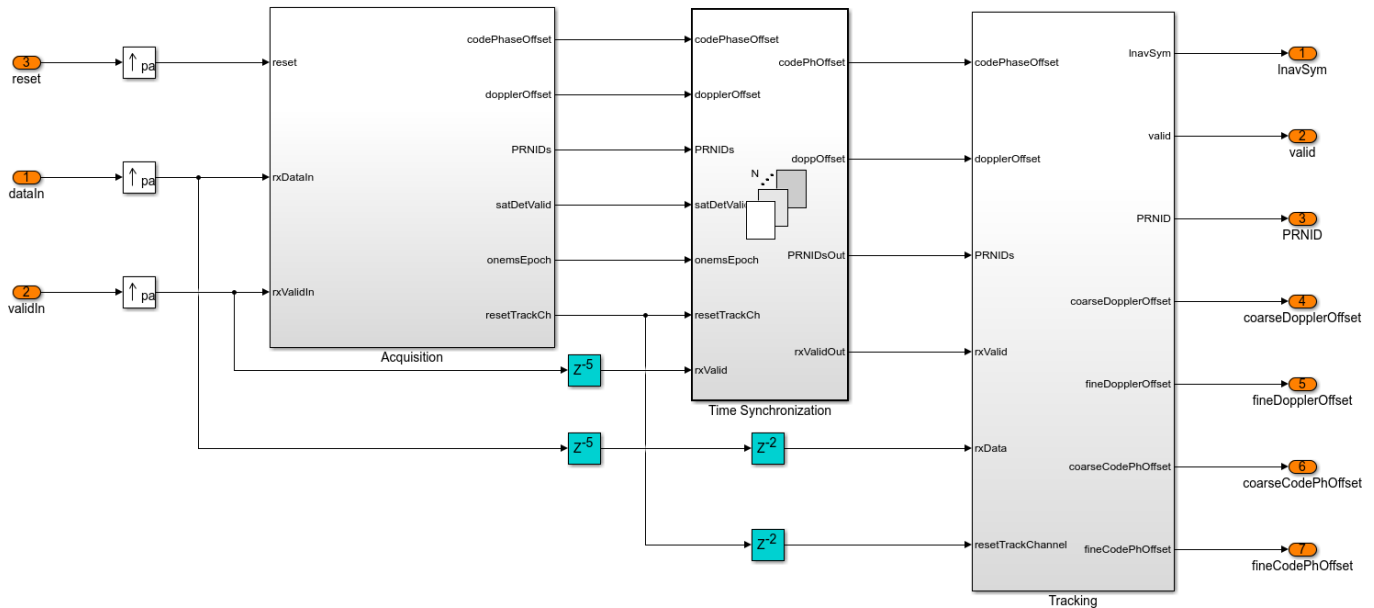
Acquisition Subsystem

The `Acquisition` subsystem accepts the GPS waveform at a sampling rate of 32.768 Msps, decimates it to 4.096 Msps, and stores one millisecond duration of the decimated waveform. The subsystem selects coarse Doppler frequencies sequentially from -10 kHz to 10 kHz in steps of 1 kHz, generates local carrier waveforms at these frequencies, compensates for these frequencies in the decimated waveform, and outputs carrier-wiped-off waveform. The subsystem converts the carrier-wiped-off waveform into the frequency domain using a 4096-point fast Fourier transform (FFT). The subsystem fetches the frequency-domain C/A code from a look-up table (LUT) and correlates it with the waveform to find the correlation peak. When the peak is greater than a dynamic threshold, the subsystem detects the satellite with the C/A code is detected and the corresponding Doppler frequency and C/A code phase are the coarse estimates of the satellite. The subsystem performs this frequency-domain correlation for four satellites in parallel and for eight sequential searches to finish searching all 32 GPS satellites. The subsystem then sorts and selects the eight detected satellites with the strongest correlation peaks. The subsystem also generates a 1 ms epoch signal, which asserts, for every 1 ms, to use that signal for time synchronization.

The `Acquisition` subsystem contains these main subsystems:

- `Control Acquisition` — Generates control signals to start acquisition and selects the satellite PRNIDs to search.

- `Decimation` Decimates the input waveform from 32.768 Msps to 4.096 Msps using Cascaded Integrator Comb (CIC) and Finite Impulse Response (FIR) decimation.

- `RAM Read and Write` — Writes one millisecond of decimated waveform to the RAM and reads the waveform multiple times from the RAM until acquisition finishes.

