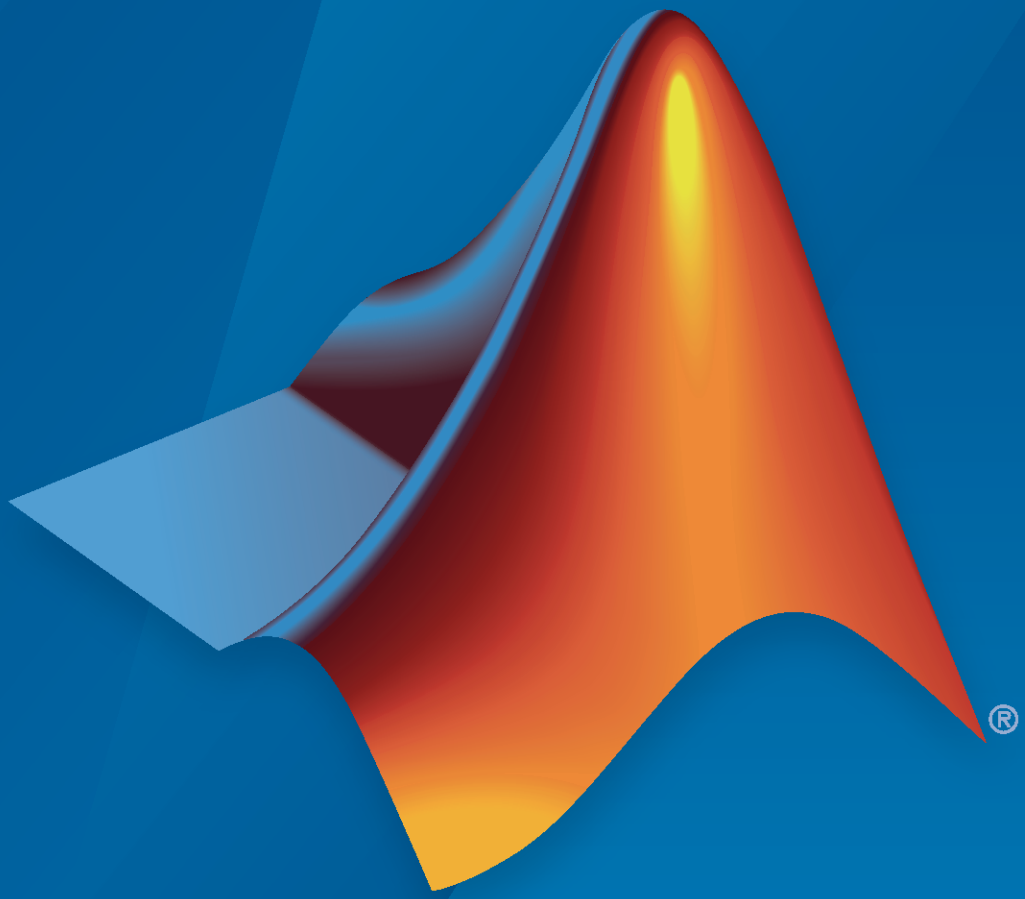
Wireless HDL Toolbox™

Reference



MATLAB® & SIMULINK®

R2021a

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

Wireless HDL Toolbox™ Reference

© COPYRIGHT 2017 - 2021 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

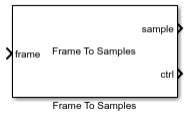
September 2017	Online only	New for Version 1.0 (Release 2017b)
March 2018	Online only	Revised for Version 1.1 (Release 2018a)
September 2018	Online only	Revised for Version 1.2 (Release 2018b)
March 2019	Online only	Revised for Version 1.3 (Release 2019a)
September 2019	Online only	Revised for Version 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)

Contents

Blocks

Frame To Samples

Convert frame-based data to sample stream
Library: Wireless HDL Toolbox / I/O Interfaces



Description

The Frame To Samples block flattens frame-based input into a stream of samples. The block also returns a stream of corresponding control signals that indicate sample validity and the boundaries of the frame. You can configure idle cycles inserted between samples or between frames, and how many values represent each sample. See “Streaming Sample Interface” for details of the streaming format.

Use this block to generate input for a subsystem targeted for HDL code generation. This block does not support HDL code generation.

Ports

Input

`frame` — Frame of input samples
column vector

Frame of input samples, specified as a column vector. All samples in the vector are considered valid. Each frame must be the same size.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

Output

`sample` — Output sample stream
scalar | vector

Output sample stream, returned **Output size** values at a time. The output stream includes idle samples as specified by **Idle cycles between samples** and **Idle cycles between frames**. Each output sample has a corresponding set of control signals on the **ctrl** port.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

`ctrl` — Control signals accompanying sample stream
`samplecontrol` bus

Control signals accompanying the sample stream, returned as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the output frame
- `end` — Indicates the end of the output frame
- `valid` — Indicates that the data on the output **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

Parameters

`Idle cycles between samples` — Number of idle cycles to insert after each sample

`0` (default) | integer

Number of idle cycles to insert after each sample, specified as a scalar integer. The block returns a vector of **Output size** zeros for each idle cycle and sets all control signals to 0 (`false`).

`Idle cycles between frames` — Number of idle cycles to insert at the end of each frame

`0` (default) | integer

Number of idle cycles to insert at the end of each frame, specified as a scalar integer. The block returns a vector of **Output size** zeros for each idle cycle and sets all control signals to 0 (`false`).

`Output size` — Number of values representing each sample

1 (default) | positive integer

Number of values representing each sample, specified as a positive integer scalar. The block outputs a vector of **Output size** values. Each vector has one corresponding set of control signals. For example, you can use this parameter to serialize turbo-encoded samples. In the LTE standard, the turbo code rate is 1/3, so each sample is represented by one systematic value and two parity values: *S_n*, *P1_n*, and *P2_n*. In this case, set **Output size** to 3.

`Compose output from interleaved input samples` — Order of output samples relative to input order

off (default) | on

Order of output samples relative to input order, when more than one value represents each sample. For example, for 1/3 turbo-encoded samples, the input frame can be ordered `[S_1 P1_1 P2_1 S_2 P1_2 P2_2]` or `[S_1 S_2 P1_1 P1_2 P2_1 P2_2]`. In the first case, the output is two vectors, `[S_1 P1_1 P2_1]` and `[S_2 P1_2 P2_2]`. To achieve the same output in the second case, select **Compose output from interleaved input samples**.

Dependencies

This parameter applies when **Output size** is greater than one.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink® accelerator and rapid accelerator modes and for DPI component generation.

See Also

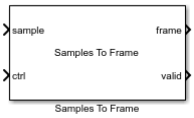
Blocks
Samples To Frame

Functions
`whdlFramesToSamples`

Introduced in R2017b

Samples To Frame

Convert sample stream to frame-based data
Library: Wireless HDL Toolbox / I/O Interfaces



Description

The Samples To Frame block reconstructs frame-based data from a stream of samples and its corresponding control signals. It removes any idle or nonvalid samples from the data. See “Streaming Sample Interface” for details of the streaming format.

Use this block to process output from a subsystem targeted for HDL code generation. This block does not support HDL code generation.

Ports

Input

`sample` — Stream of samples

scalar | vector

Stream of samples, specified as a scalar or vector. Vector input values represent a single sample, such as turbo-encoded samples represented by one systematic value and two parity values. The stream can include idle cycles between samples and between frames. Idle samples are discarded. `double` and `single` are supported for simulation but not for HDL code generation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, specified as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the input frame
- `end` — Indicates the end of the input frame
- `valid` — Indicates that the data on the input **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

Output

`frame` — Frame of output samples
column vector

Frame of output samples, returned as a column vector. Each frame is the same size. If the input frame is smaller than **Output size**, the block pads the frame with zeroes. If the output frame is larger than the **Output size**, the block forms the frame by using the first **Output size** samples. You can optionally output the number of valid samples in each frame on the **len** port.

`valid` — Validity of output frame
scalar

Validity of output frame, returned as a `Boolean` scalar. This port returns `1` (`true`) when the values on the **frame** and **len** (optional) ports, are valid.

Data Types: `Boolean`

`len` — Number of valid samples in output frame
integer

Number of valid samples in output frame, returned as an integer. The input sample stream can have frames of different sizes. The block returns a constant size vector on the **frame** port, padded with zeroes when the input frame is smaller than **Output size**. The **len** port indicates how many valid samples are in the output frame. If the output frame is larger than the **Output size**, the block forms the frame by using the first **Output size** samples.

Data Types: `double`

Parameters

`Input size` — Number of values representing each sample
1 (default) | positive integer

Number of values representing each sample, specified as a positive integer scalar. The block accepts a vector of **Input size** values. Each vector has one corresponding set of control signals. For example, you can use this parameter for turbo-encoded samples. In the LTE standard, the turbo code rate is 1/3, so each sample is represented by one systematic value and two parity values: *S_n*, *P1_n*, and *P2_n*. In this case, set **Input size** to `3`.

`Frame search window` — Number of input cycles to buffer
1 (default) | positive integer

Number of input cycles to buffer before attempting to form an output frame, specified as an integer. The block simulates faster when this parameter is larger. However, the block returns at most one frame from each search window. If more than one frame fits in this window, the block returns the first one it finds and drops the later frames. The default setting, 1 cycle, never drops frames, but results in slower simulation. Therefore, it is a best practice to set this parameter to the minimum number of cycles per frame, including idle cycles.

For example, calculate the valid cycles and idle cycles representing each frame. Each cycle may include more than one sample, depending on your **Input size** (*samplesize*) setting.

```
% Exact setting: includes idle cycles
totalframesize = ((framesamples/samplesize)*...
    (idlecyclesbetweensamples+1))+idlecyclesbetweenframes;
```

If the frame and sample spacing is variable or unknown, then a conservative compromise is to set the **Frame search window** to the minimum number of valid cycles per frame. For instance, for a turbo encoder block, the output frame size depends on the coding rate, 1/*R*, and tail bits specified by the LTE standard. The output data has *R* samples per cycle. This calculation does not include any idle cycles between samples or between frames.

```
% Conservative setting: number of valid output cycles, without idles
encoderrate = 3;
numtailbits = 12;
framesize = (framesamples+numtailbits)/encoderrate;
```

`Output size` — Maximum samples per frame
1024 (default) | positive integer

Maximum number of samples per frame, specified as an integer. The input sample stream can have frames of different sizes. The block returns a constant size vector, padded with zeroes if the frame is smaller than **Output size**. If the block receives a frame larger than **Output size**, it truncates the frame.

`Compose output from interleaved input samples` — Order of output samples relative to input order
off (default) | on

Order of output samples relative to input order, when more than one value represents each sample. For example, 1/3 turbo-encoded samples are represented by `[S_1 P1_1 P2_1]` and `[S_2 P1_2 P2_2]`. The default output order is `[S_1 P1_1 P2_1 S_2 P1_2 P2_2]`. To reorder the samples so that systematic and parity values are grouped together, select **Compose output from interleaved input samples**. The output order is then `[S_1 S_2 P1_1 P1_2 P2_1 P2_2]`.

`Enable frame length output port` — Output number of valid samples
off (default) | on

Enable frame length output port. Select this option to return the number of valid samples in each output frame. The length is returned on the **len** port and is qualified by the **valid** signal. Use this option when the sample stream has variable size frames or when a downstream block requires the frame size as input, such as LTE Turbo Decoder.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

See Also

Blocks

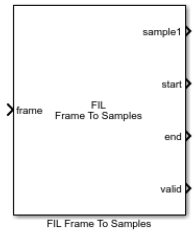
Frame To Samples

Functions
`whdlSamplesToFrames`

Introduced in R2017b

FIL Frame To Samples

Convert frame-based data to sample stream for FPGA-in-the-loop
Library: Wireless HDL Toolbox / I/O Interfaces



Description

The FIL Frame To Samples block performs the same frame-to-sample conversion as the Frame To Samples block. It returns output data as vectors of the entire frame of samples. The block returns control signal vectors of the same width as the sample data. This optimization makes more efficient use of the communication link between the FPGA board and your Simulink simulation when using FPGA-in-the-loop (FIL). To run FPGA-in-the-loop, you must have an HDL Verifier™ license.

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 to send one sample at a time to the FPGA. You can modify the autogenerated model to use the FIL Frame To Samples and FIL Samples To Frame blocks to improve communication bandwidth with the FPGA board by sending one frame at a time. For how to modify the autogenerated model, see “FPGA-in-the-Loop”.

Ports

Input

`frame` — Frame of input samples

column vector

Frame of input samples, specified as a column vector. All samples in the vector are considered valid. Each frame must be the same size.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

Output

`sampleN` — Sample stream

vector

Stream of samples, returned as a vector representing an entire frame. The output stream includes idle cycles between samples and between frames as specified in the block parameters.

If you set **Output size** greater than one, the block shows one port for each output value. In this case, a single sample is represented by N values, such as turbo-encoded samples represented by one systematic value and two parity values. The output data is one vector for each port.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

`start` — Start of frame
vector

Start of frame, returned as a `Boolean` vector containing one value for each sample in the frame. This signal is 1 (true) for one timestep corresponding to the first valid sample of the frame.

Data Types: `Boolean`

`end` — End of frame
vector

End of frame, returned as a `Boolean` vector containing one value for each sample in the frame. This signal is 1 (true) for one timestep corresponding to the last valid sample of the frame.

Data Types: `Boolean`

`valid` — Validity of samples
vector

Validity of samples, returned as a `Boolean` vector containing one value for each sample in the frame. This signal is 1 (true) on timesteps that correspond to valid samples.

Data Types: `Boolean`

Parameters

`Idle cycles between samples` — Number of idle cycles to insert after each sample
`0` (default) | integer

Number of idle cycles to insert after each sample, specified as a scalar integer. The block returns a zero on each **sampleN** port for each idle cycle and sets all control signals to 0 (`false`).

`Idle cycles between frames` — Number of idle cycles to insert at the end of each frame
`0` (default) | integer

Number of idle cycles to insert at the end of each frame, specified as a scalar integer. The block returns a zero on each **sampleN** port for each idle cycle and sets all control signals to 0 (`false`).

`Output size` — Number of values representing each sample
`1` (default) | positive integer

Number of values representing each sample, specified as a positive integer scalar. The block has **Output size** output sample ports. The control signals apply to all **sampleN** ports.

For example, you can use this parameter to serialize turbo-encoded samples. In the LTE standard, the turbo code rate is 1/3, so each sample is represented by one systematic value and two parity values: *S_n*, *P1_n*, and *P2_n*. In this case, set **Output size** to 3.

`Compose output from interleaved input samples` — Order of output samples relative to input order
off (default) | on

Order of output samples relative to input order, when more than one value represents each sample.

For example, for 1/3 turbo-encoded samples, the input frame can be ordered `[S_1 P1_1 P2_1 S_2 P1_2 P2_2]` or `[S_1 S_2 P1_1 P1_2 P2_1 P2_2]`. In the first case, the output is two vectors, `[S_1 P1_1 P2_1]` and `[S_2 P1_2 P2_2]`. To achieve the same output in the second case, select **Compose output from interleaved input samples**.

Dependencies

This parameter applies when **Output size** is greater than one.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

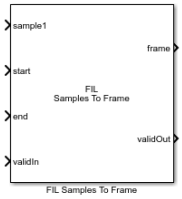
See Also
FIL Samples To Frame | Frame To Samples

Topics
“Streaming Sample Interface”
“FPGA-in-the-Loop”

Introduced in R2017b

FIL Samples To Frame

Convert sample stream from FPGA-in-the-loop to frame-based data
Library: Wireless HDL Toolbox / I/O Interfaces



Description

The FIL Samples To Frame block performs the same sample-to-frame conversion as the Samples To Frame block. It accepts input data as vectors of the entire frame of samples. The block expects control signal input vectors of the same width as the sample data. This optimization speeds up the communication link between the FPGA board and your Simulink simulation when using FPGA-in-the-loop. To run FPGA-in-the-loop, you must have an HDL Verifier license.

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 to send one sample at a time to the FPGA. You can modify the autogenerated model to use the FIL Frame To Samples and FIL Samples To Frame blocks to improve communication bandwidth with the FPGA board by sending one frame at a time. For how to modify the autogenerated model, see “FPGA-in-the-Loop”.

Ports

Input

`sampleN` — Stream of samples
vector

Stream of samples, specified as a vector representing an entire frame. The stream can include idle cycles between samples and between frames. Idle samples are discarded.

If you set **Number of input samples** greater than one, the block shows one port for each input value. In this case, a single sample is represented by N values, such as turbo-encoded samples represented by one systematic value and two parity values. The input data is one vector for each port. The control signals apply to all **sampleN** ports.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

`start` — Start of frame
vector

Start of frame, specified as a `Boolean` vector containing one value for each sample in the frame. This signal is 1 (true) for one timestep corresponding to the first valid sample of the frame.

Data Types: `Boolean`

`end` — End of frame
vector

End of frame, specified as a `Boolean` vector containing one value for each sample in the frame. This signal is 1 (true) for one timestep corresponding to the last valid sample of the frame.

Data Types: `Boolean`

`validIn` — Validity of samples
vector

Validity of samples, specified as a `Boolean` vector containing one value for each sample in the frame. This signal is 1 (true) on timesteps that correspond to valid samples.

Data Types: `Boolean`

Output

`frame` — Frame of output samples
column vector

Frame of output samples, returned as a column vector. Each frame is the same size. If the input frame is smaller than **Output size**, the block pads the frame with zeroes. If the output frame is larger than the **Output size**, the block forms the frame by using the first **Output size** samples. You can optionally output the number of valid samples in each frame on the **len** port.

`validOut` — Validity of output frame
scalar

Validity of output frame, returned as a `Boolean` scalar. This port returns `1` (`true`) when the values on the **frame** and **len** (optional) ports, are valid.

Data Types: `Boolean`

`len` — Number of valid samples in output frame
integer

Number of valid samples in output frame, returned as an integer. The input sample stream can have frames of different sizes. The block returns a constant size vector on the **frame** port, padded with zeroes when the input frame is smaller than **Output size**. The **len** port indicates how many valid samples are in the output frame. If the output frame is larger than the **Output size**, the block forms the frame by using the first **Output size** samples.

Data Types: `double`

Parameters

`Number of input samples` — Number of values representing each sample
1 (default) | positive integer

Number of values representing each sample, specified as a positive integer scalar. The block has one **sampleN** port for each value. The control signals apply to all **sampleN** ports. For example, you can use this parameter for turbo-encoded samples. In the LTE standard, the turbo code rate is 1/3, so each sample is represented by one systematic value and two parity values: *S_n*, *P1_n*, and *P2_n*. In this case, set **Number of input samples** to 3.

`Output size` — Maximum samples per frame
1024 (default) | positive integer

Maximum number of samples per frame, specified as an integer. The input sample stream can have frames of different sizes. The block returns a constant size vector, padded with zeroes if the frame is smaller than **Output size**. If the block receives a frame larger than **Output size**, it truncates the frame.

`Compose output from interleaved input samples` — Order of output samples relative to input order
off (default) | on

Order of output samples relative to input order, when more than one value represents each sample. For example, 1/3 turbo-encoded samples are represented by `[S_1 P1_1 P2_1]` and `[S_2 P1_2 P2_2]`. The default output order is `[S_1 P1_1 P2_1 S_2 P1_2 P2_2]`. To reorder the samples so that systematic and parity values are grouped together, select **Compose output from interleaved input samples**. The output order is then `[S_1 S_2 P1_1 P1_2 P2_1 P2_2]`.

`Enable frame length output port` — Output number of valid samples
off (default) | on

Enable frame length output port. Select this option to return the number of valid samples in each output frame. The length is returned on the **len** port and is qualified by the **valid** signal. Use this option when the sample stream has variable size frames or when a downstream block requires the frame size as input, such as LTE Turbo Decoder.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

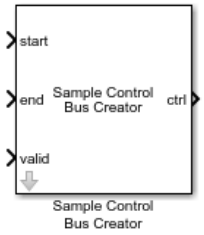
See Also
FIL Frame To Samples | Samples To Frame

Topics
“Streaming Sample Interface”
“FPGA-in-the-Loop”

Introduced in R2017b

Sample Control Bus Creator

Create control signal bus for use with Wireless HDL Toolbox blocks
Library: Wireless HDL Toolbox / Utilities



Description

The Sample Control Bus Creator block creates a `samplecontrol` bus for modeling streaming control signals in communication systems for hardware. See “Sample Control Bus”.

The block is an implementation of the Simulink Bus Creator block. See Bus Creator for more information.

Ports

Input

`start` — Start of frame
scalar

Start of frame, specified as a `Boolean` scalar. This signal is 1 (true) for one time step, corresponding to the first valid sample of the frame.

Data Types: `Boolean`

`end` — End of frame
scalar

End of frame, specified as a `Boolean` scalar. This signal is 1 (true) for one time step, corresponding to the last valid sample of the frame.

Data Types: `Boolean`

`valid` — Validity of samples
scalar

Validity of samples, specified as a `Boolean` scalar. This signal is 1 (true) on time steps that correspond to valid samples.

Data Types: `Boolean`

Output

`ctrl` — Control signals accompanying sample stream
`samplecontrol` bus

Control signals accompanying the sample stream, returned as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the output frame
- `end` — Indicates the end of the output frame
- `valid` — Indicates that the data on the output **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder™ provides additional configuration options that affect HDL implementation and synthesized logic.

To learn more about using buses for HDL code generation, see “Buses” (HDL Coder) and “Use Bus Signals to Improve Readability of Model and Generate HDL Code” (HDL Coder).

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

See Also

Blocks

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

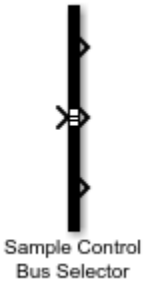
Functions
`whdlFramesToSamples` | `whdlSamplesToFrames`

Topics
“Streaming Sample Interface”
“Sample Control Bus”

Introduced in R2017b

Sample Control Bus Selector

Select signals from the control signal bus used with Wireless HDL Toolbox blocks
Library: Wireless HDL Toolbox / Utilities



Description

The Sample Control Bus Selector block selects signals from the `samplecontrol` bus. This bus is used for modeling streaming control signals in communication systems for hardware. See “Sample Control Bus”.

The block is an implementation of the Simulink Bus Selector block. See Bus Selector for more information.

Ports

Input

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, specified as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the input frame
- `end` — Indicates the end of the input frame
- `valid` — Indicates that the data on the input **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

Output

`start` — Start of frame

scalar

Start of frame, returned as a `Boolean` scalar. This signal is 1 (true) for one time step, corresponding to the first valid sample of the frame.

Data Types: `Boolean`

`end` — End of frame
scalar

End of frame, returned as a `Boolean` scalar. This signal is 1 (true) for one time step, corresponding to the last valid sample of the frame.

Data Types: `Boolean`

`valid` — Validity of samples
scalar

Validity of samples, returned as a `Boolean` scalar. This signal is 1 (true) on time steps that correspond to valid samples.

Data Types: `Boolean`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

To learn more about using buses for HDL code generation, see “Buses” (HDL Coder) and “Use Bus Signals to Improve Readability of Model and Generate HDL Code” (HDL Coder).

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

See Also

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

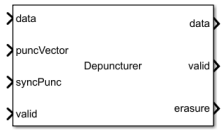
Functions
`whdlFramesToSamples` | `whdlSamplesToFrames`

Topics
“Streaming Sample Interface”
“Sample Control Bus”

Introduced in R2017b

Depuncturer

Reverse puncturing scheme to prepare for decoding
Library: Wireless HDL Toolbox / Error Detection and Correction



Description

The Depuncturer block replaces punctured symbols with neutral values as directed by an input puncture vector. The block returns erasure bits, which indicate the presence of neutral symbols in the output data stream. The block supports continuous and frame modes of operation. It provides an interface and architecture suitable for HDL code generation and hardware deployment.

Many wireless communication standards implement different code rates by puncturing patterns with a base code rate 1/2. The input to the block is a stream of one sample at a time. You can provide samples represented by hard-decision binary values or soft-decision log-likelihood ratios (LLR). The block returns output samples as 2-by-1 vectors.

The inserted neutral value depends on the data type of the input sample. For details, see the input **data** port.

Ports

Input

`data` — Input sample
scalar

Input sample, specified as a scalar. The block inserts a neutral value at punctured locations based on the data type of the input samples.

Input Data Type	Inserted Neutral Value
• `boolean` • `fixdt(0,1,0)`	0
`fixdt(0,WL,0)`	$2^{(WL-1)}$
`uint8`	128
`uint16`	32768
• `fixdt(1,WL,0)` • `int8` • `int16` • `single` • `double`	0

The block treats the input as hard-decision samples when the input type is `Boolean` or `fixdt(0,1,0)`. For signed and unsigned numeric types, the block assumes soft-decision samples. The block treats samples as signed integers for `single` and `double` data types, but these data types are not supported for HDL code generation.

The input sample must have a word length less than or equal to 16 bits, and a fraction length of 0 bits.

Data Types: `int8` | `int16` | `uint8` | `uint16` | `Boolean` | `fixdt(0,1,0)` | `fixdt(S,WL,0)` | `single` | `double`

`puncVector` — Puncture vector

column vector of binary values

Puncture vector, specified as a column vector of binary values. The length of the puncture vector must be an even number in the range [4, 28]. The length must remain constant. The block removes initial zeros from the provided vector, up to the first `1` (`true`). After the first `1` (`true`), the puncture vector cannot contain any [1:0] subvector matching [0 0].

For example, IEEE 802.11 WLAN standard [1] supports puncture rates 2/3, 3/4, and 5/6, with respective vector lengths of 4, 6, and 10. To support these multiple rates, set **Puncture vector source** to `Input port`. To support the largest vector size, the vector length must be 10 for all rates. For 2/3 and 3/4 rates, pad the **puncVector** input with zeros to create a 10-element vector. The puncture vector for rate 3/4 is `[1 1 0 1 1 0]'`. For a vector length of 10, use `[0 0 0 0 1 1 0 1 1 0]'` as the input **puncVector**.

When **Operation mode** is set to `Continuous`, the block captures the value of **puncVector** when both the **syncPunc** and input **valid** ports are `1` (`true`).

When **Operation mode** is set to `Frame`, the block captures the value of **puncVector** when both **ctrl**`.start` and **ctrl**`.valid` are `1` (`true`).

Dependencies

To enable this port, set **Puncture vector source** to `Input port`.