- `Carrier Wipeoff` — Generates the local waveform at coarse Doppler frequencies using a numerically controlled oscillator (NCO) and compensates for the Doppler frequencies in the decimated waveform to output carrier-wiped-off waveform.

- `Correlation` — Performs frequency-domain correlation of the carrier-wiped-off waveform with four satellite C/A codes simultenously. This subsystem also finds correlation peaks, generates a threshold, and compares the peaks with the threshold to detect satellites.

- `Sort and Prepare Outputs` — Sorts the PRNIDs, coarse Doppler frequencies, and coarse code phase offset values of the detected satellites in decreasing order of their correlation peaks and outputs the eight satellites with the strongest peaks .
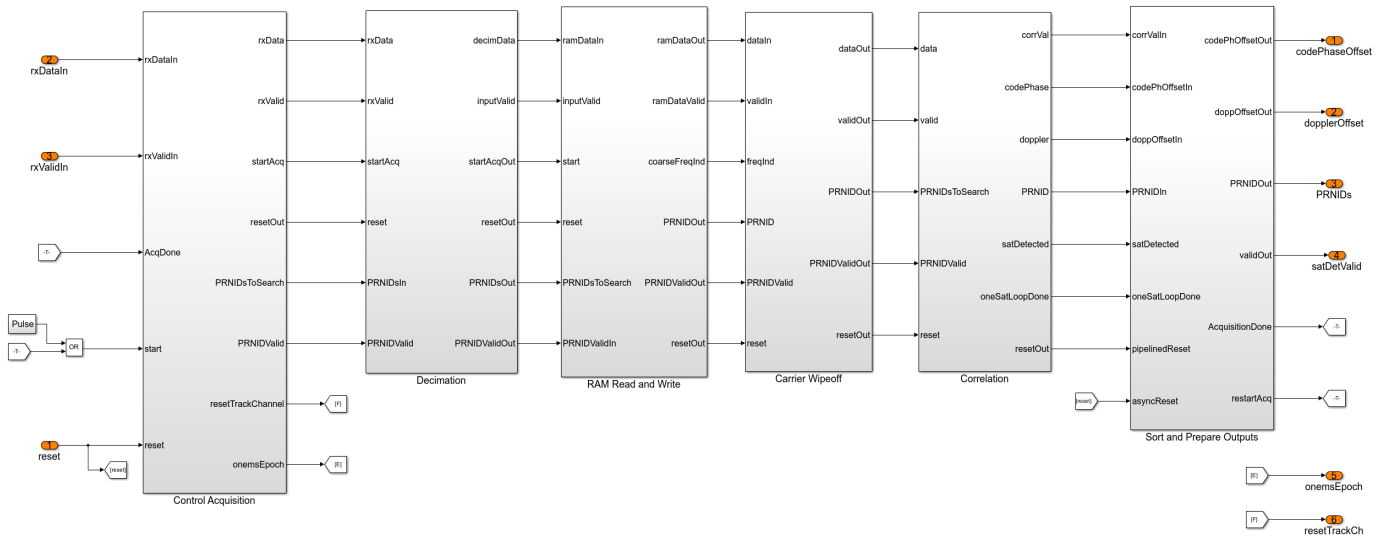
Time Synchronization Subsystem

The `Time Synchronization` subsystem accepts the GPS waveform, the detected satellite PRNIDs, coarse Doppler frequencies, coarse code phase offsets, and the 1 ms epoch signal from the `Acquisition` subsystem. The `Time Syncrhonization` subsystem synchronizes the input GPS waveform after the `Acquisition` subsystem finishes detecting the satellites and when the incoming 1 ms epoch signal asserts.

Tracking Subsystem

The `Tracking` subsystem tracks the satellites detected through acquisition. The subsystem accepts the time-synchronized waveform at 32.768 Msps, the detected satellite PRNIDs, coarse Doppler offsets, and code phase offsets. The subsystem uses the coarse Doppler estimate and phase estimate to generate a local carrier using an NCO, removes the Doppler offset from the waveform, and returns a carrier-wiped-off waveform. The subsystem uses the detected PRNID and the coarse code phase offset to fetch replica C/A code from an LUT. The subsystem generates early, prompt, and late versions of this C/A code to correlate with the carrier-wiped-off waveform. The subsystem integrates these correlated outputs for every 1 millisecond (predetection integration time) and returns integrated samples after every millisecond. The integrated prompt outputs are the LNAV symbols of the receiver. The subsystem uses the integrated early and late outputs to estimate delay error and the integrated prompt output to estimate frequency and phase errors. The subsystem filters these errors using loop filters and feeds the filtered values to the NCO and C/A code LUT to aid local carrier generation and replica C/A code generation. This tracking logic applies to a single satellite. Similarly, the `Tracking` subsystem generates eight instances of this tracking logic and tracks eight satellites simultenously.