Data Types: `Boolean`

`syncPunc` — Puncture synchronization signal

scalar

Puncture synchronization signal, specified as a `Boolean` scalar value. This input is a control signal that synchronizes the puncture vector input with the input sample. When both **syncPunc** and **valid** are `1` (`true`), the block aligns the puncture vector to begin puncturing. The block captures the vector from either the **puncVector** input port or the **Puncture vector** parameter. The block ignores the **puncVector** port when **syncPunc** is `0` (`false`).

Dependencies

To enable this port, set **Operation mode** to `Continuous`. When **Operation mode** is `Frame`, the block synchronizes the puncture vector using control signals in the input **ctrl** bus.

Data Types: `Boolean`

`valid` — Validity of input samples

scalar

Control signal that indicates when the sample from **data** input port is valid. When **valid** is 1 (`true`), the block captures the values of the **data** input port. When **valid** is 0 (`false`), the block ignores the input samples.

Dependencies

To enable this port, set **Operation mode** to `Continuous`.

Data Types: `Boolean`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, specified as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

Dependencies

To enable this port, set **Operation mode** to `Frame`.

Data Types: `bus`

Output

`data` — Output sample

2-by-1 column vector

Output sample, returned as a 2-by-1 column vector. The data type is same as the data type of the input samples.

Data Types: `int8` | `int16` | `uint8` | `uint16` | `Boolean` | `fixdt(0,1,0)` | `fixdt(S,WL,0)` | `single` | `double`

`valid` — Validity of output data samples

scalar

Control signal that indicates when the sample from the **data** output port is valid. The block sets the **valid** port to `1` (`true`) when there is a valid sample on the output **data** port.

Dependencies

Tho enable this port, set **Operation mode** to `Continuous`.

Data Types: `Boolean`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, returned as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

Dependencies

To enable this port, set **Operation mode** to `Frame`.

Data Types: `bus`

`erasure` — Neutral symbol locations

2-by-1 column vector

Neutral symbol locations, returned as a 2-by-1 column vector corresponding to the output samples. When **erasure** is 1 (`true`), the corresponding output **data** element is a depunctured neutral value.

Data Types: `Boolean`

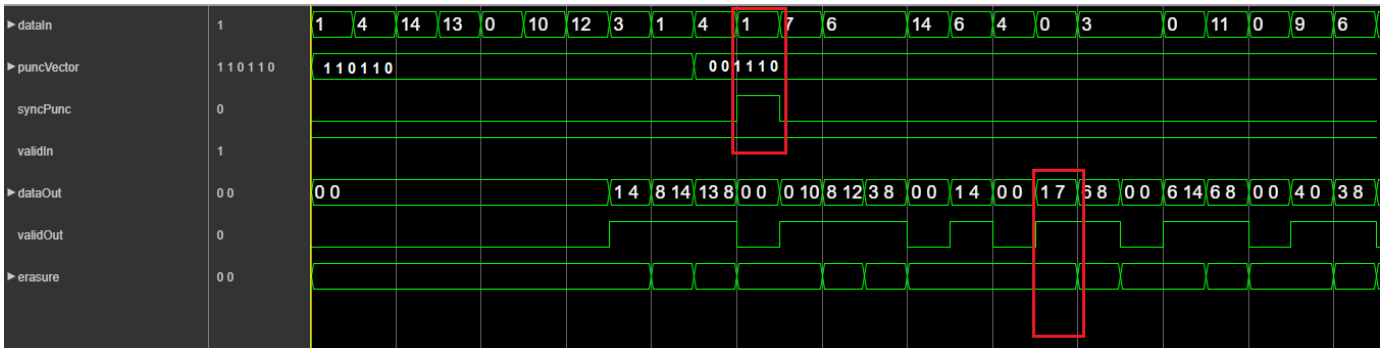
Parameters

`Operation mode` — End of frame behavior

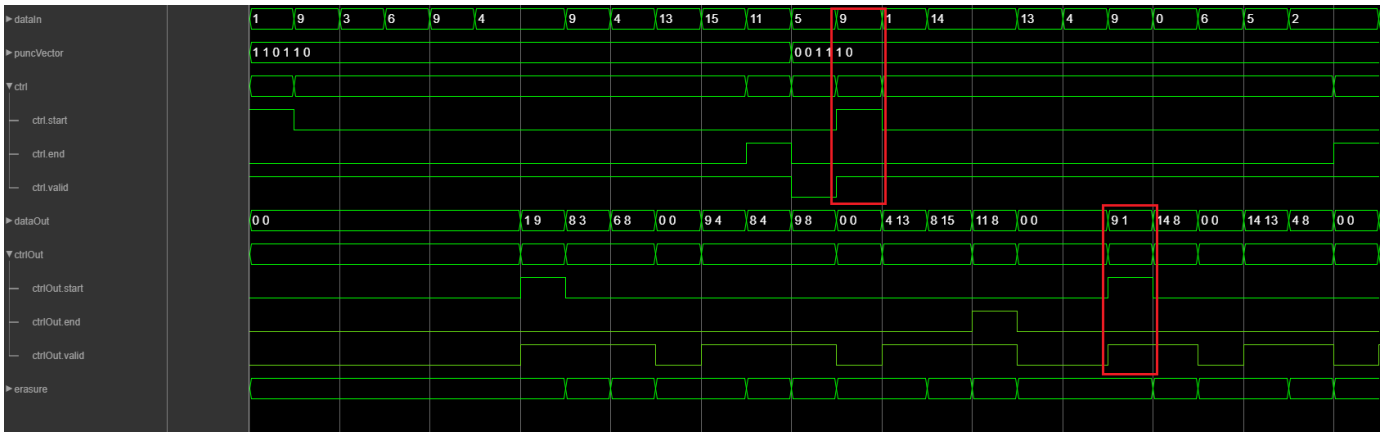
`Continuous` (default) | `Frame`

End of frame behavior, specified as one of these modes:

- `Continuous` – Allow changes to **puncVector** at any time. To force the block to capture the new puncture vector, set **syncPunc** to 1(`true`). This waveform shows `ufix4` input samples depunctured in `Continuous` mode.



- `Frame` – You can only change **puncVector** at the start of a frame, indicated by **ctrl**`.start`. This waveform shows `ufix4` input samples depunctured in `Frame` mode.



`Puncture vector source` — Source of puncture vector

`Input port` (default) | `Property`

Source of puncture vector, specified as either:

- `Input port` – Specify the puncture vector using the **puncVector** port.
- `Property` – Specify the puncture vector using the **Puncture vector** parameter.

`Puncture vector` — Locations to insert neutral values
`[1;1;0;1;1;0]` (default) | column vector of binary values

Puncture vector, specified as a column vector of binary values. The length of the puncture vector must be an even number in the range [4, 28]. The default value is the puncture vector for 3/4 code rate of IEEE 802.11 WLAN [1].

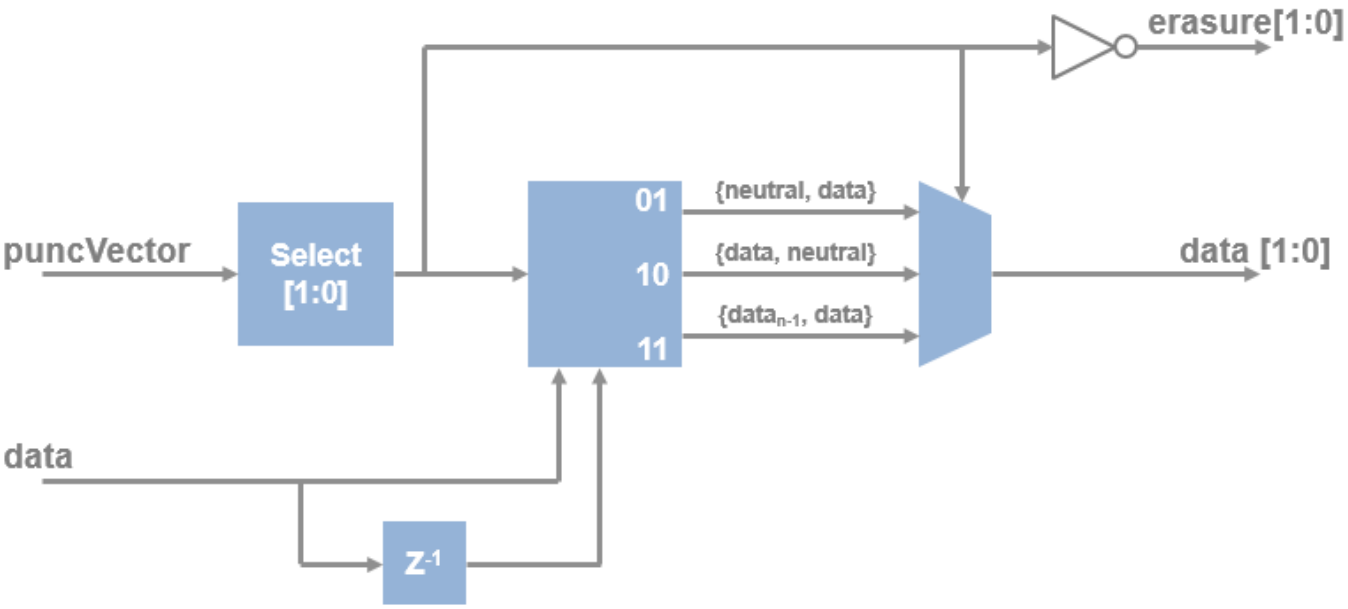
The puncture vector cannot contain any [1:0] subvector matching `[0 0]`.

Dependencies

To enable this port, set **Puncture vector source** to `Property`.

Algorithms

The depuncturing algorithm shifts through each [1:0] subvector of the puncture vector. The subvector has three valid patterns: [0 1], [1 0], or [1 1]. Based on the subvector, neutral samples are inserted in place of punctured samples. The erasure output is the inverse of the puncture subvector. The block returns an error when it encounters the invalid subvector [0 0].



Latency

When you set **Operation mode** to `Continuous`, the latency from valid input to valid output in is seven cycles. When you set **Operation mode** to `Frame`, the latency is six cycles.

Performance

These resource and performance data are the synthesis results from the generated HDL targeted to a Xilinx®Zynq®-7000 ZC706 board. The block is using `ufix4` input samples, in continuous mode with default settings. The design achieves a clock frequency of 590 MHz.

Resource	Number Used
LUT	54
FFS	67
Xilinx LogiCORE® DSP48	0
Block RAM (16k)	0

If you set **Puncture vector source** to `Property`, the design uses fewer LUT and FFS resources.

References

[1] IEEE Std 802.11ac™-2013 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 4: Enhancements for Very High Throughput for Operation in Bands below 6 GHz.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).

OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

See Also

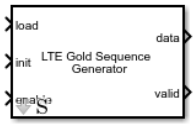
Blocks
Puncturer | Viterbi Decoder

Introduced in R2018b

LTE Gold Sequence Generator

Generate Gold sequence

Library: Wireless HDL Toolbox / Utilities

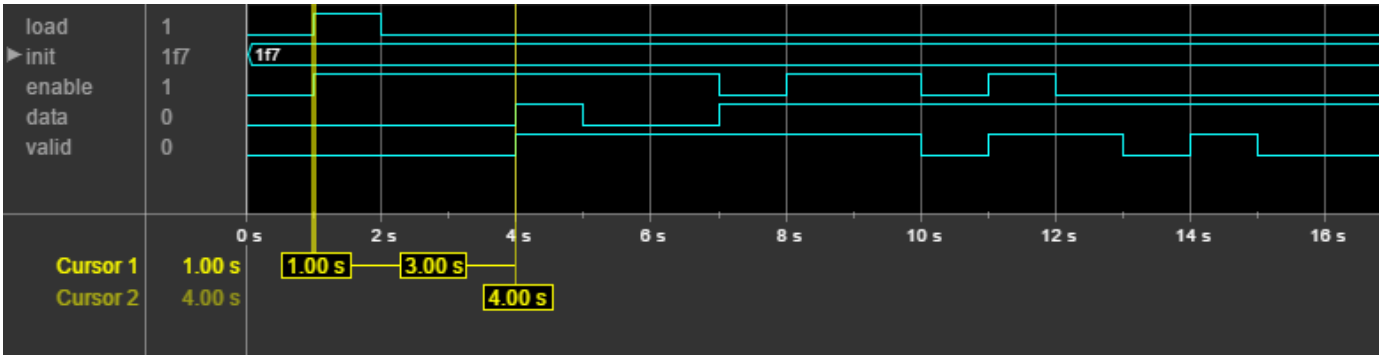


Description

The LTE Gold Sequence Generator block returns Gold sequences generated using the polynomial and shift length specified by LTE standard TS 36.212 [1]. Gold codes are pseudorandom sequences that have high autocorrelation and low crosscorrelation. Due to these properties, Gold codes are widely used in communications systems. For example, they are used to separate different mobile cells operating on the same frequency. LTE systems use a Gold sequence generator for reference symbols and for scrambling/descrambling data, such as in MIB and SIB coding and decoding.

This block provides minimal latency by implementing the shift register initialization in parallel.

Use the **load** control signal to indicate when the **init** value is valid. Use the **enable** control signal to request the next Gold sequence value. The **valid** signal indicates when an output sample is available. The first output sample is ready three cycles after **enable** is asserted. The **data** and **valid** outputs follow the pattern of the **enable** input.



Ports

Input

`load` — Load initial shift register value

scalar

When this control signal is set to `true` (1), the block loads the value on the **init** port into the shift register. You can use this signal to restart the sequence at any point in time.

Data Types: `Boolean`

`init` — Initial shift register value

scalar | vector

Initial shift register value, specified as a `ufix31` number representing the 31 binary values. To generate multiple Gold sequence outputs in parallel, specify a vector of initial values to represent multiple channels.

Data Types: `ufix31`

`enable` — Enable sequence generation
scalar

When this control signal is set to `true` (1), it enables Gold sequence generation.

Data Types: `Boolean`

Output

`data` — Generated Gold sequence
scalar | vector

Generated Gold sequence, returned as a `Boolean` scalar or vector, depending on the size of the **init** input. If **init** is a vector, then **data** is a vector of the same size, representing sequences on independent channels.

Data Types: `Boolean`

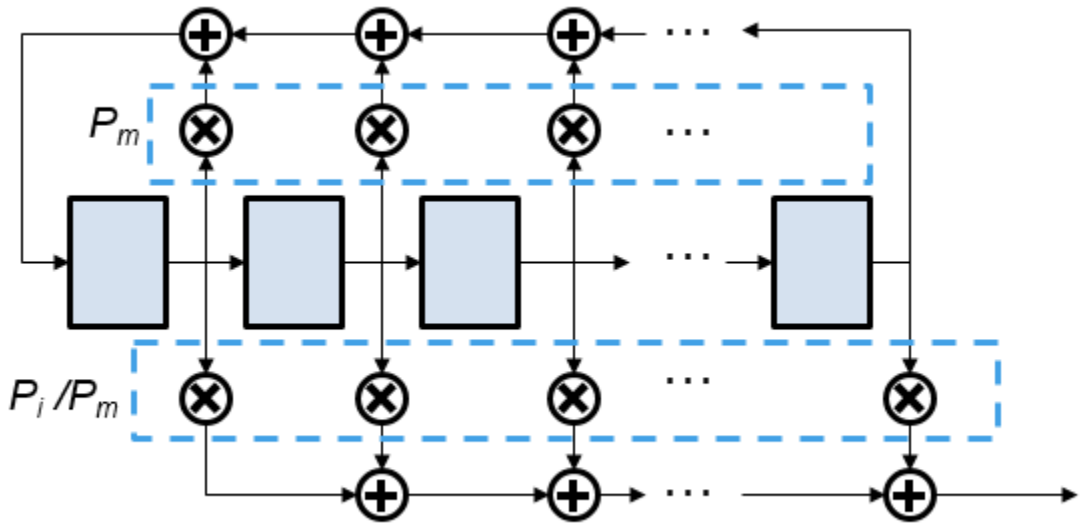
`valid` — Indicates valid output data
scalar

Control signal that indicates when the **data** output port is valid.

Data Types: `Boolean`

Algorithms

To avoid long shift latency, the block applies the initial value as a parallel mask. To calculate the mask, the block divides the initial polynomial by the linear-feedback shift register polynomial.



Performance

These resource and performance data are the synthesis results from the generated HDL targeted to a Xilinx Zynq-7000 ZC706 board. The design achieves a clock frequency of 625 MHz.

Resource	Uses
LUT	86
LUTRAM	0
FFS	107
Block RAM (16K)	0

References

[1] 3GPP TS 36.212. "Multiplexing and channel coding." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA)*. URL: https://www.3gpp.org.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

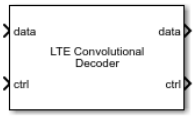
See Also

Topics
“LTE HDL MIB Recovery”

Introduced in R2018a

LTE Convolutional Decoder

Decode convolutional-encoded samples using Viterbi algorithm
Library: Wireless HDL Toolbox / Error Detection and Correction



Description

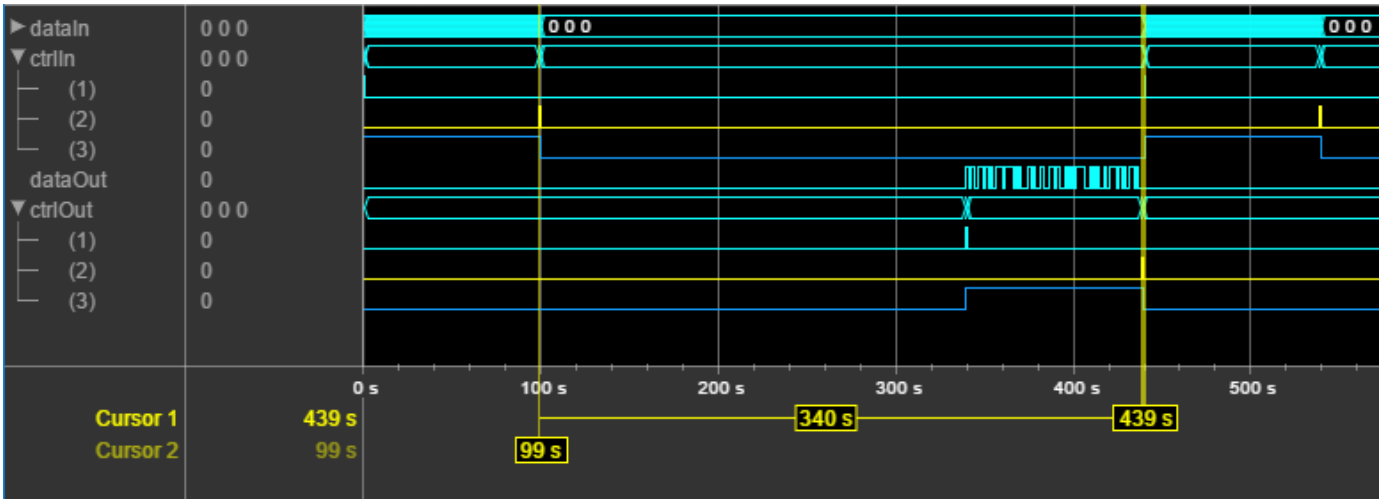
The LTE Convolutional Decoder block implements a wrap-around Viterbi algorithm (WAVA) to decode samples encoded with the tail-biting polynomials specified by LTE standard TS 36.212 [1]. The convolutional code has constraint length 7 and is tail biting with coding rate 1/3 and octal polynomials *G0*=133, *G1*=171 and *G2*=165. The block provides a hardware-optimized architecture and interface.

This block uses a streaming sample interface with a bus for related control signals. This interface enables the block to operate independently of frame size, and to connect easily with other Wireless HDL Toolbox blocks. The block accepts and returns a value representing a single sample, and a bus containing three control signals. These signals indicate the validity of each sample and the boundaries of the frame. To convert a matrix into a sample stream and these control signals, use the Frame To Samples block or the `whdlFramesToSamples` function. For a full description of the interface, see “Streaming Sample Interface”.

The block accepts input samples representing soft or hard decisions. Each sample is a 3-by-1 vector, where the three values represent the bits encoded by the three polynomials, `[G0 G1 G2]`.

Decoding of a message of *M* samples requires 2**M*+140 cycles, assuming contiguous valid input. Therefore, you must leave at least that many idle cycles between input frames. Alternatively, you can use the output signal **ctrl**.`end` to determine when the block is ready for new input.

This waveform shows an input message of 100 samples, with 340 idle cycles between frames. The input data is a vector of three encoded bits. The input and output **ctrl** buses are expanded to show the control signals. `start` and `end` show the frame boundaries, and `valid` qualifies the data samples.



Ports

Input

`data` — Input sample

3-by-1 column vector

Input sample, specified as a 3-by-1 column vector. The block performs soft-decision decoding when the input type is signed fixed point or signed integer, or performs hard-decision decoding when the input type is `Boolean` or `fixdt(0,1,0)`. The block performs unquantized soft-decision decoding for `single` and `double` types, but this mode is not supported for HDL code generation.

For a hardware soft-decision implementation, an integer or fixed-point type that is three or four bits wide is recommended. This input word length achieves decode performance while optimizing timing and resource use when the design is synthesized to an FPGA. The input data type must be less than 16 bits wide. Internal data types are derived from this data type and lower precision types can result in loss of decoding performance.

Values less than zero are most likely a logical `0`, while values greater than zero are most likely a logical `1`. The absolute value determines the level of confidence. For example, the table shows the confidence levels used if the input is `sfix4` (*WL*=4, *FL*=0).

Soft Value	–8, –7, –6, –5, –4, –3, –2, –1	0	1, 2, 3, 4, 5, 6, 7
Logic Level	logical `0`	unknown	logical `1`
Confidence	high → low	none	low → high

Data Types: `Boolean` | `fixdt(0,1,0)` | `fixdt(1,WL,FL)` | `int8` | `int16` | `single` | `double`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, specified as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the input frame
- `end` — Indicates the end of the input frame
- `valid` — Indicates that the data on the input **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

Output

`data` — Output sample
scalar

Output sample, returned as a binary scalar value.

`double` and `single` are supported for simulation but not for HDL code generation.

Data Types: `single` | `double` | `Boolean` | `ufix1`

`ctrl` — Control signals accompanying sample stream
`samplecontrol` bus

Control signals accompanying the sample stream, returned as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the output frame
- `end` — Indicates the end of the output frame
- `valid` — Indicates that the data on the output **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

Parameters

`Maximum message length` — Maximum input frame size
1024 (default) | positive integer

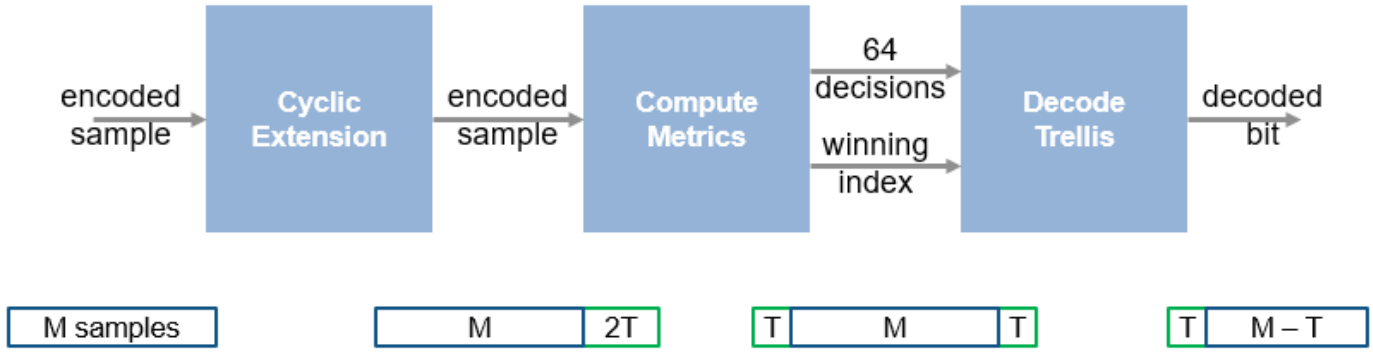
Maximum input frame size, specified as a positive integer from 6 to 2048. The block uses this parameter to determine the amount of RAM required to store intermediate decisions. If you do not specify a power of two, the block uses the next largest power of two.

If an input frame is larger than the specified maximum message length, the block returns a warning.

Algorithms

The LTE Convolutional Decoder block implements a wrap-around Viterbi algorithm (WAVA). The input message is cyclically extended to provide training samples for the Viterbi decoder. This algorithm works with tail-biting convolutional encoders, where the encoder state is the same at the beginning and end of a message.

The diagram shows a high-level view of the decoder architecture.



First, the block extends the message by repeating 2*T message samples, where T is 40 samples. This value of T provides a balance between bit error rate (BER) and optimizing hardware resources. The block uses the extended message to compute branch metrics, state metrics, and branch decisions using add-compare-select operations. The metric word lengths are derived from the data type of the input sample. The block stores a representation of the trellis that is based on the computed decisions. Then it performs traceback decoding. Once the message sample values are decided, the block removes the duplicate training samples and reorders the samples for output.

Performance

These resource and performance data are the synthesis results from the generated HDL targeted to a Xilinx Zynq-7000 ZC706 board. The implementation is for `sfix4` input samples, and a max message size of 1024 (default). HDL code was generated using these options:

- **Adaptive pipelining**: off
- **Minimize clock enables**: on
- **Reset type**: `Synchronous`

The design achieves a clock frequency of 308.45 MHz.

Resource	Uses
LUT	3575
FFS	1776
Xilinx LogiCORE DSP48	0
Block RAM (16K)	5

The input bit width affects the timing and the resources used in metric computation. The maximum message size affects the amount of RAM used in the cyclic extension and traceback stages.

References

[1] 3GPP TS 36.212. "Multiplexing and channel coding." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA)*. URL: https://www.3gpp.org.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

This block does not have any HDL Block Properties.

See Also

Blocks

LTE Convolutional Encoder

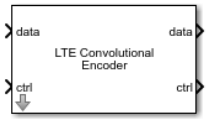
Functions

`lteConvolutionalDecode` | `lteConvolutionalEncode`

Introduced in R2017b

LTE Convolutional Encoder

Encode binary samples using tail-biting convolutional algorithm
Library: Wireless HDL Toolbox / Error Detection and Correction



Description

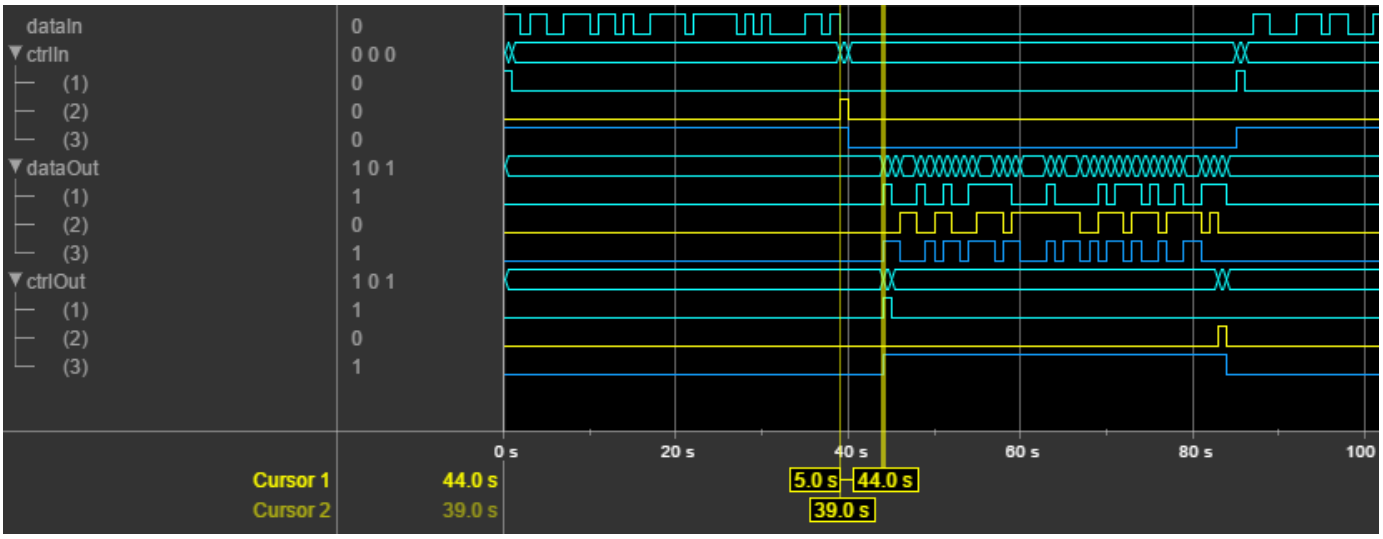
The LTE Convolutional Encoder block implements the encoding polynomials specified by LTE standard TS 36.212 [1]. The convolutional code has constraint length 7 and is tail biting with coding rate 1/3 and octal polynomials *G0*=133, *G1*=171 and *G2*=165. The block provides a hardware-optimized architecture and interface.

This block uses a streaming sample interface with a bus for related control signals. This interface enables the block to operate independently of frame size, and to connect easily with other Wireless HDL Toolbox blocks. The block accepts and returns a value representing a single sample, and a bus containing three control signals. These signals indicate the validity of each sample and the boundaries of the frame. To convert a matrix into a sample stream and these control signals, use the Frame To Samples block or the `whdlFramesToSamples` function. For a full description of the interface, see “Streaming Sample Interface”.

The message size can change dynamically. The encoded output bits for each input bit are returned as a 3-by-1 vector, `[G0 G1 G2]`.

The block takes $M + 5$ cycles to encode a frame of M samples. Therefore, you must leave $M + 5$ idle cycles between input frames. Alternatively, you can use the output signal **ctrl**.`end` to determine when the block is ready for new input.

This waveform shows an input message of 40 samples, with 45 idle cycles between frames. The output data is a vector of three encoded bits. The input and output **ctrl** buses are expanded to show the control signals. `start` and `end` show the frame boundaries, and `valid` qualifies the data samples.



Ports

Input

`data` — Input sample

scalar

Input sample, specified as a binary scalar. `double` and `single` are supported for simulation but not for HDL code generation.

Data Types: `single` | `double` | `Boolean` | `ufix1`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, specified as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the input frame
- `end` — Indicates the end of the input frame
- `valid` — Indicates that the data on the input **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

Output

`data` — Encoded sample

3-by-1 column vector

Encoded sample, returned as a 3-by-1 column vector. Each encoded sample is represented by three bits, one from each encoder polynomial.

The output data type matches the input data type.

Data Types: `single` | `double` | `Boolean` | `ufix1`

`ctrl` — Control signals accompanying sample stream
`samplecontrol` bus

Control signals accompanying the sample stream, returned as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the output frame
- `end` — Indicates the end of the output frame
- `valid` — Indicates that the data on the output **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

Parameters

`Maximum message length` — Maximum input frame size
1024 (default) | positive integer

Maximum input frame size, specified as a positive integer from 6 to 2^{16}. This parameter defines the required amount of frame memory. If you do not specify a power of two, the block uses the next largest power of two.

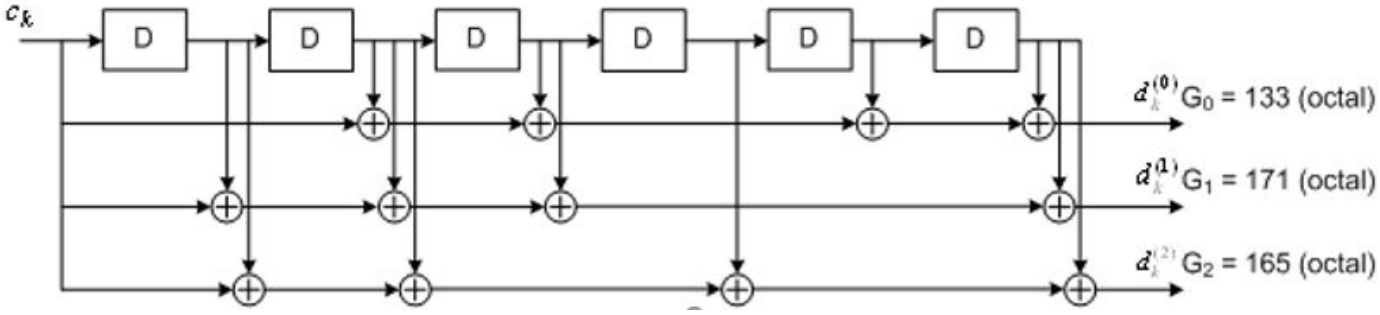
If an input frame is larger than the specified maximum message length, the block returns a warning.

Tips

- You cannot use this block inside an Enabled Subsystem or Resettable Subsystem.

Algorithms

The block implements a tail-biting convolutional encoder as specified by LTE standard TS 36.212 [1].



Performance

These resource and performance data are the synthesis results from the generated HDL targeted to a Xilinx Zynq-7000 ZC706 board. The implementation is for a max message size of 1024 (default). The design achieved a clock frequency of 476.2 MHz.

Resource	Uses
LUT	79
LUTRAM	16
FFS	46
Block RAM (16K)	0

The maximum message size affects the amount of RAM used.

References

[1] 3GPP TS 36.212. "Multiplexing and channel coding." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA)*. URL: https://www.3gpp.org.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

See Also

Blocks
LTE Convolutional Decoder

Functions
`lteConvolutionalDecode` | `lteConvolutionalEncode`

Introduced in R2017b

LTE CRC Decoder

Detect errors in input samples using checksum
Library: Wireless HDL Toolbox / Error Detection and Correction

Description

The LTE CRC Decoder block calculates a cyclic redundancy check (CRC) and compares it with the appended checksum, for each frame of streaming data samples. You can select from the polynomials specified by LTE standard TS 36.212 [1]. The block provides a hardware-optimized architecture and interface.

This block uses a streaming sample interface with a bus for related control signals. This interface enables the block to operate independently of frame size, and to connect easily with other Wireless HDL Toolbox blocks. The block accepts and returns a value representing a single sample, and a bus containing three control signals. These signals indicate the validity of each sample and the boundaries of the frame. To convert a matrix into a sample stream and these control signals, use the Frame To Samples block or the `whdlFramesToSamples` function. For a full description of the interface, see “Streaming Sample Interface”.

Ports

Input

`data` — Input sample

binary scalar | unsigned integer scalar | binary vector

Input sample, specified as a binary scalar, unsigned integer scalar, or binary vector. The vector size must be less than or equal to the length of the polynomial. The CRC length also must be divisible by the vector size. For example, for polynomial type `CRC24A`, the valid vector sizes are 24, 12, 8, 6, 4, 3, 2, and 1. An integer input is interpreted as a binary word. For example, vector input `[0 0 0 1 0 0 1 1]` is equivalent to `uint8` input `19`.

`double` and `single` are supported for simulation but not for HDL code generation.

Data Types: `single` | `double` | `Boolean` | `ufix1` | `uint8` | `uint16` | `uint32`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, specified as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the input frame
- `end` — Indicates the end of the input frame

- `valid` — Indicates that the data on the input **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

Output

`data` — Output sample
binary scalar | integer scalar | binary vector

Output sample, returned a binary scalar, unsigned integer scalar, or binary vector of the same data type and size as the input samples. The checksum is removed from the end of the frame.

`double` and `single` binary values are supported for simulation but not for HDL code generation.

Data Types: `single` | `double` | `Boolean` | `ufix1` | `uint8` | `uint16` | `uint32` | `ufixN`

`ctrl` — Control signals accompanying sample stream
`samplecontrol` bus

Control signals accompanying the sample stream, returned as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the output frame
- `end` — Indicates the end of the output frame
- `valid` — Indicates that the data on the output **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

`err` — Indicator of checksum mismatch
binary scalar | integer scalar

Indicator of checksum mismatch, returned as a binary scalar or an integer scalar. If you select **Full checksum mismatch**, this port returns the integer XOR result of the calculated checksum against the appended checksum. The **err** value is valid when **ctrl**.`end` is 1 (`true`). The data type of this port matches the data type of the input samples.

Data Types: `single` | `double` | `Boolean` | `ufix1` | `uint8` | `uint16` | `uint32` | `ufixN`

Parameters

`CRC Type` — Encode polynomial
`CRC16` (default) | `CRC8` | `CRC24A` | `CRC24B`

The encode polynomial options are the four CRC types described in the LTE standard TS 36.212 [1], Section 5.1.1.

`Full checksum mismatch` — Return bit-by-bit mismatch information
off (default) | on

When this parameter is not selected, the **err** port returns a Boolean value indicating whether any checksum bits are mismatched, after applying **CRC Mask**. When this parameter is selected, the **err** port returns an integer that represents the locations of bit mismatches in the checksum.

`CRC Mask` — Mask applied to checksum
0 (default) | integer from 0 to $2^{CRCLength} - 1$

Mask applied to checksum, specified as an integer representing a binary word from 0 to $2^{CRCLength} - 1$. This mask is typically a Radio Network Temporary Identifier (RNTI).

Dependencies

This parameter appears when **Full checksum mismatch** is cleared.

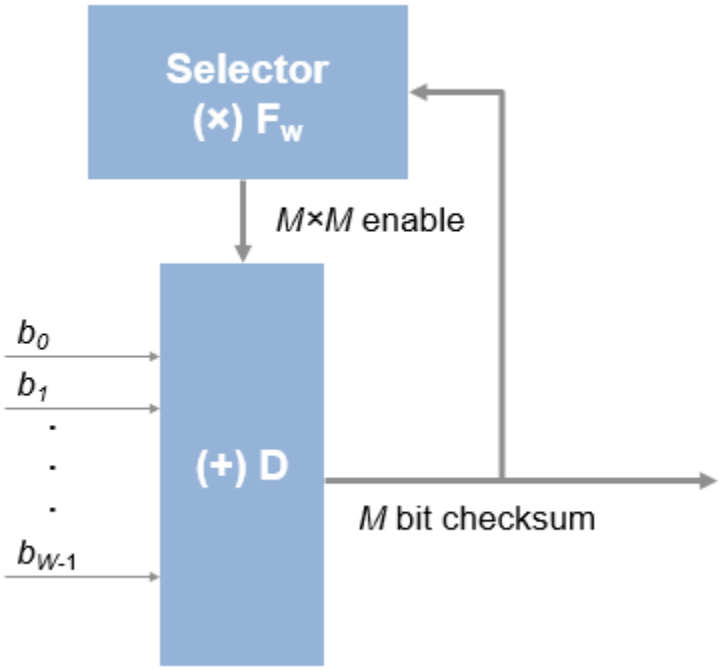
Algorithms

When you use vector or integer input, the block implements a parallel CRC algorithm [2]. The implementation is the same as the algorithm used by the Communications Toolbox™ blocks General CRC Generator HDL Optimized and General CRC Syndrome Detector HDL Optimized.

To provide high throughput for modern communications systems, the block implements the CRC algorithm with a parallel architecture. This architecture recursively calculates M bits of a CRC checksum for each W input bits. At the end of the frame, the final checksum result is appended to the message. For a polynomial length of M, the recursive checksum calculation for W bits in parallel is

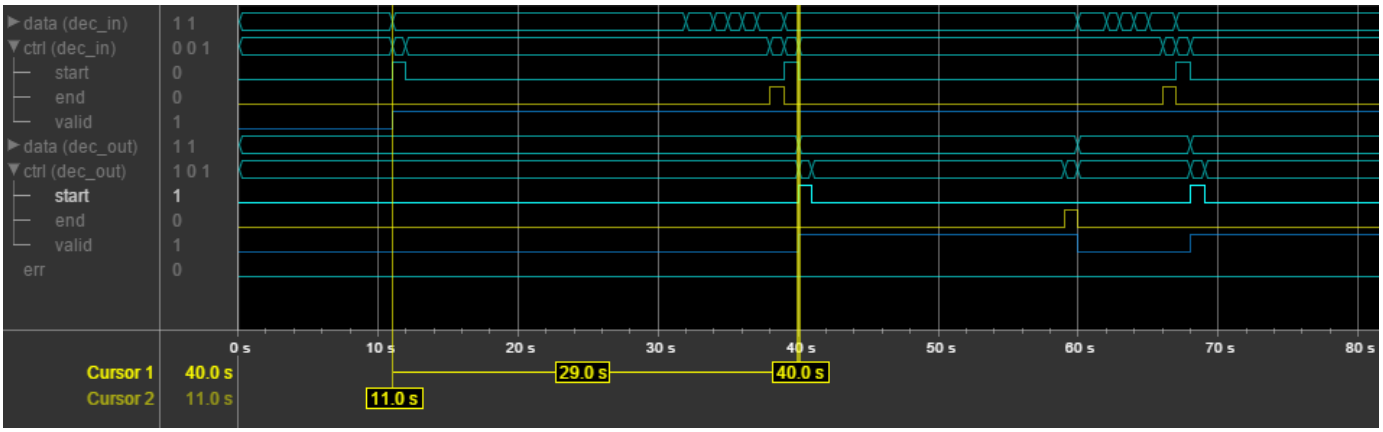
$$X' = F_W(\times)X(+)D$$

F_W is an M-by-M matrix that selects elements of the current state for the polynomial calculation with the new input bits. D is an M-element vector that provides the new input bits, ordered in relation to the generator polynomial and padded with zeros. The block implements the (×) with logical AND and (+) with logical XOR.



Latency

This waveform shows a 40-sample frame, input two samples at a time, encoded with a CRC16 polynomial. There is no gap between input frames. The output stream has removed the checksum, so there are eight cycles between output frames. The latency of the decoder is `3*CRCLength/InputSize + 5`, assuming contiguous valid input samples.



Performance

These resource and performance data are the synthesis results from the generated HDL targeted to a Xilinx Zynq-7000 ZC706 board. The implementation is for a `CRC24` polynomial, with no CRC Mask or output checksum mismatch, and scalar input. The design achieves 526.31 MHz clock frequency.

Resource	Uses
LUT	210
LUTRAM	8
FFS	305
Block RAM (16K)	0

A larger input vector size increases throughput and increases resource use.

References

[1] 3GPP TS 36.212. "Multiplexing and channel coding." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA)*. URL: https://www.3gpp.org.

[2] Campobello, Giuseppe, Giuseppe Patane, and Marco Russo. "Parallel CRC Realization." *IEEE Transactions on Computers*. Vol. 52, No. 10, October 2003, pp. 1312–1319.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

See Also

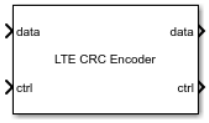
Blocks
LTE CRC Encoder

Functions
`lteCRCDecode` | `lteCRCEncode`

Introduced in R2017b

LTE CRC Encoder

Generate checksum and append to input sample stream
Library: Wireless HDL Toolbox / Error Detection and Correction



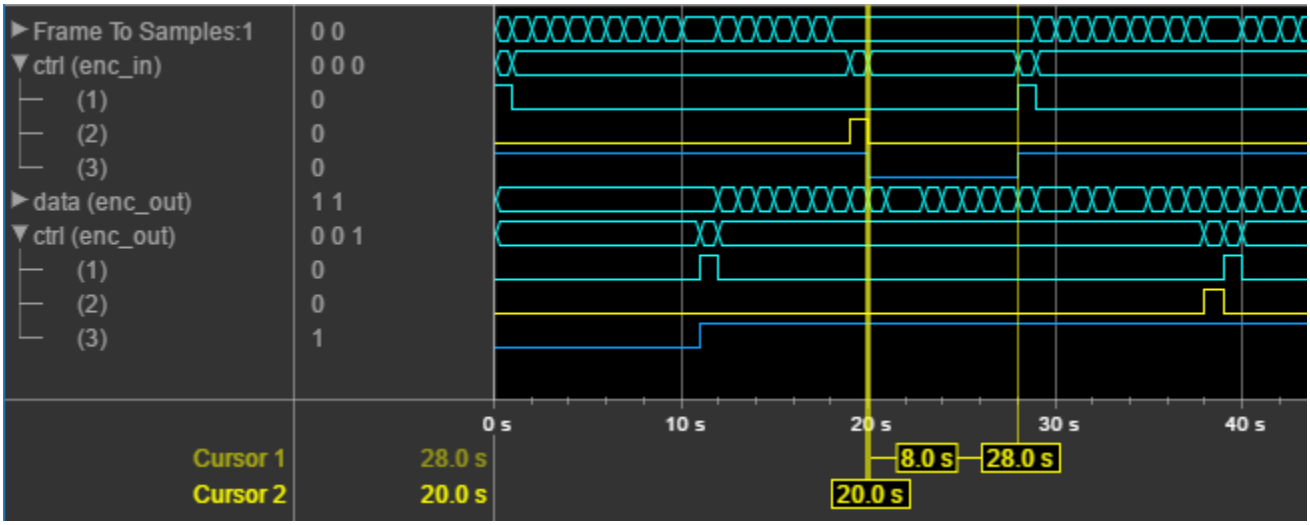
Description

The LTE CRC Encoder block calculates and appends a cyclic redundancy check (CRC) checksum for each frame of streaming data samples. You can select from the polynomials specified by LTE standard TS 36.212 [1]. The block provides a hardware-optimized architecture and interface.

This block uses a streaming sample interface with a bus for related control signals. This interface enables the block to operate independently of frame size, and to connect easily with other Wireless HDL Toolbox blocks. The block accepts and returns a value representing a single sample, and a bus containing three control signals. These signals indicate the validity of each sample and the boundaries of the frame. To convert a matrix into a sample stream and these control signals, use the Frame To Samples block or the `whdlFramesToSamples` function. For a full description of the interface, see “Streaming Sample Interface”.

You must not apply another frame before the previous frame has completed. The hardware-friendly algorithm adds `(`*`CRCLength`*` + 3)/`*`InputSize`* cycles of latency. To account for the additional cycles of the appended checksum samples, and the latency, you must apply a minimum spacing of `(2*`*`CRCLength`*` + 3)/`*`InputSize`* between input frames. Alternatively, you can use the output signal **ctrl**.`end` to determine when the block is ready for new input. If you apply the next frame too early, the **ctrl**.`start` signal resets the checksum calculation and truncates the previous frame.

This waveform shows a 40-sample frame, input two samples at a time to a CRC16 encoder. The gap between the input frames is therefore 8 cycles. Due to the insertion of the checksum, the output **ctrl**.`valid` signal stays continuously high with no gaps between frames. The input and output **ctrl** buses are expanded to show the control signals. `start` and `end` show the frame boundaries, and `valid` qualifies the data samples.



Ports

Input

`data` — Input sample

binary scalar | unsigned integer scalar | binary vector

Input sample, specified as a binary scalar, unsigned integer scalar, or binary vector. The vector size, *InputSize*, must be less than or equal to the length of the polynomial. The CRC length also must be divisible by the vector size. For example, for polynomial type `CRC24A`, the valid vector sizes are 24, 12, 8, 6, 4, 3, 2, and 1. An integer input is interpreted as a binary word. For example, vector input `[0 0 0 1 0 0 1 1]` is equivalent to `uint8` input `19`.

`double` and `single` are supported for simulation but not for HDL code generation.

Data Types: `single` | `double` | `Boolean` | `ufix1` | `uint8` | `uint16` | `uint32` | `ufixN`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, specified as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the input frame
- `end` — Indicates the end of the input frame
- `valid` — Indicates that the data on the input **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

Output

`data` — Output sample

binary scalar | integer scalar | binary vector

Output sample, returned as a binary scalar, integer scalar, or binary vector of the same data type and size as the input sample. The block appends the calculated and masked checksum at the end of each frame.

`double` and `single` binary values are supported for simulation but not for HDL code generation.

Data Types: `single` | `double` | `Boolean` | `ufix1` | `uint8` | `uint16` | `uint32` | `ufixN`

`ctrl` — Control signals accompanying sample stream
`samplecontrol` bus

Control signals accompanying the sample stream, returned as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the output frame
- `end` — Indicates the end of the output frame
- `valid` — Indicates that the data on the output **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

Parameters

`CRC Type` — Encode polynomial
`CRC16` (default) | `CRC8` | `CRC24A` | `CRC24B`

The encode polynomial options are the four CRC types described in the LTE standard TS 36.212 [1], Section 5.1.1.

`CRC Mask` — Mask applied to checksum
`0` (default) | integer from 0 to $2^{CRCLength} - 1$

Mask applied to checksum, specified as an integer representing a binary word from 0 to $2^{CRCLength} - 1$. This mask is typically a Radio Network Temporary Identifier (RNTI).

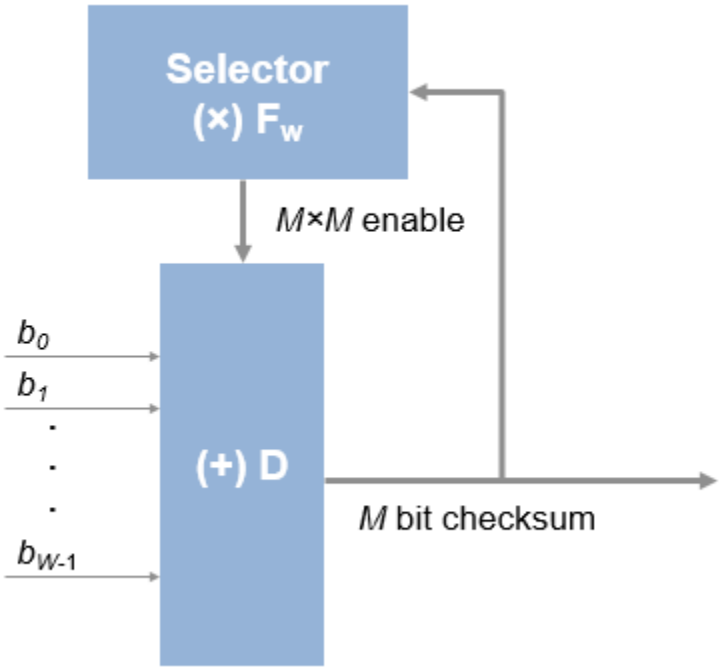
Algorithms

When you use vector or integer input, the block implements a parallel CRC algorithm [2]. The implementation is the same as the algorithm used by the Communications Toolbox blocks General CRC Generator HDL Optimized and General CRC Syndrome Detector HDL Optimized.

To provide high throughput for modern communications systems, the block implements the CRC algorithm with a parallel architecture. This architecture recursively calculates M bits of a CRC checksum for each W input bits. At the end of the frame, the final checksum result is appended to the message. For a polynomial length of M, the recursive checksum calculation for W bits in parallel is

$$X' = F_W(\times)X(+)D$$

F_W is an M-by-M matrix that selects elements of the current state for the polynomial calculation with the new input bits. D is an M-element vector that provides the new input bits, ordered in relation to the generator polynomial and padded with zeros. The block implements the (×) with logical AND and (+) with logical XOR.



Latency

The latency from start of input frame to start of output frame is `(`*`CRCLength`*` + 3)/`*`InputSize` cycles. The latency from end of input frame to end of output frame is (2*`CRCLength` + 3)/*InputSize* to account for appending the checksum.

Performance

These resource and performance data are the synthesis results from the generated HDL targeted to a Xilinx Zynq-7000 ZC706 board. The implementation is for a `CRC24` polynomial and scalar input. The design achieves 588.3 MHz clock frequency.

Resource	Uses
LUT	121
LUTRAM	3
FFS	132
Block RAM (16K)	0

A larger input vector size increases throughput and increases resource use.

References

[1] 3GPP TS 36.212. "Multiplexing and channel coding." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA)*. URL: https://www.3gpp.org.