The `Tracking` subsystem contains the `Tracking Core` subsystem that carries out the tracking logic. The `Tracking Core` subsystem contains these main subsystems:

- `NCO` — Accepts the coarse Doppler offset and filtered fine Doppler offset. The subsystem generates a local carrier to compensate for the Doppler offset in the input waveform.

- **CA Code Replica** — Accepts the detected satellite PRNID, coarse code phase offset, and filtered fine code phase offset to generate replica C/A code. This subsystem generates early, prompt, and late versions of the C/A code, each separated by a half C/A chip duration.
- **Code Wipeoff** — Multiplies the carrier-wiped-off waveform with the generated early, prompt, and late C/A codes to give out code-wiped-off waveforms.
- **Integrate and Dump** — Integrates the early, prompt, and late code-wiped-off waveforms every 1 millisecond and outputs the integrated values.
- **Discriminators and Loop Filters** — Uses the integrated prompt value to estimate the frequency and phase errors and uses the integrated early and late values to estimate the delay error. The first- and second-order loop filters filter the generated errors to provide fine estimates.

Run Model

Open the `gpshdlAcquisitionTracking` model and double-click the `Input Configuration` subsystem to the change transmitter configuration and channel impairments. Run the model.

Note: It may take around 25 minutes to complete the simulation.

```
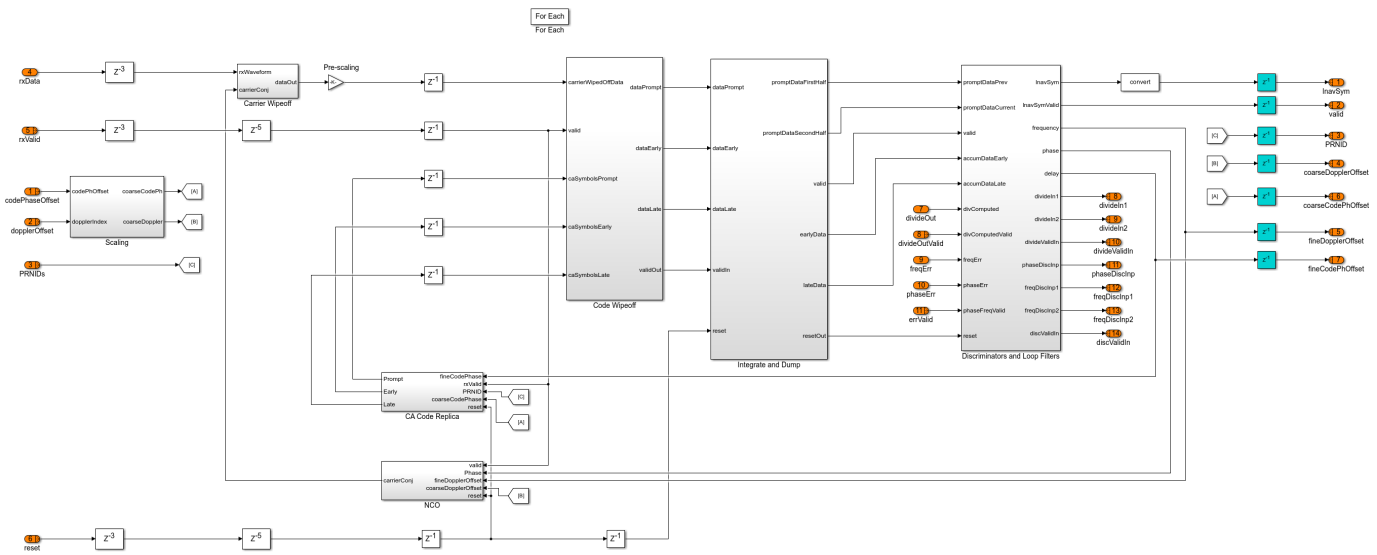### Building the rapid accelerator target for model: gpshdlAcquisitionTracking
### Successfully built the rapid accelerator target for model: gpshdlAcquisitionTracking

Build Summary

Top model rapid accelerator targets built:

Model                      Action                          Rebuild Reason
==================================================================================================
gpshdlAcquisitionTracking  Code generated and compiled.    Code generation information file does no

1 of 1 models built (0 models already up to date)
Build duration: 0h 4m 34.936s
Warning: In rapid accelerator mode, when the simulation is started from the
command line, visualization blocks are not updated. If the simulation is
started from the toolstrip, visualization blocks are updated.
    Input PRNID     Detected PRNID
```

```
  _____        _____

      11                 11
      18                 18
      22                 22
      23                 23
```