[2] Campobello, Giuseppe, Giuseppe Patane, and Marco Russo. "Parallel CRC Realization." *IEEE Transactions on Computers*. Vol. 52, No. 10, October 2003, pp. 1312–1319.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

See Also

Blocks

LTE CRC Decoder

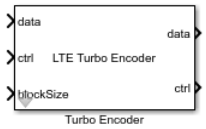
Functions

`lteCRCDecode` | `lteCRCEncode`

Introduced in R2017b

LTE Turbo Encoder

Encode binary samples using turbo algorithm
Library: Wireless HDL Toolbox / Error Detection and Correction



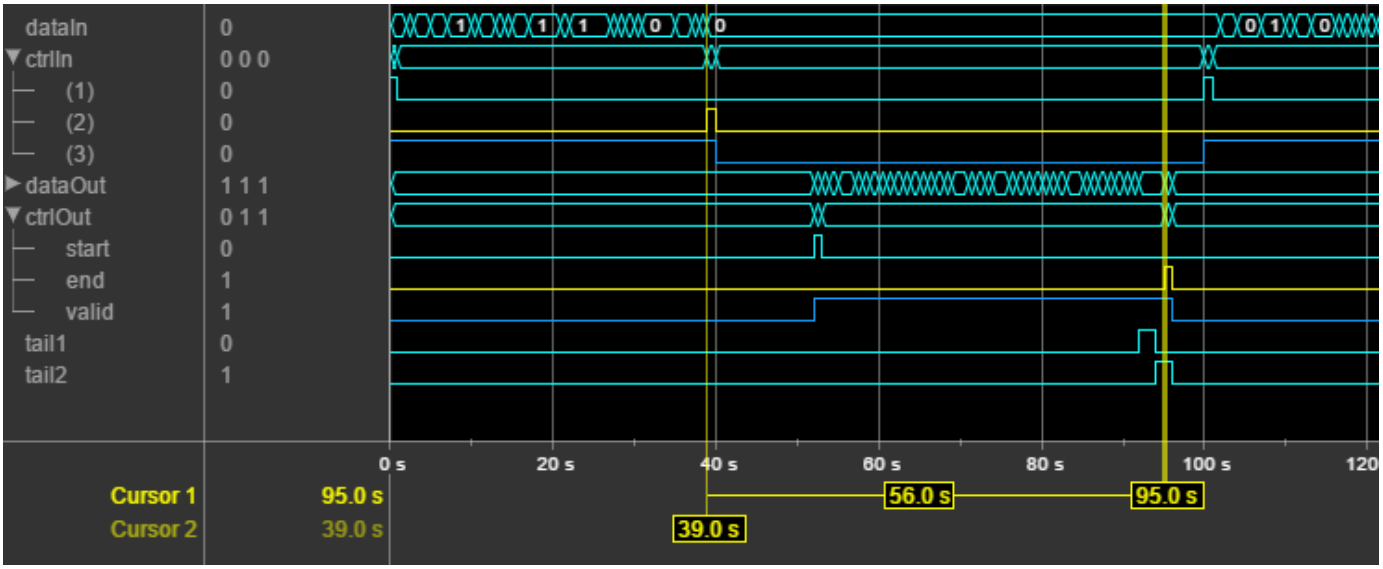
Description

The LTE Turbo Encoder block implements the turbo encoder described by LTE standard TS 36.212 [1] using an interface and architecture optimized for HDL code generation and hardware deployment. The encoder is a parallel concatenated convolutional code (PCCC) with two 8-state constituent encoders and an internal interleaver. The first encoder operates on the input data stream, and the second encoder operates on an interleaved version of the input data. The block terminates each encoder output with independent tail bits. The coding rate is 1/3. The encoded output bits for each input bit are returned as a 3-by-1 vector, `[S P1 P2]`. In this vector, `S` is the systematic bit, and `P1` and `P2` are the parity bits from the two encoders.

This block uses a streaming sample interface with a bus for related control signals. This interface enables the block to operate independently of frame size, and to connect easily with other Wireless HDL Toolbox blocks. The block accepts and returns a value representing a single sample, and a bus containing three control signals. These signals indicate the validity of each sample and the boundaries of the frame. To convert a matrix into a sample stream and these control signals, use the Frame To Samples block or the `whdlFramesToSamples` function. For a full description of the interface, see “Streaming Sample Interface”.

The block can accept new input data after the previous frame is complete. Apply input frames with at least *BlockSize* + 16 idle cycles between them. The 16 cycles consists of 12 cycles for pipeline delays in the algorithm, and 4 cycles of tail bits. This latency does not vary with block size. Or, you can use the output signal **ctrl**.`end` to determine when the block is ready for new input.

This waveform shows an input frame of 40 samples, with 57 idle cycles between frames. The input and output **ctrl** buses are expanded to show the control signals. `start` and `end` show the frame boundaries, and `valid` qualifies the data samples. The optional **tail1** and **tail2** signals indicate the cycles when the tail bits from each encoder are valid.



Ports

Input

`data` — Input sample

scalar

Input sample, specified as a binary scalar. `double` and `single` are supported for simulation but not for HDL code generation.

Data Types: `single` | `double` | `Boolean` | `ufix1`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, specified as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the input frame
- `end` — Indicates the end of the input frame
- `valid` — Indicates that the data on the input **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

`blockSize` — Turbo code block size

integer

Turbo code block size, specified as an integer. This value must be one of the 188 values specified in the LTE standard, from 40 to 6144 in these intervals: `[40:8:512 528:16:1024 1056:32:2048 2112:64:6144]`.

Dependencies

This port appears when you set **Block size source** to `Input port`.

Data Types: `single` | `double` | `uint16` | `fixdt(0,13,0)`

Output

`data` — Encoded sample stream

3-by-1 column vector

Encoded sample stream, returned as a 3-by-1 column vector. Each encoded sample is represented by one systematic bit and two parity bits.

The output data type matches the input data type.

Data Types: `single` | `double` | `Boolean` | `ufix1`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, returned as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the output frame
- `end` — Indicates the end of the output frame
- `valid` — Indicates that the data on the output **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

`tail1, tail2` — Indicate trellis termination cycles

scalar

Use the optional **tail1** and **tail2** output ports to indicate the location of the tail bits in the output data stream. These signals are 1 (`true`) for the cycles that correspond to the tail bits for each encoder.

The block returns the tail bits in the order specified by the LTE standard TS 36.212 [1]. Each encoder returns two cycles of encoded tail bits.

Cycle	1	2	3	4
tail1	1	1	0	0
tail2	0	0	1	1
data	[E1in$_K$ E1out$_K$ E1in$_{K+1}$]	[E1out$_{K+1}$ E1in$_{K+2}$ E1out$_{K+2}$]	[E2in$_K$ E2out$_K$ E2in$_{K+1}$]	[E2out$_{K+1}$ E2in$_{K+2}$ E2out$_{K+2}$]

Dependencies

Enable these ports by selecting **Enable trellis termination valid ports**.

Data Types: `Boolean`

Parameters

`Block size source` — How to specify the block size
`Input port` (default) | `Property`

Select whether you specify the block size with an input port or enter a fixed value as a parameter. If you select `Property`, the **Block size** parameter appears. If you select `Input port`, the **blockSize** port appears.

`Block size` — Turbo code block size
`6144` (default)

Turbo code block size, specified as an integer. This value must be one of the 188 values specified in the LTE standard, from 40 through 6144 in these intervals: `[40:8:512 528:16:1024 1056:32:2048 2112:64:6144]`. This value is registered for each frame, when **ctrl**.`start` = 1 (`true`).

Dependencies

This parameter appears when you set **Block size source** to `Property`.

`Enable trellis termination valid ports` — Enable ports that indicate the tail bit output samples
off (default) | on

When you select this parameter, the **tail1** and **tail2** ports appear on the block. These ports return control signals that indicate the cycles when the output samples are the tail bits for each encoder.

Tips

- You cannot use this block inside an Enabled Subsystem or Resettable Subsystem.

Algorithms

For a hardware implementation, storing the interleave indices is not practical. Supporting the 188 LTE block sizes would require 4 Mb of memory. Therefore, the algorithm uses the interleave specification to compute the indexes from the block size. This equation defines the interleave pattern:

$$\prod(i) = (f_1 \cdot i + f_2 \cdot i^2) \mathrm{mod}$$

K is the block size, $i = 0, 1, \ldots, (K - 1)$, and f_1 and f_2 are defined in the LTE standard TS 36.212 [1].

Calculation of the indexes is simplified based on these equations:

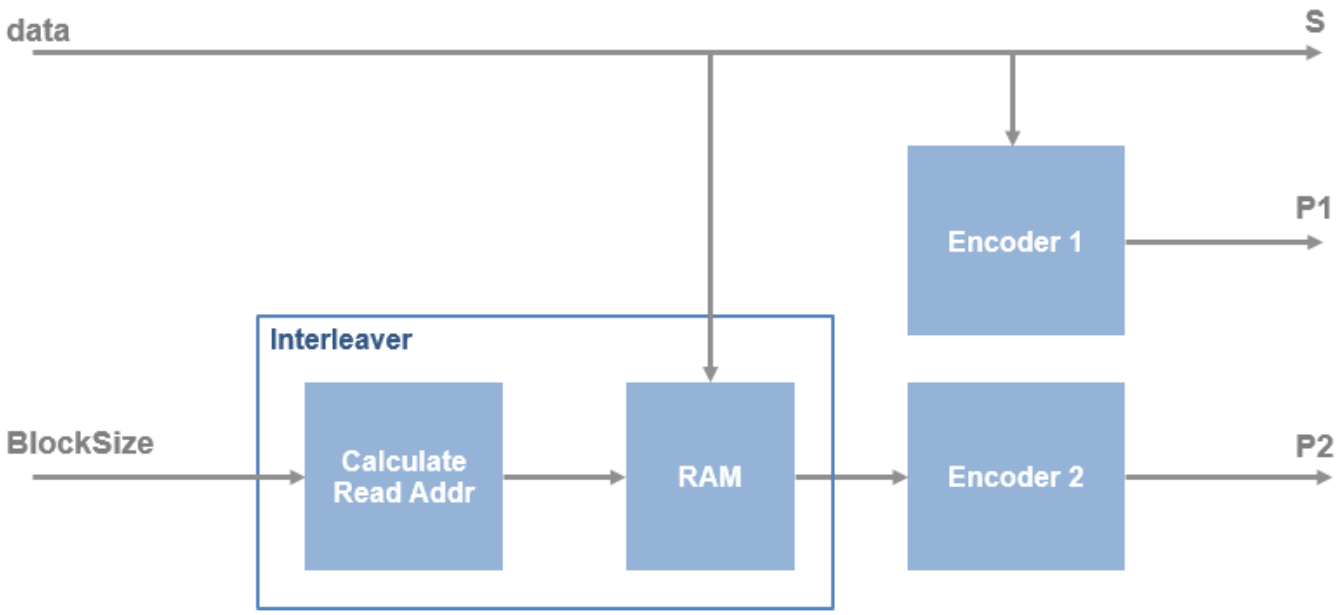
$$\pi(i+1) = \begin{cases} \pi(i) + g(i) & \text{if } \pi(i) + g(i) < \\ \pi(i) + g(i) - K & \text{otherwise} \end{cases}$$

$$\pi(0) = 0$$

$$g(i+1) = \begin{cases} g(i) + 2f_2 & \text{if } g(i) + 2f_2 < \\ g(i) + 2f_2 - K & \text{otherwise} \end{cases}$$

$$g(0) = f_1 + f_2$$

Therefore, the block stores f_1 and f_2 in memory, and uses those two constants and four adders to calculate the interleave indexes.



When **Block size source** is set to `Property`, the block uses two constant coefficients to derive the read addresses for the fixed block size. When **Block size source** is set to `Input port`, the algorithm saves the 188 pairs of coefficients in a ROM (< 5 Kb). Then the block reads the matching pair at run time to derive the interleave memory read addresses.

Performance

These resource and performance data are the synthesis results from the generated HDL targeted to a Xilinx Zynq-7000 ZC706 board. The implementation is for a fixed block size of 6144 samples. The design achieves 312.5 MHz clock frequency.

Resource	Uses
LUT	253
LUTRAM	2
FFS	222
BRAM	0.5

The RAM size grows with block size. When you enable the block size port, RAM sufficient for the largest block size is used.

References

[1] 3GPP TS 36.212. "Multiplexing and channel coding." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA)*. URL: https://www.3gpp.org.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).

OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

See Also

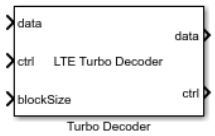
Blocks
LTE Turbo Decoder

Functions
`lteDLSCHInfo` | `lteTurboDecode` | `lteTurboEncode`

Introduced in R2017b

LTE Turbo Decoder

Decode turbo-encoded samples
Library: Wireless HDL Toolbox / Error Detection and Correction



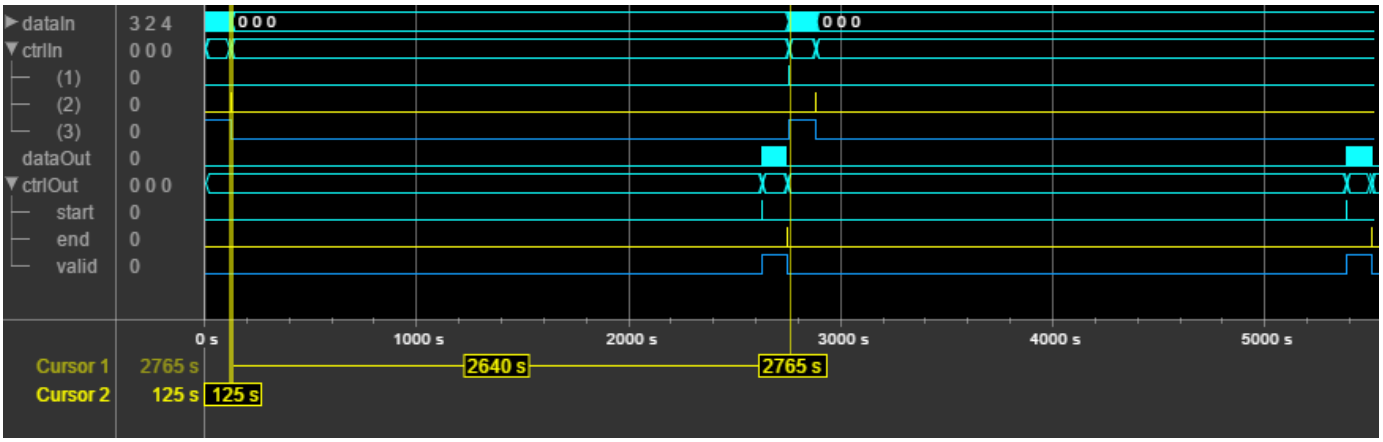
Description

The LTE Turbo Decoder block implements the turbo decoder required by LTE standard TS 36.212 [1] and provides an interface and architecture optimized for HDL code generation and hardware deployment. The block iterates over two MAX decoders. You can specify the number of iterations. The coding rate is 1/3. The block accepts encoded bits as a 3-by-1 vector of soft-coded values, `[S P1 P2]`. In this vector, `S` is the systematic bit, and `P1` and `P2` are the parity bits from the two encoders.

This block uses a streaming sample interface with a bus for related control signals. This interface enables the block to operate independently of frame size, and to connect easily with other Wireless HDL Toolbox blocks. The block accepts and returns a value representing a single sample, and a bus containing three control signals. These signals indicate the validity of each sample and the boundaries of the frame. To convert a matrix into a sample stream and these control signals, use the Frame To Samples block or the `whdlFramesToSamples` function. For a full description of the interface, see “Streaming Sample Interface”.

The block can accept the next frame only after it has completed decoding the previous frame. You must leave *Iterations**2**HalfIterationLatency*+*BlockSize*+4 idle cycles between input frames. The half-iteration latency is described in the “Algorithms” on page 1-62 section. Alternatively, you can use the output signal **ctrl**.`end` to determine when the block is ready for new input.

This waveform shows an input frame of 120 samples (+ 4 tail bits), and 2632 idle cycles between frames. Each input sample is a vector of three fixed-point soft-decision values. The input and output **ctrl** buses are expanded to show the control signals. `start` and `end` show the frame boundaries, and `valid` qualifies the data samples.



Ports

Input

`data` — Input sample
three-element vector

Input sample, specified as a three-element integer vector. The values represent soft-coded probabilities. If the value is negative, the bit is more likely to be 0. If the value is positive, the bit is more likely to be 1. The first element is the sequential bit, and the other two elements are parity bits. The block expects input frames of *BlockSize* + 12 samples. This frame size includes the tail bits, in the order specified by LTE standard TS 36.212 [1].

For a hardware implementation, use a fixed-point type with two or three integer bits and one to four fractional bits. Internal data types are derived from this data type, and lower precision types can result in loss of decoding precision. If the input data type has zero fractional bits or less than two integer bits, the block returns a warning. `double` and `single` are supported for simulation but not for HDL code generation.

Data Types: `fixdt(1,WL,FL)` | `single` | `double`

`ctrl` — Control signals accompanying sample stream
`samplecontrol` bus

Control signals accompanying the sample stream, specified as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the input frame
- `end` — Indicates the end of the input frame
- `valid` — Indicates that the data on the input **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

`blockSize` — Turbo code block size
integer

Turbo code block size, specified as an integer. This value must be one of the 188 values specified in the LTE standard, from 40 to 6144 in these intervals: `[40:8:512 528:16:1024 1056:32:2048 2112:64:6144]`.

Dependencies

This port appears when you set **Block size source** to `Input port`.

Data Types: `single` | `double` | `uint16` | `fixdt(0,13,0)`

Output

`data` — Output sample
scalar

Output sample, returned as a binary scalar. `double` and `single` are supported for simulation but not for HDL code generation.

Data Types: `single` | `double` | `Boolean` | `ufix1`

`ctrl` — Control signals accompanying sample stream
`samplecontrol` bus

Control signals accompanying the sample stream, returned as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the output frame
- `end` — Indicates the end of the output frame
- `valid` — Indicates that the data on the output **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

Parameters

`Block size source` — How to specify the block size
`Input port` (default) | `Property`

Select whether you specify the block size with an input port or enter a fixed value as a parameter. If you select `Property`, the **Block size** parameter appears. If you select `Input port`, the **blockSize** port appears.

`Block size` — Turbo code block size
`6144` (default)

Turbo code block size, specified as an integer. This value must be one of the 188 values specified in the LTE standard, from 40 through 6144 in these intervals: `[40:8:512 528:16:1024 1056:32:2048 2112:64:6144]`. This value is registered for each frame, when **ctrl**.`start` = 1 (`true`).

Dependencies

This parameter appears when you set **Block size source** to `Property`.

`Number of decoding iterations` — How many MAX decoding iterations to perform
`6` (default) | positive integer

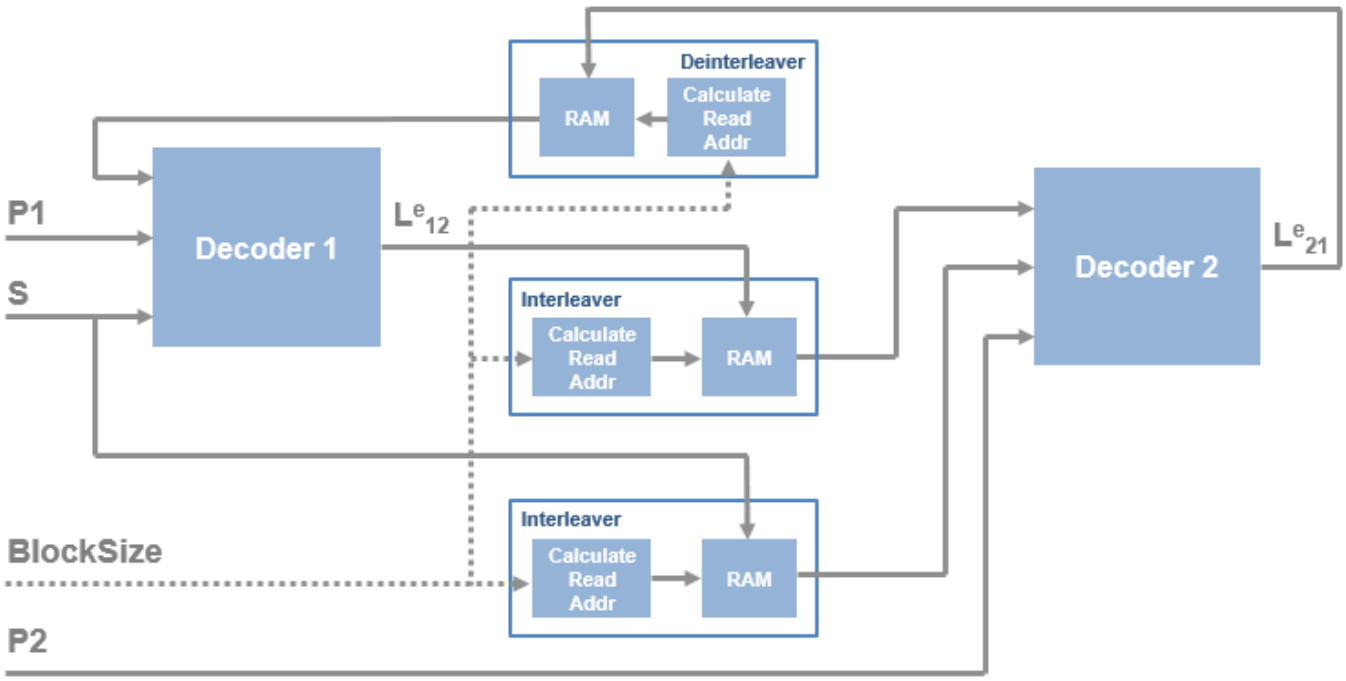
How many MAX decoding iterations to perform, specified as a positive integer. The decoder implementation uses a single MAX decoder, with the output data of each half-iteration routed back to the input. A higher iteration count increases accuracy and adds latency. After about 15 iterations, the algorithm does not provide further accuracy.

Algorithms

The block implements an iterative decode algorithm using a single decoder and single interleaver.

This diagram shows the conceptual algorithm for one iteration. Although the diagram shows two decoders and three interleavers, the block actually implements the algorithm using only one decoder and one interleaver. The decoder performs one half-iteration and interleaves the results. Then the

output is routed back to the input for the next half-iteration. The interleaver computes the interleave indexes from the block size. For details of the interleaver implementation, see LTE Turbo Encoder.



The odd half-iterations compute the likelihood ratio from uninterleaved bits (P1, S, and deinterleaved results of P2 decoding). The even half-iterations compute the likelihood ratio from the interleaved bits (P2 and interleaved results of P1 and S decoding).

The decoder block uses the BCJR algorithm to find the likelihood ratio of a particular bit [2].

$$L(u_k) \triangleq \log\left(\frac{P(u_k = +1 \mid y)}{P(u_k = +1 \mid y)}\right)$$

$$= \log\left(\frac{\sum_{s^+} p(s_{k-1} = s', s_k = s, y)/p(}{\sum_{s^-} p(s_{k-1} = s', s_k = s, y)/p(}\right.$$

The probabilities can also be represented in terms of current and future states:

$$p(s', s, y) = \alpha_{k-1}(s') \cdot \gamma_k(s', s) \cdot \beta_k($$

$$\alpha_{k-1}(s') = P(s', y_{<k})$$

$$\gamma_k(s', s) = P(y_k, s \mid s')$$

$$\beta_k(s) = P(y_{<k} \mid s)$$

The α probability represents the previous state, β represents the current state probability, and ɣ represents the next state. The algorithm calculates ɣ from the input values. The α and β probabilities are calculated using forward and backward recursion over the possible states of the trellis, and also depend on ɣ. All calculations are done in the `log` domain.

$$\alpha_k = \alpha_{k-1}(s') \cdot \gamma_k(s', s)$$

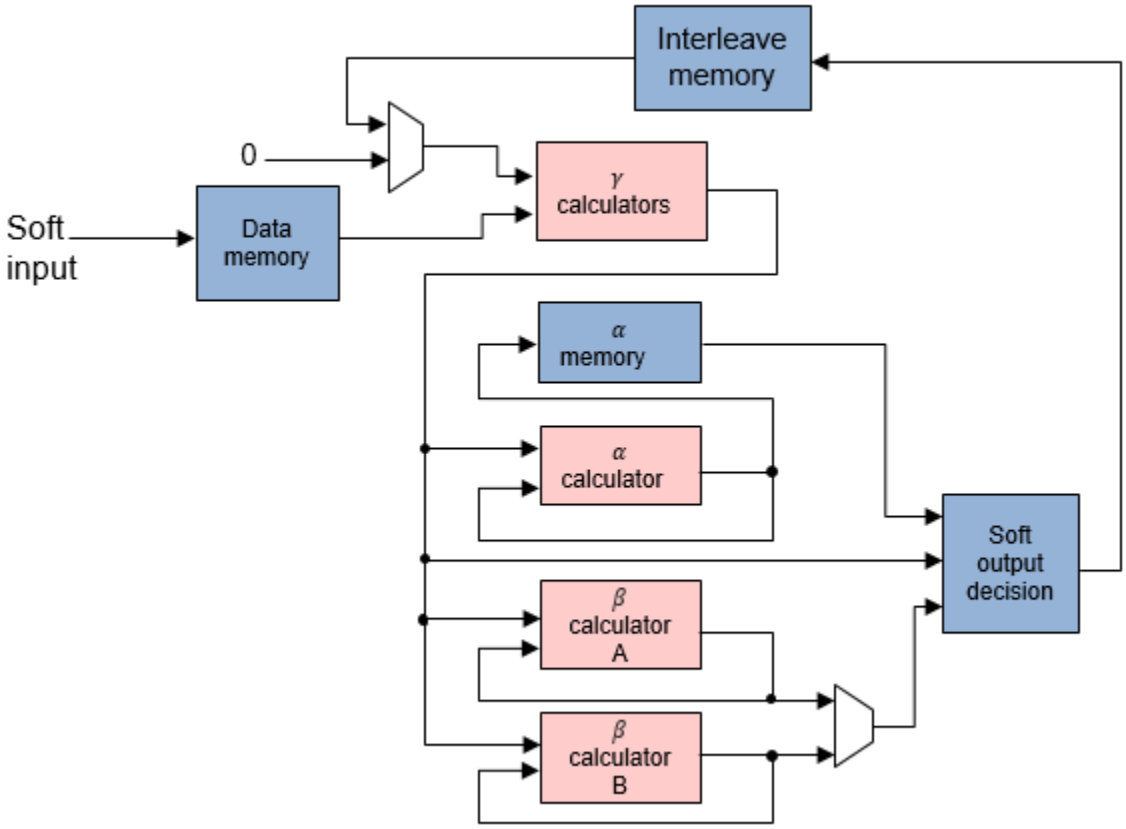
$$\beta_{k-1}(s') = \beta_k(s) \cdot \gamma_k(s', s)$$

The initial conditions for α and β are:

$$\alpha_0(s) = \begin{cases} 1, s = \\ 0, s \neq \end{cases}$$

$$\beta_0(s) = \begin{cases} 1, s = \\ 0, s \neq \end{cases}$$

This diagram shows the half-iteration decoder and interleaver architecture. The initial likelihood is set to zero.

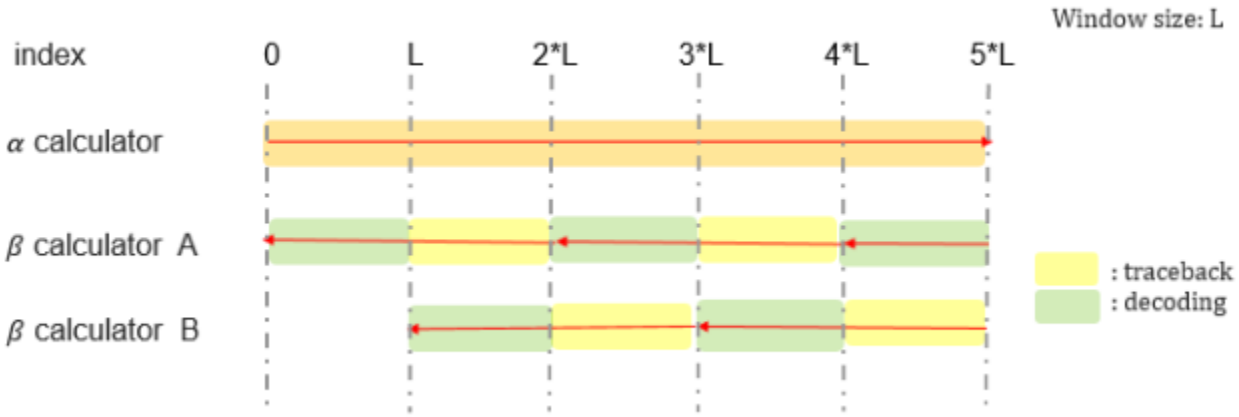


In a normal BCJR architecture, the algorithm cannot compute β until the entire frame is in memory. It must perform a full-frame forward trace and then a full-frame backward trace, which means the latency of one half-iteration is two frame lengths. The required memory is `BlockSize`*`NumStates`*`DataWidth`. In this case, *NumStates* is eight, from LTE standard TS 36.212 [1].

However, this decoder implementation uses a sliding window to reduce the required memory and the latency of the algorithm [3]. The window size is 32 samples, which is five times the trellis constraint length of 7. The latency of one half-iteration is:

$$\left(ceil\left(\frac{BlockSize}{WinSize}\right)+2\right)\times WinSize+PipeDe$$

The required memory with sliding window is 2*32**NumStates***DataWidth*. This figure shows how the β calculation traces and decodes one window at a time, alternating input between the A and B calculation blocks.



Performance

These resource and performance data are the synthesis results from the generated HDL targeted to a Xilinx Zynq-7000 ZC706 board. This implementation is for a fixed block size of 6144, six decode iterations, and `sfix5_en2` input data samples. The design achieves 306.6 MHz clock frequency.

Resource	Uses
LUT	4771
LUTRAM	212
FFS	4691
Block RAM (16K)	7

The RAM size grows with block size. When you enable the block size port, RAM sufficient for the largest block size is used.

References

[1] 3GPP TS 36.212. "Multiplexing and channel coding." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA)*. URL: https://www.3gpp.org.

[2] Bahl, L. R., J. Cocke, F. Jelinek, and J. Raviv. "Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate." *IEEE Transactions on Information Theory*. Vol 1T-20, March 1974, pp. 284–287.

[3] Viterbi, Andrew J. "An Intuitive Justification and a Simplified Implementation of the MAP Decoder for Convolutional Codes." *IEEE Journal on Selected Areas in Communications*. Vol. 16, No. 2, February 1998.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem or an Enabled Synchronous Subsystem.

See Also

Blocks

LTE Turbo Encoder

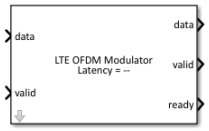
Functions

`lteTurboDecode` | `lteTurboEncode`

Introduced in R2017b

LTE OFDM Modulator

Modulate LTE resource grid and return time-domain OFDM samples
Library: Wireless HDL Toolbox / Modulation



Description

The LTE OFDM Modulator block implements an algorithm for modulating LTE resource grid samples specified by LTE standard TS 36.212 [1]. The block uses an orthogonal frequency-division multiplexing (OFDM) mechanism in its operation and converts the resource grid input samples to an equivalent time-domain signal output. OFDM is effective for communication over channels with high-frequency selectivity and is widely used in the development of the LTE downlink transmitter. The block implements a windowing feature to reduce the spectral regrowth, or adjacent channel leakage ratio (ACLR), of an OFDM signal.

The block provides an interface and architecture suitable for HDL code generation and hardware deployment.

You can select the number of downlink resource blocks (NDLRB) and choose either normal or extended cyclic prefix (CP), as described in the LTE standard. The latency from the first input sample to the first output sample depends on your selection of the NDLRB.

NDLRB	Latency
6	6268
15	6376
25	6496
50	6796
75	7096
100	7396

Ports

Input

`data` — Input data

scalar

Input data, specified as a signed real or complex number. `double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `signed fixed point`

`valid` — Indicates valid input data

scalar

Control signal that indicates if the data from the **data** input port is valid. When this value is `1` (`true`), the block captures the value on the **data** input port. When this value is `0` (`false`), the block ignores the values on the **data** input port.

Data Types: `Boolean`

`NDLRB` — Number of downlink resource blocks

`6` | `15` | `25` | `50` | `75` | `100`

Number of downlink resource blocks, specified as `6`, `15`, `25`, `50`, `75`, or `100`. The NDLRB must be one of these six values specified by LTE standard TS 36.212 [1]. The block samples this port at the start of each subframe and ignores any changes within a subframe.

Dependencies

To enable this port, set the **NDLRB source** parameter to `Input port`.

Data Types: `uint8` | `uint16` | `uint32` | `fixdt(0,K,0), K >= 7` | `single` | `double`

`cyclicPrefixType` — Type of CP

scalar

Type of CP, specified as a `Boolean` scalar. When this value is `0` (false), the block selects normal CP. When this value is `1` (true), the block selects extended CP. The block samples this port at the start of each subframe and ignores any changes within a subframe.

Dependencies

To enable this port, set the **Cyclic prefix source** parameter to `Input port`.

Data Types: `Boolean`

`reset` — Clears internal states

scalar

Clears internal states, specified as a `Boolean` scalar. When this value is `1` (`true`), the block stops the current calculation and clears all internal states. When this value is `0` (`false`) and the **valid** input value is `1` (`true`), the block begins a new subframe.

Dependencies

To enable this port, select the **Enable reset input port** parameter.

Data Types: `Boolean`

Output

`data` — Output data

scalar

Output data, returned as a signed real or complex number. The data type is the same as the data type of the **data** input port. When you clear the **Divide butterfly outputs by two** parameter, the word length increases by 1 bit per stage in inverse fast fourier transform (IFFT).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `signed fixed point`

`valid` — Indicates valid output data

scalar

Control signal that indicates if the data from the **data** output port is valid. The block sets this value to `1` (`true`) when the modulated samples are available on the **data** output port.

Data Types: `Boolean`

`ready` — Indicates block is ready

scalar

Control signal that indicates that the block is ready for new input data. When this value is `1` (`true`), the block accepts input data in the next time step. When this value is `0` (`false`), the block ignores input data in the next time step.

Data Types: `Boolean`

Parameters

Main

`NDLRB source` — Source of NDLRB

`Property` (default) | `Input port`

You can set the NDLRB by selecting a parameter value or using an input port. To enable the **NDLRB** parameter, select `Property`. To enable the **NDLRB** port, select `Input port`.

`NDLRB` — Number of downlink resource blocks

`6` (default) | `15` | `25` | `50` | `75` | `100`

Number of downlink resource blocks, specified as `6`, `15`, `25`, `50`, `75`, or `100`. NDLRB must be one of these six values specified by LTE standard TS 36.212 [1].

Dependencies

To enable this parameter, set the **NDLRB source** parameter to `Property`.

`Cyclic prefix source` — Source of cyclic prefix

`Property` (default) | `Input port`

You can set the cyclic prefix by selecting a parameter value or using an input port. To enable the **Cyclic prefix type** parameter, select `Property`. To enable the **cyclicPrefixType** port, select `Input port`.

`Cyclic prefix type` — Type of cyclic prefix

`Normal` (default) | `Extended`

Type of cyclic prefix, specified as `Normal` or `Extended`.

Dependencies

To enable this parameter, set the **Cyclic prefix source** parameter to `Property`.

`Windowing` — Spectral growth reduction

`off` (default) | `on`

Select this parameter to perform a windowing operation that reduces spectral growth and uses the NDRLB window length specified by the **Window length per NDLRB** parameter. Clear this parameter to disable windowing operation.

`Window length per NDLRB` — NDRLB window length
`[4, 6, 4, 6, 8, 8]` (default) | row vector of NDLRB window lengths

NDRLB window length, specified as a row vector of nonnegative integers whose elements correspond to the window lengths for NDRLB `6`, `15`, `25`, `50`, `75`, and `100` respectively. By default, the window lengths for NDRLB 6, 15, 25, 50, 75, and 100 are 4, 6, 4, 6, 8, and 8, respectively. The window length for each NDLRB can range from 0 to the minimum CP value.

- For normal CP, the minimum CP values for NDRLB `6`, `15`, `25`, `50`, `75`, and `100` are 9, 18, 36, 72, 144, and 144, respectively.
- For extended CP, the minimum CP values for NDRLB `6`, `15`, `25`, `50`, `75`, and `100` are 32, 64, 128, 256, 512, and 512, respectively.

Dependencies

To enable this parameter, select the **Windowing** parameter.

`Enable reset input port` — Reset signal
`off` (default) | `on`

Select this parameter to enable the **reset** port on the block icon.

`Output data sample rate` — Output sample rate
`Use maximum output data sample rate` (default) | `Match output data sample rate to NDLRB`

This parameter specifies the type of sample rate for the block to select for the output data.

- To provide an output data sample rate of 30.72 MHz, select `Use maximum output data sample rate`.
- To provide an output data sample rate based on the **NDLRB** parameter, select `Match output data sample rate to NDLRB`. The output sampling rates for NDLRB `6`, `15`, `25`, `50`, `75`, and `100` are 1.92 MHz, 3.84 MHz, 7.68 MHz, 15.36 MHz, 30.72 MHz, and 30.72 MHz, respectively.

For more information, see “Base Rate Controller” on page 1-76.

IFFT Block Parameters

`Divide butterfly outputs by two` — Bit-width control
`on` (default) | `off`

When you select this parameter, the IFFT HDL Optimized block in the LTE OFDM Modulator block implements an overall 1/*N* scale factor by dividing the output of each butterfly multiplication by two. This adjustment keeps the output of the IFFT block in the same amplitude range as its input. If you disable this parameter, the block avoids overflow by increasing the word length by 1 bit after each butterfly multiplication.

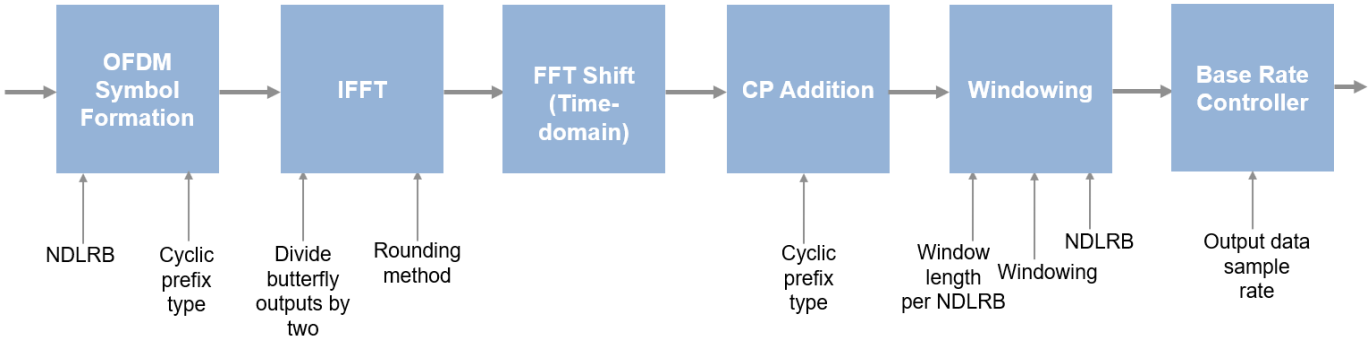
`Rounding Method` — Rounding mode for internal fixed-point calculations
`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Zero`

This parameter specifies the type of rounding mode for internal fixed-point calculations. Rounding applies to twiddle-factor multiplication and scaling operations. For more information about rounding modes, see Rounding Modes (DSP System Toolbox).

When the input is any integer or fixed-point data type, the IFFT algorithm uses fixed-point arithmetic for internal calculations. This parameter does not apply when the input data is of data type `single` or `double`.

Algorithms

The LTE OFDM Modulator block operation sequence is carried over using these blocks: OFDM Symbol Formation, IFFT, FFT Shift, CP Addition, Windowing, and Base Rate Controller. The OFDM Symbol Formation block maps the resource grid input to active subcarrier bins to form 2048 subcarriers. The IFFT block converts the frequency-domain signal to time-domain signal, and the FFT Shift block performs time-domain FFT shift. The CP Addition block adds CP-length samples from the end of the symbol to its prefix. The Windowing block performs windowing and overlapping of adjacent OFDM symbols of complex symbols in the resource array. The Base Rate Controller block defines the sample rate of the output data. The parameters shown in the following figure configure the behavior of the block.



OFDM Symbol Formation

An OFDM Symbol Formation subsystem calculates the number of active and inactive subcarriers and the number of CP samples. It generates a **ready** signal and subcarriers (active, inactive, and DC) for each OFDM symbol as per LTE standard.

The OFDM Symbol Formation subsystem calculates the number of active subcarriers based on the NDLRB value.

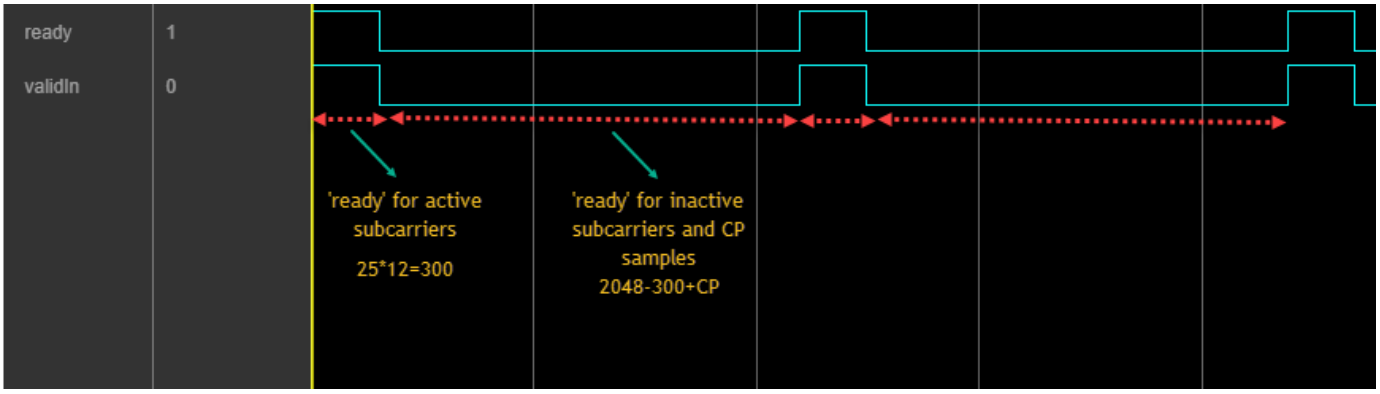
Number of active subcarriers = 12 x **NDLRB**

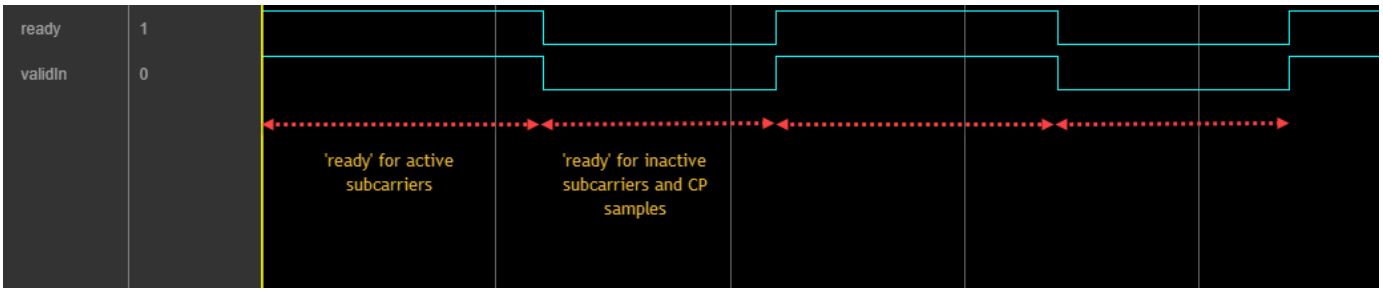
The number of inactive subcarriers is a difference of IFFT size and the number of active subcarriers. To save hardware resources by avoiding multiple IFFTs, the IFFT size is fixed to 2048.

Number of inactive subcarriers = 2048 — Number of active carriers

The block outputs a signal from **ready** port to indicate when the block is ready to accept input data. This signal depends on the **valid** input signal, NDLRB, CP type, and OFDM symbol number. The **ready** output signal is generated for one time step, and the **valid** input signal is checked for next time step. The block accepts the input data and the **ready** output signal remains high (`1`) until the OFDM symbol data is received. If the **valid** input is low (`0`), the **ready** signal extends until the **valid** input signal receives high (`1`). After receiving active subcarriers, the block sets the **ready** output signal to low (`0`) for a time period equal to the sum of the number of inactive subcarriers and the number of CP samples.

These figures show the timing diagrams of the **ready** output signal for NDLRB values 25 and 100, respectively.





IFFT

The IFFT block converts a frequency-domain signal to a time-domain signal. LTE supports six standard bandwidth options: 1.4 MHz, 3 MHz, 5 MHz, 10 MHz, 15 MHz, and 20 MHz. These bandwidth options require an FFT length of 128, 256, 512, 1024, and 2048, respectively. The block uses 2048 FFT length, which corresponds to the maximum bandwidth of LTE, that is 20-MHz. The FFT length of IFFT is configured to the highest FFT size to generate single hardware, which supports all LTE bandwidth options.

The **Divide butterfly outputs by two** parameter controls if the FFT implements an overall 1/*N* scale factor by dividing the output of each butterfly multiplication by two. This adjustment keeps the output of the IFFT in the same amplitude range as its input. When the **Divide butterfly outputs by two** parameter is cleared, the block avoids overflow by increasing the word length by 1 bit after each butterfly multiplication.

FFT Shift

Conventionally, transceivers perform an FFT shift in the frequency domain. However, this method requires memory and introduces latency related to the size of the FFT. Instead, a transceiver can execute the same operation in the time domain using the frequency shifting property of Fourier transforms. Shifting a function in one domain corresponds to a multiplication by a complex exponential function in the other domain. To reduce hardware resources and latency, this block performs the FFT shift by multiplying the time-domain samples by a complex exponential function.

These equations describe an FFT shift. The equation for an *N*-point FFT is

$$X(k) = F[x(n)] = \sum_{n=0}^{N-1} x(n) e^{-\frac{j2\pi nk}{N}}$$

For an FFT shift of $N/2$ carriers in either direction, substitute $k = k - \frac{N}{2}$, resulting in

$$X(k - \frac{N}{2}) = \sum_{n=0}^{N-1} x(n) e^{-\frac{j2\pi n(k - \frac{N}{2})}{N}}$$

This equation simplifies to

$$X(k - \frac{N}{2}) = \sum_{n=0}^{N-1} e^{j\pi n} x(n) e^{-\frac{j2\pi nk}{N}}$$

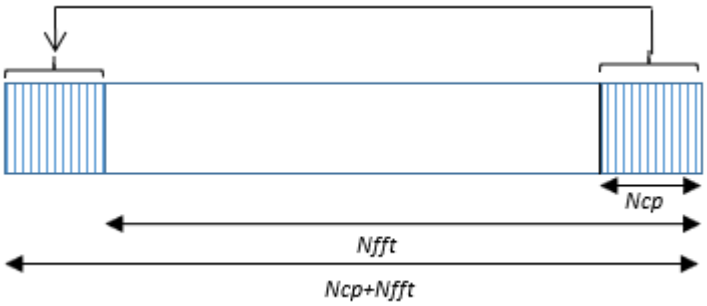
Since $\sum_{n=0}^{N-1} x(n) e^{-\frac{j2\pi nk}{N}}$ is equivalent to $F[x(n)]$, and $e^{j\pi} = -1$, this equation simplifies to

$$X(k - \frac{N}{2}) = F[(-1)^n x(n)]$$

The final equation shows that an FFT shift in the time domain simplifies to multiplication by $(-1)^n$. Therefore, the block implements the FFT shift by multiplying the time-domain samples by either +1 or –1.

CP Addition

Cyclic prefix addition is a process of adding the last samples of an OFDM symbol as a prefix to each OFDM symbol. CP addition for an OFDM symbol with *Nfft* samples and CP samples N_{CP} is shown in this figure.



The LTE OFDM Modulator block uses FFT size of 2048 for all NDLRB resources to avoid multiple IFFTs. The block uses CP values corresponding to NDLRB 100.

The **Cyclic prefix type** parameter controls whether the block expects a normal or an extended CP. The block requires the input to maintain a sample rate of 30.72 MHz. It assumes that each symbol is 2048 samples plus the cyclic prefix size associated with the rate. When using a normal CP, the prefix of the first symbol in each slot has 160 samples, while each subsequent symbol contains a prefix of 144 samples. When using an extended CP, all symbols contain 512 samples. For more information about the cyclic prefix length (in samples) of each OFDM symbol in a subframe, see the `lteOFDMModulate` function.

Windowing

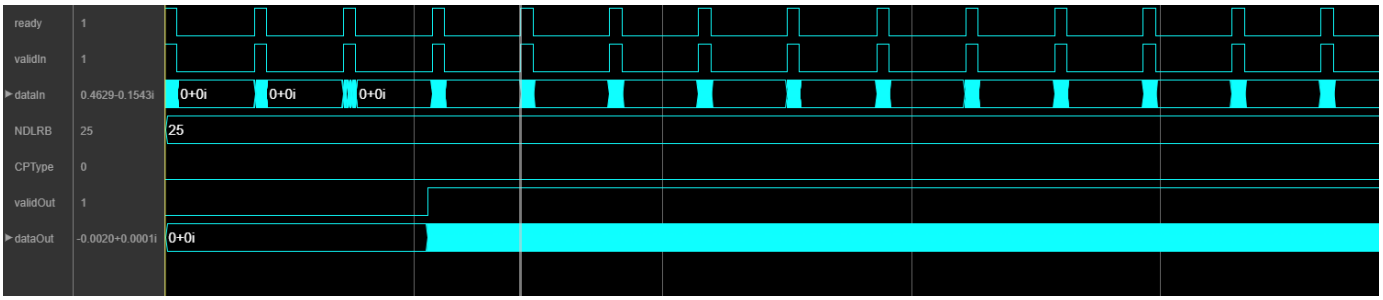
Windowing reduces the spectral regrowth, or adjacent channel leakage ratio (ACLR), of an OFDM signal. The feature is optional. To enable windowing, select the **Windowing** parameter.

For more information about windowing, see the `lteOFDMModulate` function.

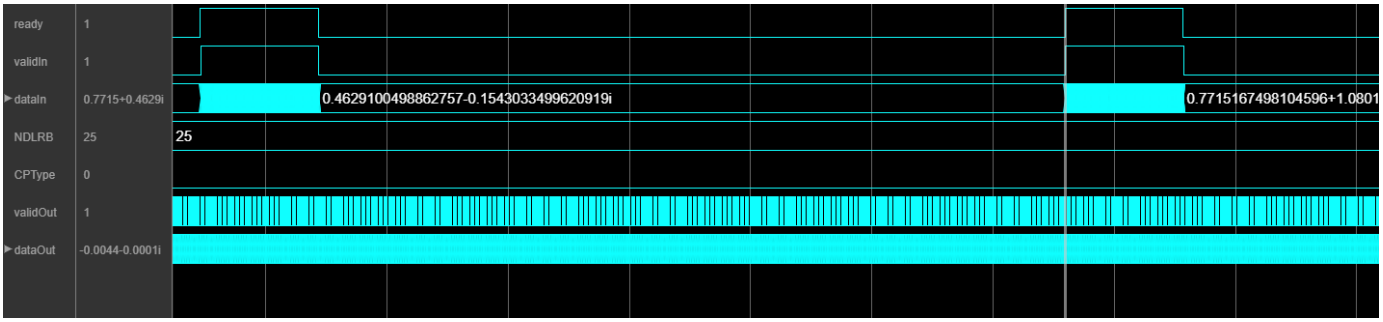
Base Rate Controller

This block generates the output data at a sample rate of 30.72 MHz by using the maximum output data sample rate or output data sample rate with the specified NDLRB.

This figure shows the output data when you set the **Output data sample rate** parameter to `Use maximum output data sample rate`.



This figure shows the output data when you set the **Output data sample rate** parameter to `Match output data sample rate to NDLRB` parameter. For NDLRB value 25, the output sample rate is 7.68 MHz, and the block returns valid output data at every fourth cycle.



Performance

The performance of the synthesized HDL code varies with your target and synthesis options. The input data type used for generating HDL code is `fixdt(1,16,14)`.

This table shows the resource and performance data synthesis results when using the block with default configuration. The generated HDL targeted to Xilinx Zynq XC7Z045I-FFG900-2L FPGA. The design achieves a clock frequency of 247 MHz.

Resource	Number Used
LUTs	8050
Registers	9682

Resource	Number Used
DSPs	22
Block RAM	22
F7 Muxes	0
F8 Muxes	0
RAMB36/FIFO	12
RAMB18	20

References

[1] 3GPP TS 36.212. "Multiplexing and channel coding." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA)*. URL: https://www.3gpp.org.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

See Also

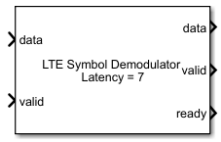
Blocks
LTE OFDM Demodulator

Functions
`lteOFDMDemodulate` | `lteOFDMModulate`

Introduced in R2019a

LTE Symbol Demodulator

Demodulate complex LTE data symbols to data bits or LLR values
Library: Wireless HDL Toolbox / Modulation



Description

The LTE Symbol Demodulator block demodulates complex data symbols to data bits or log likelihood ratios (LLR) values based on the modulation types supported by LTE standard TS 36.211 [1]. The block provides an architecture suitable for HDL code generation and hardware deployment. You can use this block in the development of an LTE receiver.

The block accepts data symbols, along with a valid signal, and outputs demodulated bits or LLR values with valid and ready signals. The number of demodulated bits or LLR values for a given symbol depends on the modulation type, as shown in this table.