Input code phase offset   Estimated code phase offset
_____   _____

```
        300.34                     300.19
        312.88                     312.72
        587.21                     587.07
        425.89                     425.72
```

Input doppler offset   Estimated doppler offset
_____   _____

```
        3289                    3253.8
        1568                    1534.7
        5856                    5892.2
        7796                    7836.1
```

Estimated C/A Code Phase Offset
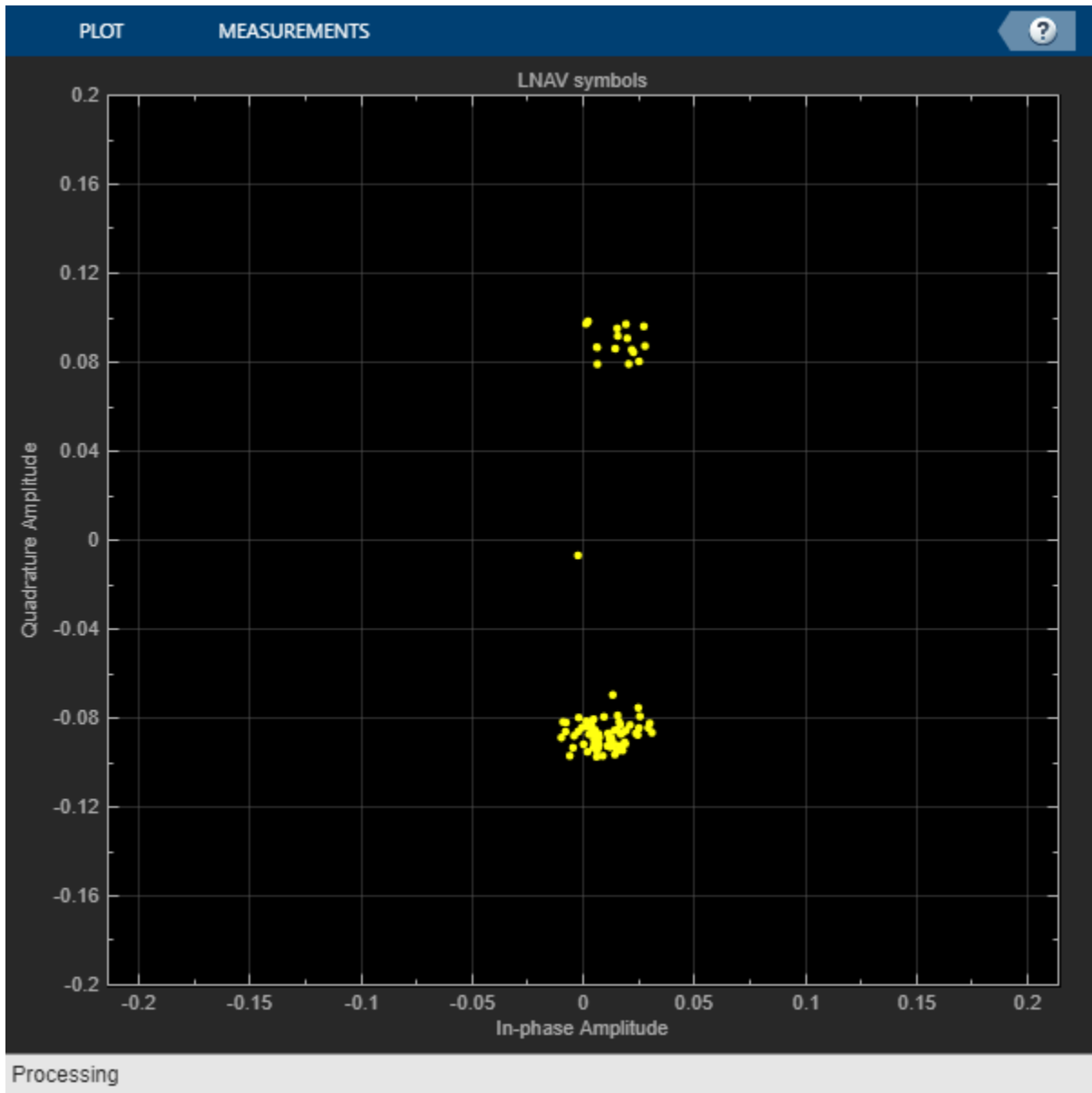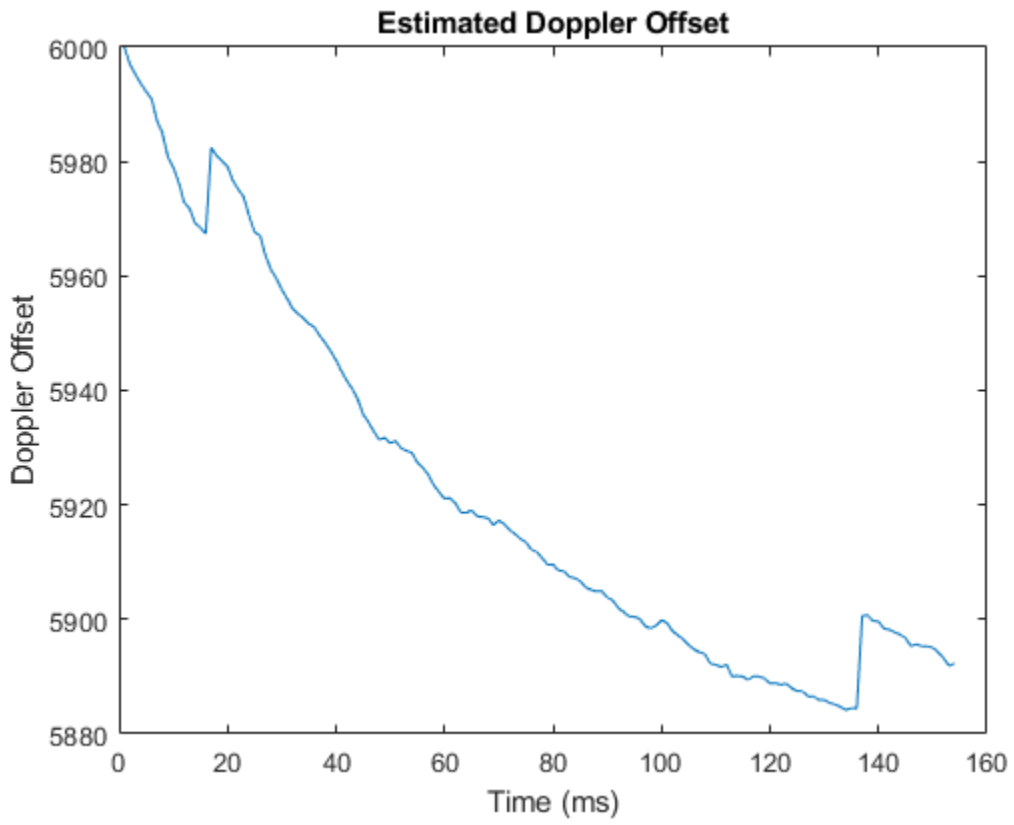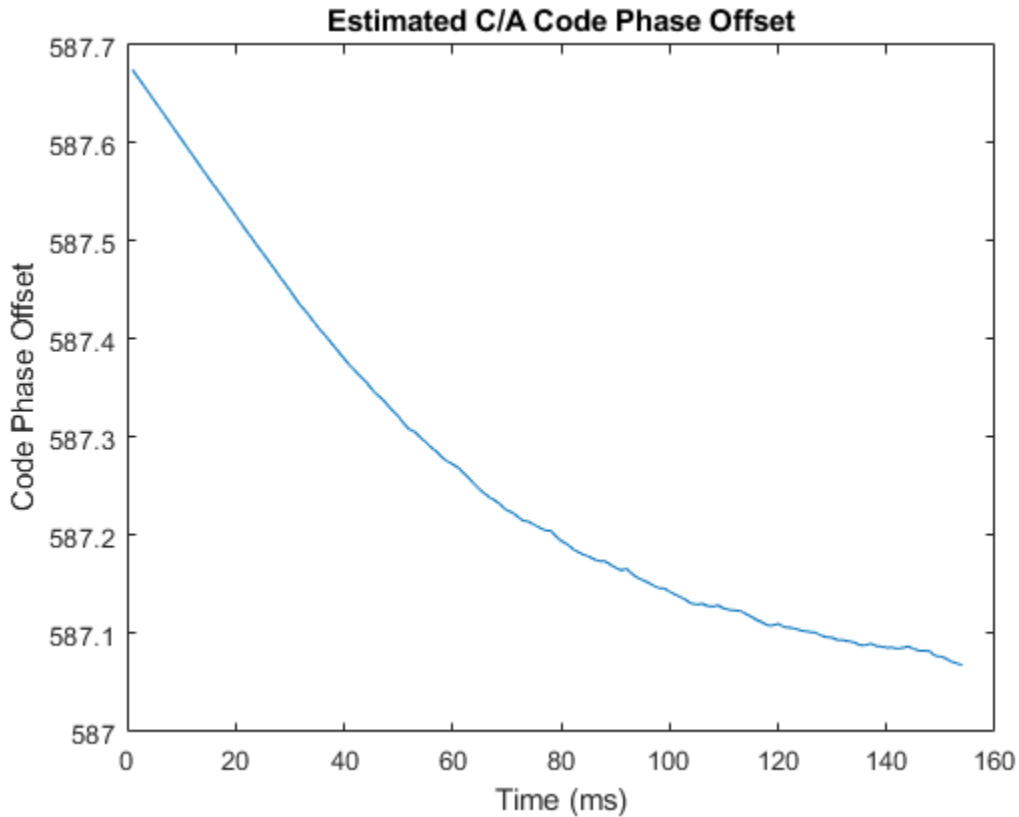
Generate HDL Code