Modulation Type	Number of Bits per Symbol (NBPS)
BPSK	1
QPSK	2
16-QAM	4
64-QAM	6
256-QAM	8

The **ready** output port indicates when the block can accept an input data sample. You can use **ready** output port to control the upstream data coming to the block.

Ports

Input

`data` — Input data symbols
complex scalar

Input data symbols, specified as a complex scalar. The block performs demodulation assuming the input constellation power normalization is in accordance with LTE standard TS 36.211, Section 7.1 [1]. The normalization values are based on the modulation type.

- 1/√2 for BPSK and QPSK
- 1/√10 for 16-QAM
- 1/√42 for 64-QAM
- 1/√170 for 256-QAM

Example: For BPSK modulation, the input values can be `[0.707 +0.707i; -0.707 -0.707i]`

`double` and `single` data types are supported for simulation, but not for HDL code generation.

For HDL code generation, the input data type must be `signed fixed point` and the maximum input word length the block supports is 32 bits.

Data Types: `single` | `double` | `signed fixed point`

`valid` — Valid input data indication

scalar

Control signal that indicates if the input data is valid. When this value is `1` (true), the block accepts the values on the **data** input port. When this value is `0` (false), the block ignores the values on the **data** input port.

Data Types: `Boolean`

`modSel` — Modulation selection

integer from 0 to 4

Select the modulation type by specifying its corresponding value shown in this table. Valid **modSel** values are from 0 to 4. Each value represents a specific modulation type, as shown in this table.

Value	Modulation Type
`0`	BPSK
`1`	QPSK
`2`	16-QAM
`3`	64-QAM
`4`	256-QAM

If you specify a value other than one listed in this table, the block displays a warning message and applies QPSK modulation.

For HDL code generation, specify this value in `fixdt(0,3,0)` format. `double` and `single` data types are supported for simulation but not for HDL code generation.

Dependencies

To enable this port, set the **Modulation source** parameter to `Input port`.

Data Types: `single` | `double` | `signed fixed point`

Output

`data` — Output demodulated data bits or LLR values

scalar

Output demodulated data bits or LLR values, returned as a scalar.

- When you set the **Decision type** parameter to `Soft`, the block outputs demodulated LLR values. A positive LLR output value is considered as 1 and a negative LLR output value is considered as 0. The magnitude of the output gives a piecewise linear approximation to the LLR of the demodulated bits. The algorithm used for the LLR approximation is described in [1]. The returned LLR values for the input signal located on these constellation points lie within these magnitudes.

- 1 for BPSK
- 1/√2 for QPSK
- [1 3]/√10 for 16-QAM
- [1 3 5 7]/√42 for 64-QAM
- [1 3 5 7 9 11 13 15]/√170 for 256-QAM

The output word length increases by 2 bits for inputs with data type `signed fixed point`. For input with data types `double` or `single`, the output data type is the same as the input data type.

- When you set the **Decision type** parameter to `Hard`, the block results in the output containing the bit sequences corresponding to the closest constellation points to the input. The data type of this output is `Boolean`.

Data Types: `single` | `double` | `signed fixed point` | `Boolean`

`valid` — Valid output data indication

scalar

Control signal that indicates if data from the **data** output port is valid. When this value is `1` (true), the block returns valid data on the **data** output port. When this value is `0` (false), the values on the **data** output port are not valid.

Data Types: `Boolean`

`ready` — Block ready indicator

scalar

Control signal that indicates when the block is ready to accept new input data. When this value is `1` (true), the block accepts input data in the next time step. When this value is `0` (false), the block ignores the input data in the next time step.

The **ready** signal remains `0` (false) until the block outputs data of the corresponding input data symbol. The number of clock cycles the **ready** signal remains `0` (false) depends on the selected modulation type. If the selected modulation type is 16-QAM, the **ready** signal remains `0` (false) for 3 clock cycles, calculated as *NBPS* – 1 and then it changes to `1` (true) indicating that the block is ready to accept data in the next time step.

Data Types: `Boolean`

Parameters

`Modulation source` — Source for modulation type

`Property` (default) | `Input port`

To specify the modulation type by using the **Modulation** parameter, select `Property`. To specify the modulation type from the **modSel** port during run time, select `Input port`.

`Modulation` — Modulation type

`BPSK` (default) | `QPSK` | `16-QAM` | `64-QAM` | `256-QAM`

Select the modulation type.

Dependencies

To enable this parameter, set the **Modulation source** parameter to `Property`.

`Decision type` — Type of demapping

`Soft` (default) | `Hard`

Select the demapping type.

- `Soft` — Demap data symbols to LLR values. This LLR value for each bit indicates how likely the bit is `1` or `0`.
- `Hard` — Demap data symbols to bits `1` or `0`.

`Rounding Method` — Rounding mode for internal fixed-point calculations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Zero`

This parameter specifies the type of rounding mode for internal fixed-point calculations. For more information about rounding modes, see Rounding Modes (DSP System Toolbox). This parameter does not apply when the input is of data type `double` or `single`.

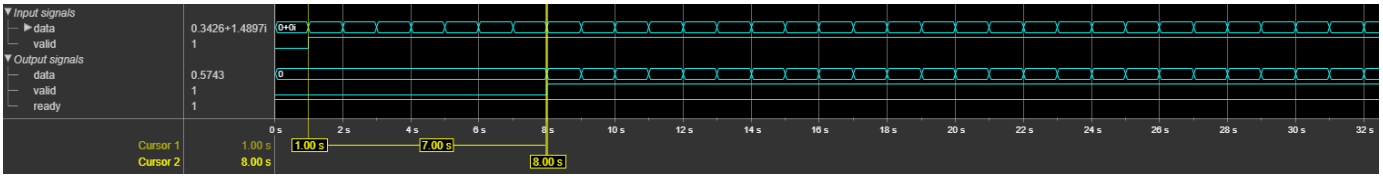
Algorithms

The block outputs data in the form of bits or LLR values based on the demapping type you specify for the **Decision type** parameter: `Hard` or `Soft` respectively. For this demapping, the block implements simplified approximate LLR algorithm [2].

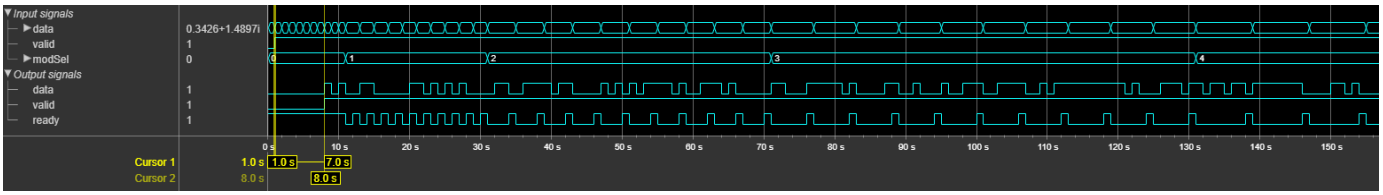
Latency

The block captures output bits at valid cycles. The latency of the block is seven clock cycles.

This figure shows a sample output and latency of the block when you set the **Modulation** parameter to BPSK and the **Decision type** parameter to `Soft`.



This figure shows a sample output and latency of the block when you specify the **modSel** values as `0` (BPSK), `1` (QPSK), `2` (16-QAM), `3` (64-QAM), and `4` (256-QAM) and set the **Decision type** parameter to `Hard`.



Performance

This table shows the resource and performance data synthesis results of the block for soft-decision and hard-decision demapping types when you provide an input data type of `fixdt(1,16,14)`. The modulation type is applied using the **modSel** input port. The generated HDL is targeted to a Xilinx Zynq- 7000 ZC706 evaluation board.

The design achieves a clock frequency of 475.9 MHz for soft-decision and 454.7 MHz for hard-decision.

Resource Utilization	Soft Decision	Hard Decision
Slice LUTs	234	188
Slice Registers	276	225
DSP	1	0

References

[1] 3GPP TS 36.211. "Physical channels and modulation." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA)*. URL: https://www.3gpp.org.

[2] F. Tosato and P. Bisaglia. "Simplified soft-output de-mapper for binary interleaved coded OFDM with application to HIPERLAN/2." ICC 2002, Vol. 2, pp. 664–668.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

This block does not have any HDL Block Properties.

See Also

Blocks

LTE Symbol Modulator | NR Symbol Demodulator

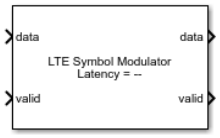
Functions

`lteSymbolDemodulate`

Introduced in R2019b

LTE Symbol Modulator

Modulate data bits to complex LTE data symbols
Library: Wireless HDL Toolbox / Modulation



Description

The LTE Symbol Modulator block maps a group of data bits to complex data symbols by using the modulation types supported by LTE standard TS 36.211 [1]. The block provides an architecture suitable for HDL code generation and hardware deployment. You can use this block in the development of an LTE transmitter.

The block accepts 1-bit of data at a time, along with control signals, and outputs a modulated complex symbol with a valid signal. Each complex symbol comprises a standard number of bits based on the modulation type, as shown in this table. If you provide a nonmultiple of modulation-order bits as an input, the block ignores the output symbol with insufficient or excessive bits. The modulation order is the number of bits per symbol.

Modulation Type	Modulation Order - Number of Bits Per Symbol
BPSK	1
QPSK	2
16-QAM	4
64-QAM	6
256-QAM	8

Ports

Input

`data` — Input data bits
scalar

Input data bits, specified as a scalar. The block accepts `Boolean` or `ufix1` data bits.

Data Types: `Boolean` | `fixdt(0,1,0)`

`valid` — Indicate valid input data
scalar

Control signal that indicates if the input data is valid. When this value is `1` (true), the block accepts the values on the **data** input port. When this value is `0` (false), the block ignores the values on the **data** input port.

Data Types: `Boolean`

`modSel` — Modulation selection

integer

Select the modulation type by specifying its corresponding value shown in this table.

The **modSel** values are from 0 to 4. Each value represents a specific modulation type.

Value	Modulation Type
`0`	BPSK
`1`	QPSK
`2`	16-QAM
`3`	64-QAM
`4`	256-QAM

If you specify a value other than one listed in this table, the block displays a warning message and:

- Applies QPSK modulation when **load** is `1` (true).
- Continues with the current modulation when **load** is `0` (false).

For HDL code generation, specify this value in `fixdt(0,3,0)` format.

Dependencies

To enable this port, set the **Modulation source** parameter to `Input port`.

Data Types: `fixdt(0,3,0)` | `double` | `single`

`load` — Modulation control

scalar

Control signal to sample modulation.

When this value is `1` (true), the block applies the modulation based on the **modSel** value. When this value is `0` (false), the block ignores any changes in the **modSel** value and continues with the current modulation until **load** changes to `1` (true).

If the **load** value changes to `1` (true) during the block operation, the block resynchronizes and restarts modulation using the current value of **modSel**. This restart occurs whether or not the **modSel** value has changed. For example, if the block is operating in 256-QAM mode and the **load** value changes to `1` (true) after four of the eight required input bits are sent into the block, the block discards those first four bits and restarts its operation from the fifth bit.

If you do not apply the **load** value as `1` (true) at the start of block operation, by default, the block operates with QPSK modulation.

Dependencies

To enable this port, set the **Modulation source** parameter to `Input port`.

Data Types: `Boolean`

Output

`data` — Modulated complex data symbols

scalar

Modulated complex data symbols, returned as a scalar.

Data Types: `single` | `double` | `fixdt(1,wordlength,wordlength-2)`

`valid` — Valid output data indication

scalar

Control signal that indicates if data from the **data** output port is valid. When this value is `1` (true), the block returns valid data on the **data** output port. When this value is `0` (false), the block ignores values on the **data** output port.

Data Types: `Boolean`

Parameters

Main

`Modulation source` — Source for modulation type

`Property` (default) | `Input port`

To specify the modulation type by using the **Modulation** parameter, select `Property`. To specify the modulation type from the **modSel** port during run time, select `Input port`.

`Modulation` — Modulation type

`BPSK` (default) | `QPSK` | `16-QAM` | `64-QAM` | `256-QAM`

Select the modulation type.

Dependencies

To enable this parameter, set the **Modulation source** parameter to `Property`.

Data Types

`Output data type` — Output data type selection

`double` (default) | `single` | `Custom`

Specify the data type for the output data.

`double` and `single` data types are supported for simulation.

For simulation and HDL code generation, set this value to `Custom`.

`Word length` — Output word length

`16` (default) | integer in the range [3, 32]

Specify the output word length. This value must be an integer in the range [3, 32].

Dependencies

To enable this parameter, set the **Output data type** parameter to `Custom`.

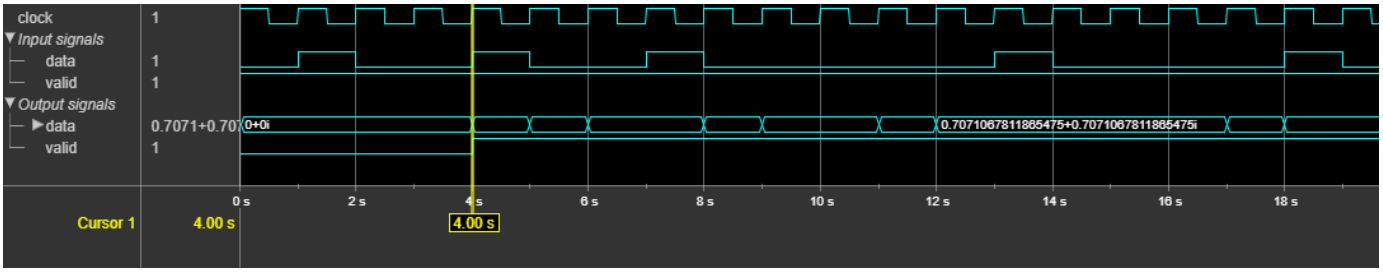
Algorithms

Output Waveforms

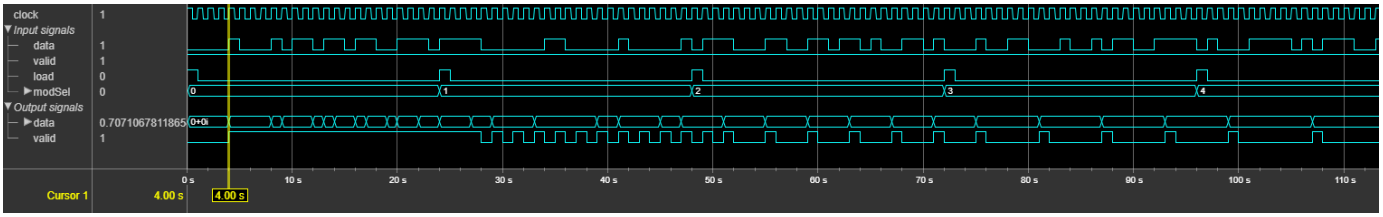
These figures show sample outputs of the block when operated with a fixed modulation type applied using the **Modulation** parameter and when operated with varied modulation types applied using the **modSel** input port. The latency of the block is equal to the sum of three clock cycles and the number of bits per symbol.

The block captures the output symbols at valid cycles.

This figure shows a sample output when you select `BPSK` modulation by using the **Modulation** parameter. In this sample output, the latency of the block is 4.



This figure shows a sample output when you select `BPSK`, `QPSK`, `16-QAM`, `64-QAM`, and `256-QAM` modulations by using the **modSel** input port. The latency of the block varies with the modulation selection.



Performance

This table shows the resource and performance data synthesis results of the block when using the **modSel** input port as the modulation source and an output word length of 16. The generated HDL is targeted to a Xilinx Zynq- 7000 ZC706 evaluation board. The design achieves a clock frequency of 872 MHz.

Resource	Number Used
Slice LUTs	61
Slice Registers	67
DSP48	0

References

[1] 3GPP TS 36.211. "Physical channels and modulation." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA)*. URL: https://www.3gpp.org.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

This block does not have any HDL Block Properties.

See Also

Blocks

LTE Symbol Demodulator | NR Symbol Modulator

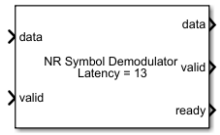
Functions

`lteSymbolModulate`

Introduced in R2019a

NR Symbol Demodulator

Demodulate complex NR data symbols to data bits or LLR values
Library: Wireless HDL Toolbox / Modulation



Description

The NR Symbol Demodulator block demodulates complex data symbols to data bits or log likelihood ratios (LLR) values based on the modulation types supported by 5G New Radio (NR) standard TS 38.211 [1]. The block provides an architecture suitable for HDL code generation and hardware deployment. You can use this block in the development of an NR receiver.

The block accepts data symbols, along with a valid signal, and outputs demodulated bits or LLR values with valid and ready signals. The number of demodulated bits or LLR values for a given symbol depends on the modulation type, as shown in this table.

Modulation Type	Number of Bits per Symbol (NBPS)
BPSK	1
QPSK	2
16-QAM	4
64-QAM	6
256-QAM	8
pi/2-BPSK	1

The **ready** output port indicates when the block can accept an input data sample. You can use **ready** output port to control the upstream data coming to the block.

Ports

Input

`data` — Input data symbols
complex scalar

Input data symbols, specified as a complex scalar. The block performs demodulation assuming the input constellation power normalization is in accordance with NR standard TS 38.211, Section 5.1 [1]. The normalization values are based on the modulation type.

- 1/√2 for BPSK, QPSK, and pi/2-BPSK
- 1/√10 for 16-QAM
- 1/√42 for 64-QAM
- 1/√170 for 256-QAM

Example: For BPSK modulation, the input values can be `[0.707 +0.707i; -0.707 -0.707i]`

`double` and `single` data types are supported for simulation, but not for HDL code generation.

For HDL code generation, the input data type must be `signed fixed point` and the maximum input word length the block supports is 32 bits.

Data Types: `single` | `double` | `signed fixed point`

`valid` — Indicate valid input data

scalar

Control signal that indicates if the input data is valid. When this value is `1` (true), the block accepts the values on the **data** input port. When this value is `0` (false), the block ignores the values on the **data** input port.

Data Types: `Boolean`

`modSel` — Modulation selection

integer from 0 to 5

Select the modulation type by specifying its corresponding value shown in this table. Valid **modSel** values are from 0 to 5. Each value represents a specific modulation type, as shown in this table.

Value	Modulation Type
`0`	BPSK
`1`	QPSK
`2`	16-QAM
`3`	64-QAM
`4`	256-QAM
`5`	pi/2-BPSK

If you specify a value other than one listed in this table, the block displays a warning message and applies QPSK modulation.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

For HDL code generation, specify this value in `fixdt(0,3,0)` format.

Dependencies

To enable this port, set the **Modulation source** parameter to `Input port`.

Data Types: `single` | `double` | `signed fixed point`

Output

`data` — Demodulated data bits or LLR values

scalar

Output demodulated data bits or LLR values, returned as a scalar.

- When you set the **Decision type** parameter to `Soft`, the block outputs demodulated LLR values. A positive LLR output value is considered as 0 and a negative LLR output value is considered as 1.

The magnitude of the output gives a piecewise linear approximation to the LLR of the demodulated bits. The algorithm used for the LLR approximation is described in [1]. The block scales the returned LLRs with a respective scaling factor based on the modulation type as shown in this table.

Modulation Type	Scaling Factor
BPSK	4/√2
QPSK	4/√2
16-QAM	[4 8]/√10
64-QAM	[4 8 12 16]/√42
256-QAM	[4 8 12 16 20 24 28 32]/√170
pi/2-BPSK	4/√2

The output word length increases by 3 bits for inputs with data type `signed fixed point`. For input with data types `double` or `single`, the output data type is the same as the input data type.

- When you set the **Decision type** parameter to `Hard`, the block results in the output containing the bit sequences corresponding to the closest constellation points to the input. The data type of this output is `Boolean`.

Data Types: `single` | `double` | `signed fixed point` | `Boolean`

`valid` — Valid output data indication

scalar

Control signal that indicates if data from the **data** output port is valid. When this value is `1` (true), the block returns valid data on the **data** output port. When this value is `0` (false), the values on the **data** output port are not valid.

Data Types: `Boolean`

`ready` — Indicates block is ready

scalar

Control signal that indicates when the block is ready to accept new input data. When this value is `1` (true), the block accepts input data in the next time step. When this value is `0` (false), the block ignores the input data in the next time step.

The **ready** signal remains `0` (false) until the block outputs data of the corresponding input data symbol. The number of clock cycles the **ready** signal remains `0` (false) depends on the selected modulation type. If the selected modulation type is 16-QAM, the **ready** signal remains `0` (false) for 3 clock cycles, calculated as *NBPS* – 1 and then it changes to `1` (true) indicating that the block is ready to accept data in the next time step.

Data Types: `Boolean`

Parameters

`Modulation source` — Source for modulation type

`Property` (default) | `Input port`

To specify the modulation type by using the **Modulation** parameter, select `Property`. To specify the modulation type from the **modSel** port during run time, select `Input port`.

`Modulation` — Modulation type

`BPSK` (default) | `QPSK` | `16-QAM` | `64-QAM` | `256-QAM` | `pi/2-BPSK`

Select the modulation type.

Dependencies

To enable this parameter, set the **Modulation source** parameter to `Property`.

`Decision type` — Type of demapping

`Soft` (default) | `Hard`

Select the demapping type.

- `Soft` — Demap data symbols to LLR values. This LLR value for each bit indicates how likely the bit is `1` or `0`.
- `Hard` — Demap data symbols to bits `1` or `0`.

`Rounding Method` — Rounding mode for internal fixed-point calculations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Zero`

This parameter specifies the type of rounding mode for internal fixed-point calculations. For more information about rounding modes, see Rounding Modes (DSP System Toolbox). This parameter does not apply when the input is of data type `double` or `single`.

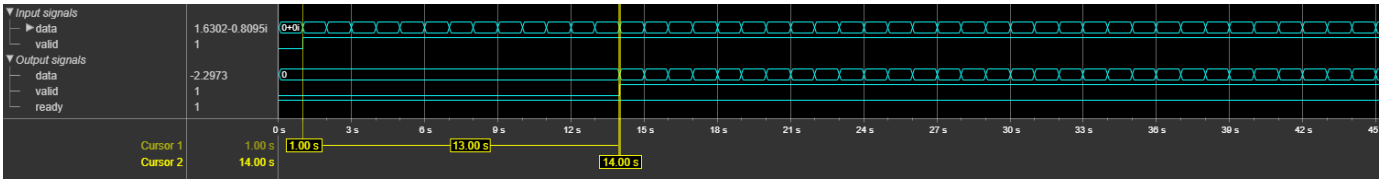
Algorithms

The block outputs data in the form of bits or LLR values based on the demapping type you specify for the **Decision type** parameter: `Hard` or `Soft` respectively. For this demapping, the block implements simplified approximate LLR algorithm [2].

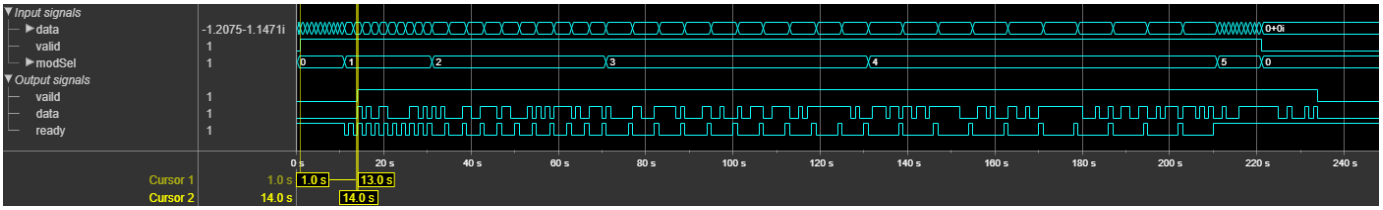
Latency

The block captures output bits at valid cycles. The latency of the block is 13 clock cycles.

This figure shows a sample output and latency of the block when you set the **Modulation** parameter to BPSK and the **Decision type** parameter to `Soft`.



This figure shows a sample output and latency of the block when you specify the **modSel** values as `0` (BPSK), `1` (QPSK), `2` (16-QAM), `3` (64-QAM), `4` (256-QAM), and `5` (pi/2-BPSK) and set the **Decision type** parameter to `Hard`.



Performance

This table shows the resource and performance data synthesis results of the block for soft-decision and hard-decision demapping types when you provide an input data type of `fixdt(1,16,14)`. The modulation type is applied using the **modSel** input port. The generated HDL is targeted to a Xilinx Zynq- 7000 ZC706 evaluation board.

The design achieves a clock frequency of 443.26 MHz for soft-decision and 420.34 MHz for hard-decision.

Resource Utilization	Soft Decision	Hard Decision
Slice LUTs	436	422
Slice Registers	336	337
DSP	1	0

References

[1] 3GPP TS 38.211. "NR; Physical channels and modulation." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*. URL: https://www.3gpp.org.

[2] F. Tosato and P. Bisaglia. "Simplified soft-output de-mapper for binary interleaved coded OFDM with application to HIPERLAN/2." ICC 2002, Vol. 2, pp. 664–668.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

This block does not have any HDL Block Properties.

See Also

Blocks

LTE Symbol Demodulator | NR Symbol Modulator

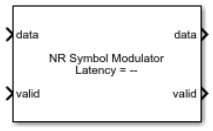
Functions

`nrSymbolDemodulate`

Introduced in R2019b

NR Symbol Modulator

Modulate data bits to complex NR data symbols
Library: Wireless HDL Toolbox / Modulation



Description

The NR Symbol Modulator block maps a group of data bits to complex data symbols by using the modulation types supported by NR standard TS 38.211 [1]. The block provides an architecture suitable for HDL code generation and hardware deployment.

The block accepts 1-bit of data at a time, along with control signals, and outputs a modulated complex symbol with a valid signal. Each complex symbol comprises a standard number of bits based on the modulation type, as shown in this following table. If you provide a nonmultiple of modulation-order bits as an input, the block ignores the output symbol with insufficient or excessive bits. The modulation order is the number of bits per symbol.

Modulation Type	Modulation Order - Number of Bits per Symbol
BPSK	1
QPSK	2
16-QAM	4
64-QAM	6
256-QAM	8
pi/2-BPSK	1

Ports

Input

`data` — Input data bits
scalar

Input data bits, specified as a scalar. The block accepts `Boolean` or `ufix1` data bits.

Data Types: `Boolean` | `fixdt(0,1,0)`

`valid` — Indicate valid input data
scalar

Control signal that indicates if the input data is valid. When this value is `1` (true), the block accepts the values on the **data** input port. When this value is `0` (false), the block ignores the values on the **data** input port.

Data Types: `Boolean`

`modSel` — Modulation selection

integer

Select the modulation type by specifying its corresponding value shown in this table.

The **modSel** values are from 0 to 5. Each value represents a specific modulation type.

Value	Modulation Type
`0`	BPSK
`1`	QPSK
`2`	16-QAM
`3`	64-QAM
`4`	256-QAM
`5`	pi/2-BPSK

If you specify a value other than one listed in this table, the block displays a warning message and:

- Applies QPSK modulation when **load** is `1` (true).
- Continues with the current modulation when **load** is `0` (false).

For HDL code generation, specify this value in `fixdt(0,3,0)` format.

Dependencies

To enable this port, set the **Modulation source** parameter to `Input port`.

Data Types: `fixdt(0,3,0)` | `double` | `single`

`load` — Modulation control

scalar

Control signal to sample modulation.

When this value is `1` (true), the block applies the modulation based on the **modSel** value. When this value is `0` (false), the block ignores any changes in the **modSel** value and continues with the current modulation until **load** changes to `1` (true).

If the **load** value changes to `1` (true) during the block operation, the block resynchronizes and restarts modulation using the current value of **modSel**. This restart occurs whether or not the **modSel** value has changed. For example, if the block is operating in 256-QAM mode and the **load** value changes to `1` (true) after four of the eight required input bits are sent into the block, the block discards those first four bits and restarts its operation from the fifth bit.

If you do not apply the **load** value as `1` (true) at the start of block operation, by default, the block operates with QPSK modulation.

Dependencies

To enable this port, set the **Modulation source** parameter to `Input port`.

Data Types: `Boolean`

Output

data — Modulated complex data symbols
scalar

Modulated complex data symbols, returned as a scalar.

Data Types: `single` | `double` | `fixdt(1,wordlength,wordlength-2)`

valid — Valid output data indication
scalar

Control signal that indicates if data from the **data** output port is valid. When this value is `1` (true), the block returns valid data on the **data** output port. When this value is `0` (false), the block ignores values on the **data** output port.

Data Types: `Boolean`

Parameters

Main

Modulation source — Source for modulation type
`Property` (default) | `Input port`

To specify the modulation type by using the **Modulation** parameter, select `Property`. To specify the modulation type from the **modSel** port during run time, select `Input port`.

Modulation — Modulation type
`BPSK` (default) | `QPSK` | `16-QAM` | `64-QAM` | `256-QAM` | `pi/2-BPSK`

Select the modulation type.

Dependencies

To enable this parameter, set the **Modulation source** parameter to `Property`.

Data Types

Output data type — Output data type selection
`double` (default) | `single` | `Custom`

Specify the data type for the output data.

`double` and `single` data types are supported for simulation.

For simulation and HDL code generation, set this value to `Custom`.

Word length — Output word length
`16` (default) | integer in the range [3, 32]

Specify the output word length. This value must be an integer in the range [3, 32].

Dependencies

To enable this parameter, set the **Output data type** parameter to `Custom`.

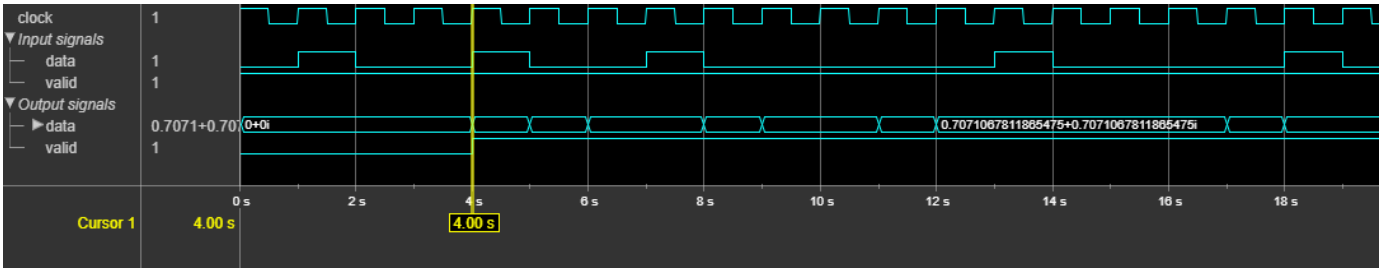
Algorithms

Latency

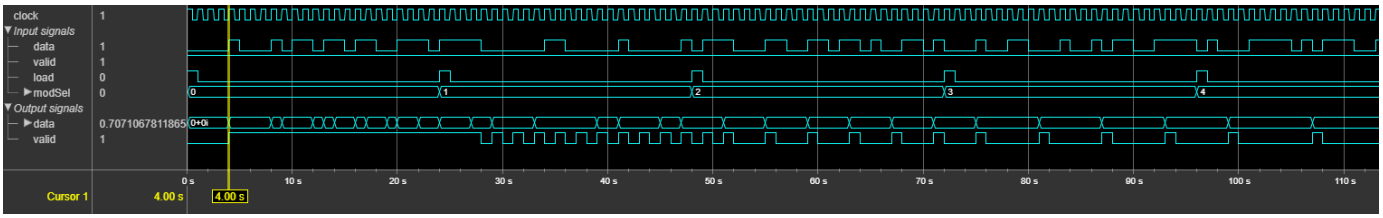
These figures show sample outputs of the block when operated with a fixed modulation type applied using the **Modulation** parameter and when operated with varied modulation types applied using the **modSel** input port. The latency of the block is equal to the sum of three clock cycles and the number of bits per symbol.

The block captures the output symbols at valid cycles.

This figure shows a sample output when you select `BPSK` modulation by using the **Modulation** parameter. In this sample output, the latency of the block is 4.



This figure shows a sample output when you select `BPSK`, `QPSK`, `16-QAM`, `64-QAM`, `256-QAM`, and `pi/2-BPSK` modulations by using the **modSel** input port. The latency of the block varies with the modulation selection.



Performance

This table shows the resource and performance data synthesis results of the block when using the **modSel** input port as the modulation source and an output word length of 16. The generated HDL is targeted to a Xilinx Zynq- 7000 ZC706 evaluation board. The design achieves a clock frequency of 872 MHz.

Resource	Number Used
Slice LUTs	61
Slice Registers	67
DSP48	0

References

[1] 3GPP TS 38.211. "NR; Physical channels and modulation." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*. URL: https://www.3gpp.org.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

This block does not have any HDL Block Properties.

See Also

Blocks

LTE Symbol Modulator | NR Symbol Demodulator

Functions

`nrSymbolModulate`

Introduced in R2019a

NR Polar Encoder

Perform polar encoding according to 5G NR standard
Library: Wireless HDL Toolbox / Error Detection and Correction

Description

The NR Polar Encoder block implements a streaming polar encoder with hardware-friendly control signals. You can configure the block to use downlink or uplink encoding schemes as defined by the 5G NR standard. The 5G NR standard requires polar encoding for channel coding of the DCI, UCI, and BCH transmit channels.

The encoder implementation matches the `nrPolarEncode` function.

Because the latency of this operation can vary, the block provides an output signal, **nextFrame**, that indicates when the block is ready to accept new inputs. For more details, see the “Latency” on page 1-102 section on this page.

Ports

Input

`data` — Input data bit

scalar

Input data bit, specified as a scalar.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `fixdt(0,1,0)` | `Boolean` | `double` | `single`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, specified as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the input frame
- `end` — Indicates the end of the input frame
- `valid` — Indicates that the data on the input **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

K — Length of information block in bits

positive integer

Length of information block in bits, specified as a positive integer. For downlink messages, **K** must be in the range 36 to 164. For uplink messages, **K** must be in the range 31 to 1023.

The block does not support **K** values from 18 to 25 because the 5G NR standard requires parity-aided codes for those sizes.

Dependencies

To enable this port, set the **Configuration source** parameter to `Input port`.

Data Types: `fixdt(0,10,0)`

E — Rate-matched output length in bits

positive integer

Rate-matched output length in bits, specified as a positive integer. Specify a value for **E** that is greater than **K** and less than or equal to 8192.

Dependencies

To enable this port, set the **Configuration source** parameter to `Input port`.

Data Types: `fixdt(0,14,0)`

Output

data — Encoded data bit

scalar

Encoded data bit, returned as a scalar. The block returns a message of *N* sequential bits. *N* is a power of two determined from the values of **K** and **E**. The maximum output message size is 512 bits when the **Link direction** is `Downlink` and 1024 bits when the **Link direction** is `Uplink`.

Data Types: `fixdt(0,1,0)` | `Boolean` | `double` | `single`

ctrl — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, returned as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the output frame
- `end` — Indicates the end of the output frame
- `valid` — Indicates that the data on the output **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

nextFrame — Ready for new inputs

scalar

The block sets this signal to `1` when the block is ready to accept the start of the next frame. 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.

For more information, see “Using the nextFrame Output Signal”.

Data Types: `Boolean`

Parameters

`Link direction` — Direction of 5G NR link

`Downlink` (default) | `Uplink`

When you select `Downlink`, the block performs interleaving, as specified in the 5G NR standard. When you select `Uplink`, the block omits the interleaving logic.

`Configuration source` — Source for K and E

`Property` (default) | `Input port`

Select `Input port` to enable the **K** and **E** ports. Select `Property` to use the **Message length (K)** and **Rate-matched length (E)** parameters.

`Message length (K)` — Length of information block in bits

`56` (default) | positive integer

For downlink messages, **K** must be in the range 36 to 164. For uplink messages, **K** must be in the range 31 to 1023.

The block does not support **K** values from `18` to `25` because the 5G NR standard requires parity-aided codes for those sizes.

Dependencies

To enable this parameter, set the **Configuration source** parameter to `Property`.

`Rate-matched length (E)` — Rate-matched output length in bits

`864` (default) | positive integer

Specify a value for **E** that is greater than **K** and less than or equal to 8192.

Dependencies

To enable this parameter, set the **Configuration source** parameter to `Property`.

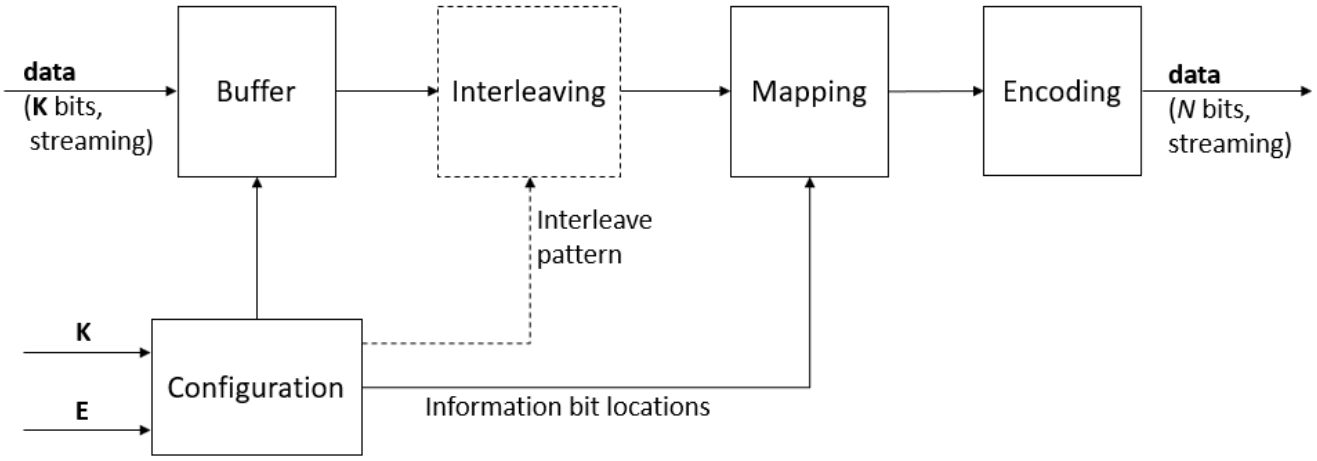
Tips

- You cannot use this block inside an Enabled Subsystem or Resettable Subsystem.

Algorithms

This block implements the encoder by using $\log_2(N)$ parallel encoding stages. The block stores the whole message in the buffer, then interleaves and maps the information bits based on the pattern specified in the standard for the values of **K** and **E**. The interleaving step is included only when you set the **Link direction** parameter to `Downlink`.

This diagram shows the architecture of the polar encoder.



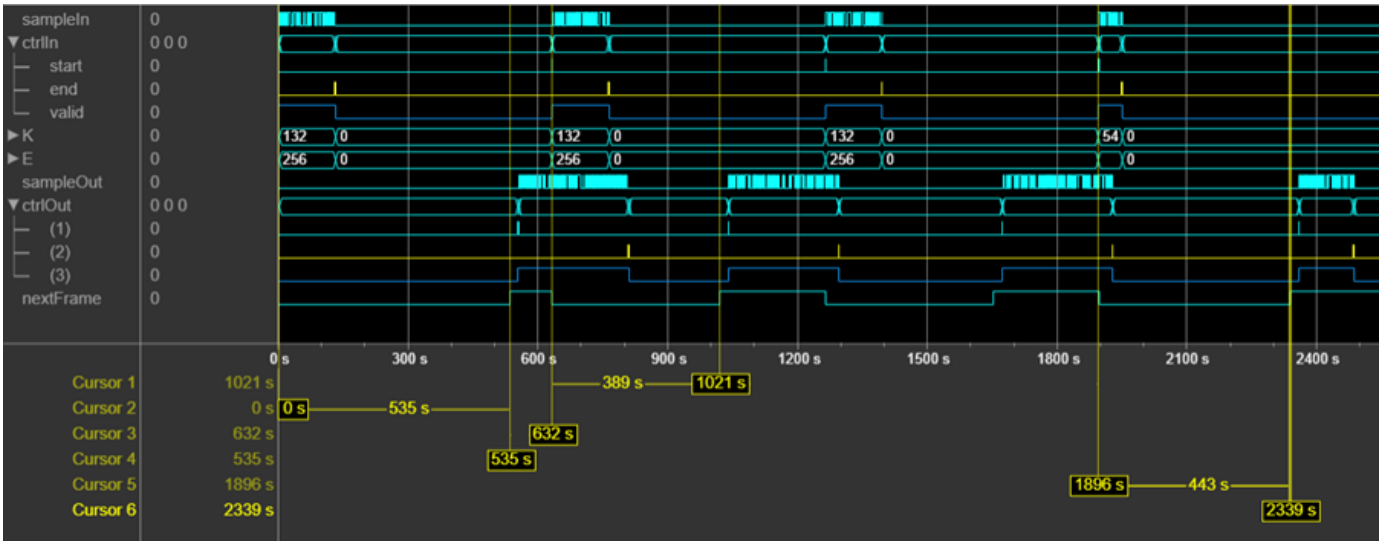
The block uses the Configuration stage when the input **K** and **E** values change. The block computes the new message length, *N*, and the locations of the information bits, then passes them to the buffer and the mapping stage. Because the mapping patterns are computed as needed, rather than stored in hardware, the block supports all **K** and **E** values within the supported range. The Configuration stage also computes the interleave pattern when you set the **Link direction** parameter to `Downlink`.

When you set the **Configuration source** parameter to `Property`, the **K** and **E** values are constants, so the decoder does not implement the Configuration stage. In this case, the block includes static lookup tables that contain the precomputed configuration.

Latency

The exact latency varies based on the values of **K** and **E**. The latency is longer for frames where the **K** and **E** values change and the block must compute the new configuration. Because the latency varies, use the output **nextFrame** control signal to determine when the block is ready for a new input frame.

This waveform shows how the latency varies with values of **K** and **E**. For the first frame with a given **K** and **E** value, the block must determine the message length and information bit mapping for those values. This configuration step means that the block takes longer to start returning the encoded samples. In this case, the block also takes longer before it is ready to accept the next input frame. When the input **K** and **E** values are 132 and 256, respectively, the block has a latency of 535 cycles from the input **start** signal to the output **nextFrame**. 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. The waveform shows this new latency is 389 cycles. When the **K** and **E** values change to 54 and 124, respectively, the block must compute the new configuration and the latency changes to 443 cycles.



Performance

This table shows the resource and performance data synthesis results of the block when it is configured with **K** and **E** as input ports and the **Link direction** parameter set to `Uplink`. The generated HDL is targeted to a Xilinx Zynq-7000 ZC706 evaluation board. The design achieves a clock frequency of 450 MHz.

Resource	Number Used
Slice LUTs	637
Slice Registers	934
Block RAM	2.5

This table shows the resource and performance data synthesis results of the block when it is configured with **K** and **E** as input ports and the **Link direction** parameter set to `Downlink`. The generated HDL is targeted to a Xilinx Zynq-7000 ZC706 evaluation board. The design achieves a clock frequency of 450 MHz.

Resource	Number Used
Slice LUTs	600
Slice Registers	948
Block RAM	3.5

The block uses fewer resources when **K** and **E** are specified by parameters.

References

[1] 3GPP TS 38.211. "NR; Physical channels and modulation." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*. URL: https://www.3gpp.org.

[2] Arikan, Erdal. "Channel Polarization: A Method for Constructing Capacity-Achieving Codes for Symmetric Binary-Input Memoryless Channels." *IEEE Transactions on Information Theory* 55, no. 7 (July 2009): 3051–73. https://doi.org/10.1109/TIT.2009.2021379.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

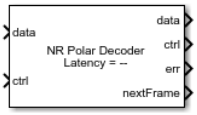
See Also

NR Polar Decoder | `nrPolarEncode`

Introduced in R2020a

NR Polar Decoder

Perform polar decoding according to 5G NR standard
Library: Wireless HDL Toolbox / Error Detection and Correction



Description

The NR Polar Decoder block implements a streaming polar decoder with hardware-friendly control signals. You can configure the block to use downlink or uplink coding schemes as defined by the 5G NR standard. The 5G NR standard uses polar codes for channel coding of the DCI, UCI, and BCH transmit channels.

This block implements a CRC-aided successive-cancellation list decoder. This implementation matches the performance of the `nrPolarDecode` function. You can choose a list length of 2, 4, or 8. Increasing the list length increases the error correction performance but uses more hardware resources and increases the decoding latency. You can improve decoding performance for DCI messages by using the optional **RNTI** port to specify an expected RNTI value.

This block also performs CRC decoding of the message, equivalent to the `nrCRCDecode` function. The block detects DCI messages from the values of **K** and **E**, and automatically prepends 1s to the message, equivalent to the `padCRC` input argument of the `nrPolarDecode` function.

This block does not support parity-aided codes.

Because the latency of the polar decoding operation can vary, the block provides an output signal, **nextFrame**, that indicates when the block is ready to accept new inputs. For more details, see the “Latency” on page 1-111 section of this page.

Ports

Input

`data` — Input sample
scalar

Input sample, specified as a scalar log-likelihood ratio (LLR). The block supports builtin types, and signed fixed-point values with a wordlength of 4 to 16 bits.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `fixed point` | `int8` | `int16` | `double` | `single`

`ctrl` — Control signals accompanying sample stream
`samplecontrol` bus

Control signals accompanying the sample stream, specified as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the input frame
- `end` — Indicates the end of the input frame
- `valid` — Indicates that the data on the input **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

K — Length of information block in bits

positive integer

Length of information block in bits, specified as a positive integer. For downlink messages, **K** must be in the range 36 to 164. For uplink messages, **K** must be in the range 31 to 1023.

The block does not support **K** values from 18 to 25 because the 5G NR standard requires parity-aided codes for those sizes.

Dependencies

To enable this port, set the **Configuration source** parameter to `Input port`.

Data Types: `fixdt(0,10,0)`

E — Rate-matched output length in bits

scalar positive integer

Rate-matched output length in bits, specified as a scalar positive integer. Specify a value for **E** that is greater than **K** and less than or equal to 8192.

Dependencies

To enable this port, set the **Configuration source** parameter to `Input port`.

Data Types: `fixdt(0,14,0)`

RNTI — Target RNTI

16-bit unsigned integer

Target RNTI, specified as a 16-bit unsigned integer. This value increases decoding accuracy for DCI messages. For example, when decoding SIB1 DCI messages, enable this port and set the target RNTI to 65,535. For MIB decoding, you can disable this port or set the target RNTI to 0.

Dependencies

To enable this port, set **Link direction** to `Downlink` and select the **Enable target RNTI port** parameter.

Data Types: `uint16`

Output

data — Decoded data bit

scalar

Decoded data bit, returned as a scalar. The output message length is *A* bits, where $A = K - CRCLen$. For downlink messages, *CRCLen* is 24. For uplink messages, *CRCLen* is 11, as defined by the 5G NR standard.

Data Types: `fixdt(0,1,0)` | `Boolean` | `double` | `single`

`ctrl` — Control signals accompanying sample stream
`samplecontrol` bus

Control signals accompanying the sample stream, returned as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the output frame
- `end` — Indicates the end of the output frame
- `valid` — Indicates that the data on the output **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

`err` — CRC result
scalar

CRC result, returned as a scalar. If you clear the **Full checksum mismatch** parameter, this value is a Boolean. When you select the **Full checksum mismatch** parameter, this value is a `ufix24` scalar for downlink messages and a `ufix11` scalar for uplink messages.

If you enable the **RNTI** port, the block compares the internal CRC checksum against the target RNTI value. Otherwise, the block compares the CRC checksum against a value of 0.

Data Types: `Boolean` | `ufix11` | `ufix24`

`nextFrame` — Ready for new inputs
scalar

The block sets this signal to `1` when the block is ready to accept the start of the next frame. 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.

For more information, see “Using the nextFrame Output Signal”.

Data Types: `Boolean`

Parameters

`Link direction` — Direction of 5G NR link
`Downlink` (default) | `Uplink`

Direction of 5G NR link, specified as `Downlink` or `Uplink`. When you choose `Downlink`, the block performs deinterleaving, as specified in the 5G NR standard. When you select `Uplink`, the block omits the deinterleaving logic.

`List length` — Number of decoding paths
2 (default) | 4 | 8

This parameter is the maximum number of parallel paths maintained in the decoding tree. Increasing the list length increases the error correction performance but uses more hardware resources and increases the decoding latency. When you use list lengths of 4 and 8, the latency can vary depending

on the SNR of the input signal, and is not constant for given values of **K** and **E**. Use the **nextFrame** output signal to determine when the block is available for a new message.

`Configuration source` — Source for K and E
`Property` (default) | `Input port`

Select `Input port` to enable the **K** and **E** ports. Select `Property` to use the **Message length (K)** and **Rate-matched length (E)** parameters.

`Message length (K)` — Length of information block in bits
56 (default) | positive integer

For downlink messages, **K** must be in the range 36 to 164. For uplink messages, **K** must be in the range 31 to 1023.

The block does not support **K** values from 18 to 25 because the 5G NR standard requires parity-aided codes for those sizes.

Dependencies

To enable this parameter, set the **Configuration source** parameter to `Property`.

`Rate-matched length (E)` — Rate-matched output length in bits
864 (default) | positive integer

Specify a value for **E** that is greater than **K** and less than or equal to 8192.

Dependencies

To enable this parameter, set the **Configuration source** parameter to `Property`.

`Full checksum mismatch` — Return checksum from final decoding stage
`off` (default) | `on`

When you clear this parameter, the block returns a Boolean scalar on the **err** port that indicates whether the CRC was successful. When you select this parameter, the block returns the full CRC checksum on the **err** port. If your design decodes DCI messages and makes use of the RNTI remainder, select this parameter.

If you enable the **RNTI** port, the block compares the internal CRC checksum with the target RNTI value. Otherwise, the block compares the CRC checksum against a value of 0.

The `nrPolarDecode` function returns a decoded message that includes the CRC bits. This block returns the decoded message without the CRC bits, and returns the CRC status separately on the **err** port. This behavior is equivalent to calling the `nrCRCDecode` function after using the `nrPolarDecode` function. Not recomputing the CRC bits saves hardware latency and resources.

`Enable target RNTI port` — Optional port to specify target RNTI value
`off` (default) | `on`

Select this parameter to enable the **RNTI** input port. Providing a target RNTI value increases decoding accuracy for DCI messages. For example, when decoding SIB1 DCI messages, enable this port and set the target RNTI to 65535. For MIB decoding, you can disable this port or set the target RNTI to 0.

Enabling this port also changes how the block computes the **err** output port value. If you enable the **RNTI** port, the block compares the internal CRC checksum with the target RNTI value. Otherwise, the block compares the CRC checksum against a value of 0.

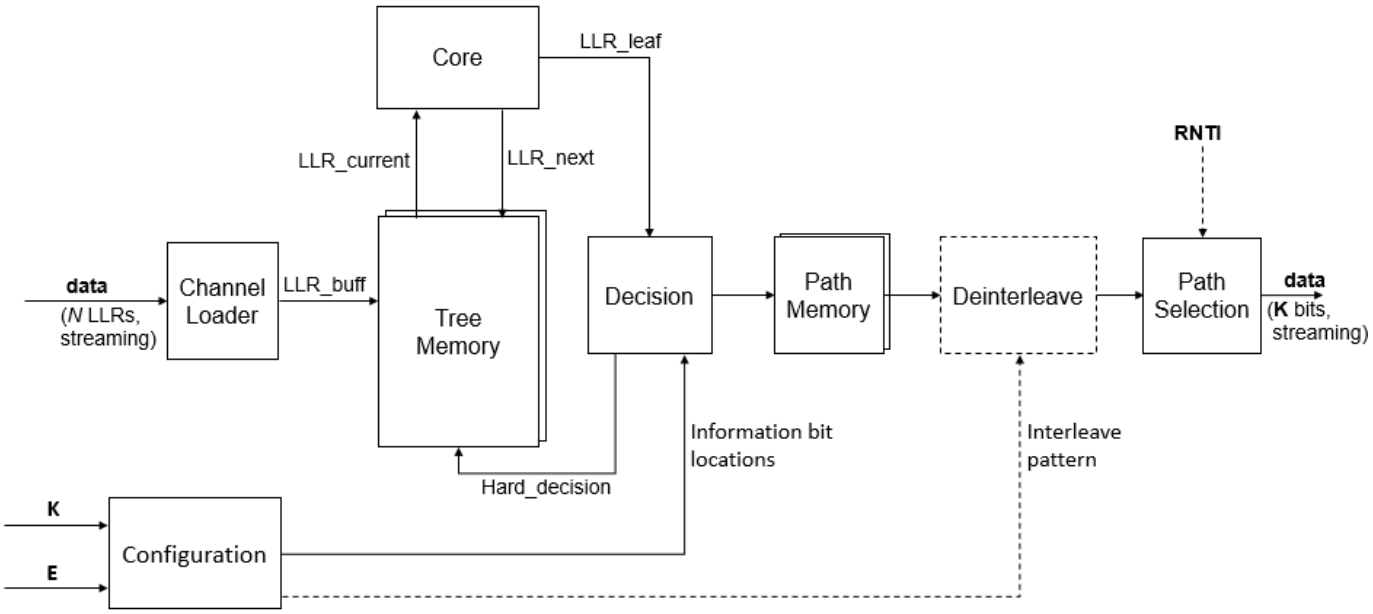
Dependencies

To enable this parameter, set the **Link direction** parameter to `Downlink`.