To generate the HDL code, you must have an HDL Coder™ license. Use `makehdl` and `makehdltb` functions to generate HDL code and a HDL test bench for the `Acquisition and Tracking` subsystem.

The resulting HDL code is synthesized for a Xilinx® Zynq®-7000 ZC706 evaluation board. This table shows the post place and route resource utilization. The maximum frequency of operation is 200 MHz.

```
       Resources       Usage

    _____    _____

    Slice LUT           65695
    Slice Registers     71376
    RAMB36              176
    RAMB18              52
    DSP48               269
```

Appendix

This example uses these helper files:

- HelperGPSAlmanac2Struct.m — Convert text file of almanac to structure
- HelperGPSCEIConfig.m — Create configuration object for GPS navigation data

- HelperGPSNAVDataEncode.m — Encode navigation data from configuration object into bits
- HelperGPSNavigationConfig.m — Create configuration object for GPS navigation data
- HelperGPSNum2BitsVector.m — Scale and convert the value to bits using left MSB

References

[1]. Kaplan, Elliot D., and C. Hegarty, eds., _Understanding GPS/GNSS: Principles and Applications. Third edition. GNSS Technology and Applications Series. Boston; London; Artech House, 2017.

[2]. IS-GPS-200, Rev: L. "NAVSTAR GPS Space Segment/Navigation User Segment Interfaces." GPS Enterprise Space & Missile Systems Center (SMC) - LAAFB. https://www.gps.gov/technical/icwg/IS-GPS-200L.pdf

[3]. Ward, P.W. "GPS Receiver Search Techniques." In Proceedings of Position, Location and Navigation Symposium - PLANS '96, 604 - 11. Atlanta, GA, USA: IEEE, 1996. https://doi.org/10.1109/PLANS.1996.509134.

Deploy Satellite Communications HDL Reference Applications on FPGAs and SoCs

This section contains the list of examples that show how to deploy satellite communications Wireless HDL Toolbox™ reference applications on FPGAs and SoCs.

- "DVB-S2 HDL PL Header Recovery Using Analog Devices AD9361/AD9364" (Communications Toolbox Support Package for Xilinx Zynq-Based Radio): Deploy a DVB-S2 time, frequency, and phase synchronization and PL header recovery algorithm.

This example reuses the satellite communication Simulink® models to generate HDL for the FPGA logic. They use hardware-software co-design modeling techniques and hardware support packages to add all the software modeling and interfacing required to implement the algorithm in real-time on hardware.

See Also

Related Examples
- "DVB-S2 HDL Receiver" on page 5-280
- "DVB-S2 HDL PL Header Recovery" on page 5-260