Algorithms

This block implements a CRC-aided successive-cancellation list decoder. It can use a list length of 2, 4, or 8 as configured by the **List length** parameter. The decoder iterates over all LLRs in the tree to reach a decision for a bit and then uses that decision to decode the next bit. The deinterleaving step is included only when you set the **Link direction** parameter to `Downlink`.

This diagram shows the architecture of the polar decoder.



The block uses the Configuration stage when the **K** and **E** input port values change. The block computes the locations of the information bits and passes them to the Decision stage. Because the mapping patterns are computed as needed, rather than stored in hardware, the block supports all **K** and **E** values within the supported range. The Configuration stage also computes the interleave pattern when you set the **Link direction** parameter to `Downlink`.

When you set the **Configuration source** parameter to `Property`, the **K** and **E** values are constants, so the decoder does not implement the Configuration stage. In this case, the block includes static lookup tables that contain the precomputed configuration.

To minimize computations for each decode, the Tree Memory stores the probability of each node being a one or a zero. Each iteration updates only the LLRs that have changed. The Core decoding stage uses the LLR update equations from [3].

The Decision stage checks the LLR value against the expected locations of information bits and frozen bits and returns a hard decision to the Tree Memory. If the bit is expected to be frozen, the Decision

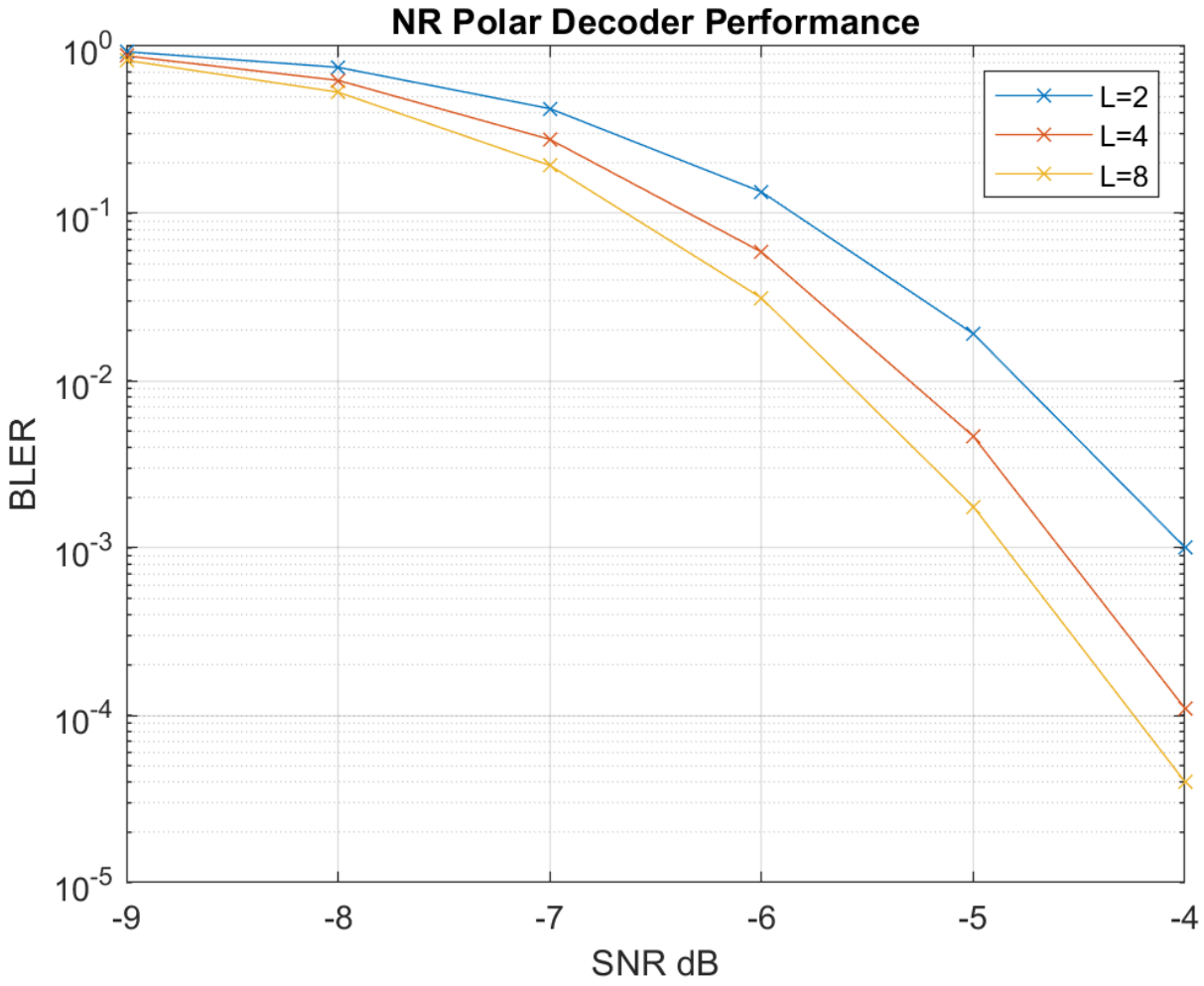
stage returns a hard decision of zero and updates the probabilities of related paths. The Path Memory reconstructs the most likely paths from the hard decision results and passes the paths and scores to the next stage.

Tree Memory and Path Memory contain up to **List length** paths. If all frozen bits on a path are zeros (as expected), then the block discards the other parallel paths. This optimization results in variable latency in the decoding operation for list lengths greater than two. For signals with a high noise level, the decoder must increase the number of parallel paths and the cycles for decoding. For low-noise signals, the decoder can use only two parallel paths and reduce the decoding latency.

The Path Selection stage computes the CRC for all paths and then chooses the path that passes the CRC. When you use the **RNTI** input port, the block compares the internal CRC checksum with the target RNTI value. Otherwise, the block compares the CRC checksum against a value of 0. If all CRCs fail, the block returns the path that has the higher score.

This implementation matches the performance of the 5G Toolbox™ function `nrPolarDecode` with the same list length. Because the block uses fixed-point internal types, any differences are a result of quantization.

This plot shows the block error rate performance with the three possible list lengths. The input is 6-bit LLR values.



Latency

The table shows example latencies of the NR Polar Decoder block for each *N*, when decoding for uplink and downlink channels with a list length of two. *N* is the power-of-two encoded message length determined from the values of **K** and **E**.

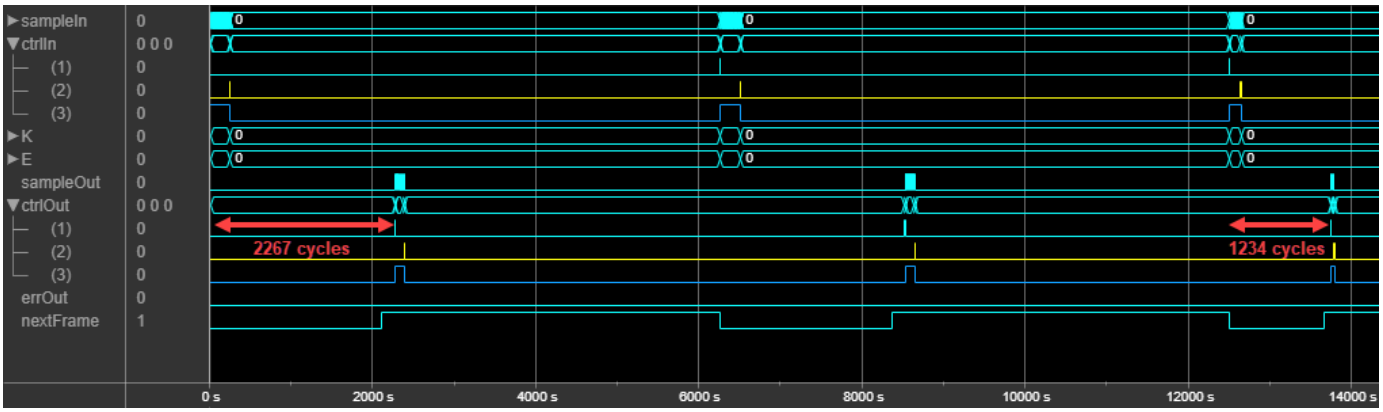
N	Uplink Latency	Downlink Latency
32	349	Not applicable
64	576	677
128	1034	1135
256	1961	2062
512	3896	3996
1024	8202	Not applicable

The exact latency varies based on the values of **K** and **E**. The latency is longer for frames where the **K** and **E** input port values change and the block must compute the new configuration.

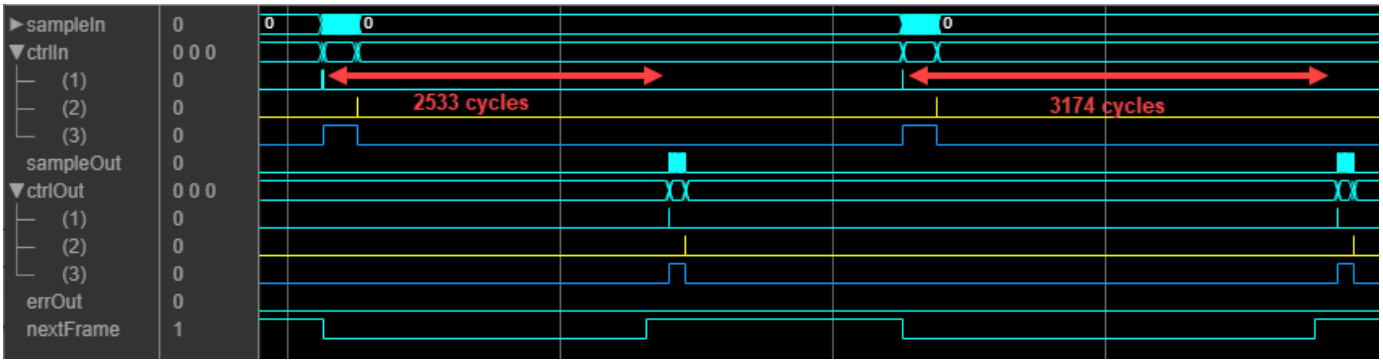
Increasing the list length increases the latency. List lengths greater than two do not have a fixed latency for given **K** and **E** values. To provide minimal latency, the block traces more than 2 paths only when the frozen bits are not decoded as zeroes. This optimization means that the latency can increase with the SNR of the input signal. For example, for a list length of 4 and N=512, the best case latency is 4108 cycles, and the worst case latency is 4985 cycles.

Because the latency varies, use the output **nextFrame** control signal to determine when the block is ready for a new input frame.

This waveform shows how the latency varies with the **K** and **E** input port values for a list length of two. When the input **K** and **E** port values are 132 and 256, the block has a latency of 2272 cycles from the input **start** signal to the output **nextFrame**. When the **K** and **E** port values change to 54 and 124, the latency changes to 1234 cycles.



This waveform shows how the latency can vary with the noise level of the input signal when using a list length of 4. The block has **K** and **E** parameter values of 132 and 256 and **Link direction** parameter set to `Uplink`. The first message has a latency of 2533 cycles. This message data is generated with low noise and has few bit errors. In this case, the decoder can collapse to two paths and produce a result in fewer cycles than when decoding a noisier signal. The second message is generated with a high noise level, and the decoding latency increases to 3174 cycles. When the input signal has more bit errors, the decoder must trace more paths to determine the correct bits.



Performance

This table shows the resource and performance data synthesis results of the block when it is configured with **K** and **E** as input ports, the **Link direction** parameter set to `Downlink`, and 6-bit input LLRs. The generated HDL is targeted to a Xilinx Zynq-7000 ZC706 evaluation board. The design achieves a clock frequency of 250 MHz.

Resource	List Length of 2	List Length of 4	List Length of 8
Slice LUTs	3048	4725	9963
Slice Registers	2509	3804	6471
DSP48	0	0	0
Block RAM	4.5	5.5	6.0

The block uses fewer resources when **K** and **E** are specified by parameters. When you set the **Link direction** parameter to `Uplink`, the block uses more memory to accommodate larger message sizes.

References

[1] 3GPP TS 38.211. "NR; Physical channels and modulation." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*. URL: https://www.3gpp.org.

[2] Arikan, Erdal. "Channel Polarization: A Method for Constructing Capacity-Achieving Codes for Symmetric Binary-Input Memoryless Channels." *IEEE Transactions on Information Theory* 55, no. 7 (July 2009): 3051–73. https://doi.org/10.1109/TIT.2009.2021379.

[3] Balatsoukas-Stimming, Alexios, Mani Bastani Parizi, and Andreas Burg. "LLR-Based Successive Cancellation List Decoding of Polar Codes." *IEEE Transactions on Signal Processing* 63, no. 19 (October 2015): 5165–79. https://doi.org/10.1109/TSP.2015.2439211.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).

OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

See Also

NR Polar Encoder | `nrPolarDecode`

Introduced in R2020a

LTE OFDM Demodulator

Demodulate time-domain OFDM samples and return LTE resource grid
Library: Wireless HDL Toolbox / Modulation

Description

The LTE OFDM Demodulator block implements an algorithm for demodulating LTE signals specified by LTE standard TS 36.212 [1]. The block returns the LTE resource grid that is used for cell ID detection, master information block (MIB) recovery, system information block (SIB)1 recovery, and further decoding.

You can select the number of downlink resource blocks (NDLRB) and choose either normal or extended cyclic prefix (CP), as described in the LTE standard. The block implements a CP fraction to support windowed LTE transmission and provides a parameter to configure the location of prefix removal.

The block provides an interface and architecture suitable for HDL code generation and hardware deployment.

The block accepts input data either at maximum rate of 30.72 MHz, or at a sample rate corresponding to NDLRB. The input sampling rates for NDLRB 6, 15, 25, 50, 75, and 100 are 1.92 MHz, 3.84 MHz, 7.68 MHz, 15.36 MHz, 30.72 MHz, and 30.72 MHz, respectively. The block uses a 2048-point fast fourier transform (FFT) for all values of NDLRB and returns the number of resource grid samples needed for the selected NDLRB. By default, the block excludes the direct current (DC) carrier.

The latency from the first input sample to the first output sample depends on your selection of the NDLRB and type of cyclic prefix, as shown in this table.

	Maximum Sample Rate		**Corresponding to NDLRB Sample Rate**	
NDLRB	**Latency — Normal CP**	**Latency — Extended CP**	**Latency — Normal CP**	**Latency — Extended CP**
6	5295	5647	6654	6676
15	5241	5593	6520	6564
25	5181	5533	6660	6748
50	5031	5383	6700	6876
75	4881	5233	6930	7282
100	4731	5083	6780	7132

Ports

Input

`data` — Input data
scalar

Input data, specified as a signed real or complex number.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `signed fixed point`

`valid` — Indicates valid input data
`Boolean` scalar

Control signal that indicates when the sample from the **data** input port is valid. When this value is `1` (true), the block captures the values on the **data** input port. When this value is `0` (false), the block ignores the input **data** samples.

Data Types: `Boolean`

`NDLRB` — Number of downlink resource blocks
`6` | `15` | `25` | `50` | `75` | `100`

Number of downlink resource blocks, specified as `6`, `15`, `25`, `50`, `75`, or `100`. NDLRB must be one of these six values specified by LTE standard TS 36.212 [1]. The block samples this port at the start of each subframe and ignores any changes within a subframe.

Dependencies

To enable this port, set the **NDLRB source** parameter to `Input port`.

Data Types: `uint8` | `uint16` | `uint32` | `fixdt(0,K,0), K >= 7` | `single` | `double`

`cyclicPrefixType` — Type of CP
scalar

Type of CP, specified as a `Boolean` scalar. When this value is `0` (false), the block selects normal CP. When this value is `1` (true), the block selects extended CP. The block samples this port at the start of each subframe and ignores any changes within a subframe.

Dependencies

To enable this port, set the **Cyclic prefix source** parameter to `Input port`.

Data Types: `Boolean`

`reset` — Clear internal states
scalar

Clears internal state, specified as a `Boolean` scalar. When this value is `1` (true), the block stops the current calculation and clears all internal states. When this value is `0` (false), and the **valid** input value is `1` (true), the block begins a new subframe.

Dependencies

To enable this port, select the **Enable reset input port** parameter.

Data Types: `Boolean`

Output

`data` — Output data
scalar

Output data, returned as a signed real or complex number. The data type is the same as the data type of the input **data** port. When you clear the **Divide butterfly outputs by two** parameter, the output word length increases by 11 bits to avoid overflow.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `signed fixed point`

`valid` — Indicates valid output data
scalar

Control signal that indicates when the **data** output port is valid. The block sets this value to `1` (true) when the resource grid samples are available on the **data** output port. When **Remove DC subcarrier** is selected, this value is set to `0` (false) at the center of the output samples to exclude the DC carrier.

Data Types: `Boolean`

`ready` — Indicates block is ready
scalar

Control signal that indicates when the block is ready for new input data. When this value is `1` (true), the block accepts input data in the next time step. When this value is `0` (false), the block ignores input data in the next time step.

Dependencies

To enable this port, set the **Input data sample rate** parameter to `Match input data sample rate to NDLRB`.

Data Types: `Boolean`

Parameters

Main

`NDLRB source` — Source of NDLRB
`Property` (default) | `Input port`

You can set NDLRB with an input port or by selecting a value for the parameter. To enable the **NDLRB** parameter, select `Property`. To enable the **NDLRB** port, select `Input port`.

`NDLRB` — Number of downlink resource blocks
`6` (default) | `15` | `25` | `50` | `75` | `100`

Number of downlink resource blocks, specified as `6`, `15`, `25`, `50`, `75`, or `100`. NDLRB must be one of these six values specified by LTE standard TS 36.212 [1].

Dependencies

To enable this parameter, set the **NDLRB source** parameter to `Property`.

`Cyclic prefix source` — Source of cyclic prefix type
`Property` (default) | `Input port`

You can set the cyclic prefix by selecting a parameter value or using an input port. To enable the **Cyclic prefix type** parameter, select `Property`. To enable the **cyclicPrefixType** port, select `Input port`.

`Cyclic prefix type` — Type of cyclic prefix
`Normal` (default) | `Extended`

Type of cyclic prefix, specified as `Normal` or `Extended`.

Dependencies

To enable this parameter, set the **Cyclic prefix source** parameter to `Property`.

`CP fraction` — Percent of cyclic prefix to remove
`0.55` (default) | value from 0 to 1

Cyclic prefix fraction, specified as a value from 0 to 1, inclusive. This parameter specifies the percentage of CP samples that the block removes from the start of the OFDM symbol. The block shifts the remaining CP samples to the end of the OFDM symbol.

When this parameter is `0.55`, the block removes 55% of the CP from the beginning of the symbol, and shifts 45% to the end of the symbol. When you set this parameter to `1`, the block removes 100% of the CP from the start of the OFDM symbol, and does not shift any samples to the end.

CP fraction provides windowed LTE transmission support. When a transmitter applies windowing, symbols are cyclically extended and overlapped. In a receiver design, the best location to remove the prefix and extract the symbol depends on windowing settings at the transmitter. For more information on windowing for an LTE transmitter, see the Algorithms section of `lteOFDMModulate` function.

`Remove DC subcarrier` — Exclude or include DC subcarrier
`off` (default) | `on`

When you select this parameter, the block excludes the DC subcarrier in the resource grid output. The DC subcarrier is present at the center of the 12×NDLRB subcarriers. The block excludes the DC subcarrier by setting the **valid** signal low (`false`) for the center cycle of the output subcarriers.

`Enable reset input port` — Reset signal
`off` (default) | `on`

Select this parameter to enable the **reset** port on the block icon.

`Input data sample rate` — Input sample rate
`Use maximum input data sample rate` (default) | `Match input data sample rate to NDLRB`

This parameter specifies the type of sample rate to select for the input data.

- To provide an input data sample rate of 30.72 MHz, select `Use maximum input data sample rate`.
- To provide an input data sample rate based on the **NDLRB** parameter, select `Match input data sample rate to NDLRB`. The input sampling rates for **NDLRB** values `6`, `15`, `25`, `50`, `75`, and `100` are 1.92 MHz, 3.84 MHz, 7.68 MHz, 15.36 MHz, 30.72 MHz, and 30.72 MHz, respectively.

For more information, see “Data Rate Controller” on page 1-120.

FFT Parameters

`Divide butterfly outputs by two` — Divide FFT butterfly outputs by two
`off` (default) | `on`

This parameter controls the scaling option of the FFT HDL Optimized block inside the LTE OFDM Demodulator.

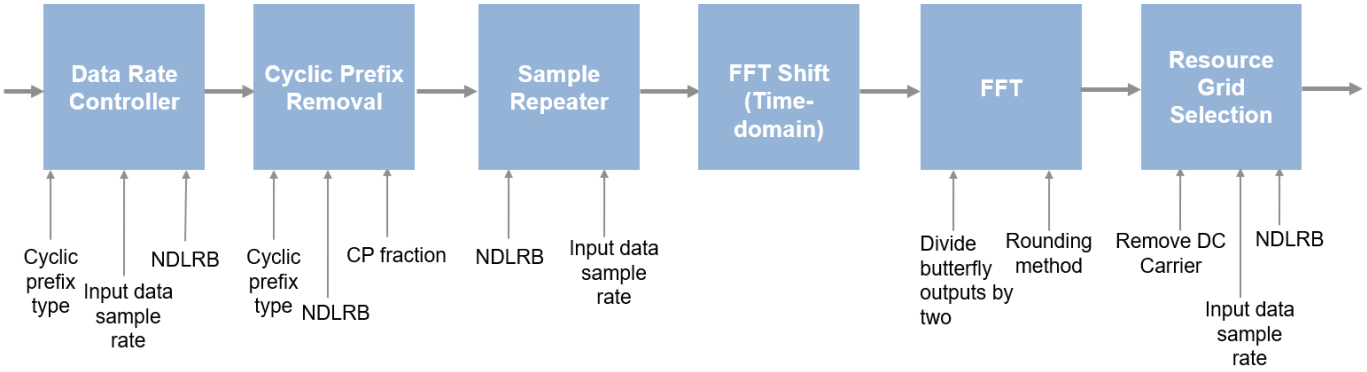
When you select this parameter, the FFT implements an overall 1/*N* scale factor by dividing the output of each butterfly multiplication by two. This adjustment keeps the output of the FFT in the same amplitude range as its input. If you disable this parameter, the block avoids overflow by increasing the word length by one bit after each butterfly multiplication.

`Rounding Method` — Rounding mode for internal fixed-point calculations
`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Zero`

This parameter specifies the type of rounding mode for internal fixed-point calculations. For more information about rounding modes, see Rounding Modes (DSP System Toolbox). When the input is any integer data type or fixed-point data type, the FFT algorithm uses fixed-point arithmetic for internal calculations. This parameter does not apply when the input is of data type `single` or `double`. Rounding applies to twiddle-factor multiplication and scaling operations.

Algorithms

The LTE OFDM Demodulator block operation sequence is carried over using these blocks: Data Rate Controller, CP Prefix Removal, Sample Repeater, FFT Shift, FFT, and Resource Grid Selection. The Data Rate Controller block helps in controlling the input data rate by generating a ready signal. The CP Removal block removes the part of the CP at the start of a symbol and the remainder of the CP at the end of the symbol. The Sample Repeater block repeats the samples based on the NDLRB values. The block repeats the samples until they form 2048 samples and converts the input data rate to the maximum rate supported by LTE. The FFT Shift block performs a time-domain FFT shift. The FFT block converts the frequency-domain signal to a time-domain signal. The Resource Grid Selection block extracts the resource grid elements based on the NDLRB and the input data sample rate, and provides the demodulated output. The parameters shown in this figure configure the behavior of the block.



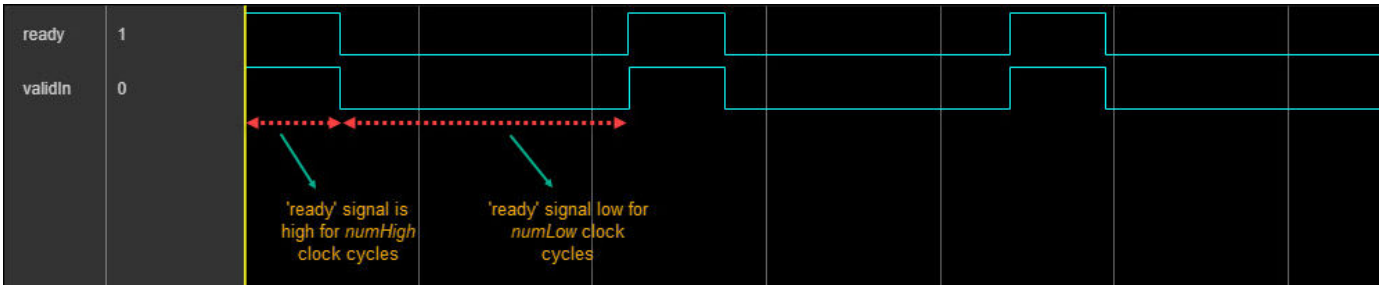
Data Rate Controller

The block accepts input data either at maximum rate of 30.72 MHz, or at a sample rate corresponding to NDLRB. The input sampling rates for NDLRB values 6, 15, 25, 50, 75, and 100 are 1.92 MHz, 3.84 MHz, 7.68 MHz, 15.36 MHz, 30.72 MHz, and 30.72 MHz, respectively.

When you set the **Input data sample rate** parameter to `Use maximum input data sample rate`, the block operates based on the demodulation parameters (**NDLRB** and **CP prefix type**) and provides output data along with an output **valid** signal to the next block.

When you set the **Input data sample rate** parameter to `Match input data sample rate to NDLRB`, the block generates an output **ready** signal and controls the input at a rate with respect to the NDLRB. The block accepts data samples with respect to the NDLRB when the **ready** signal is `1` (true).

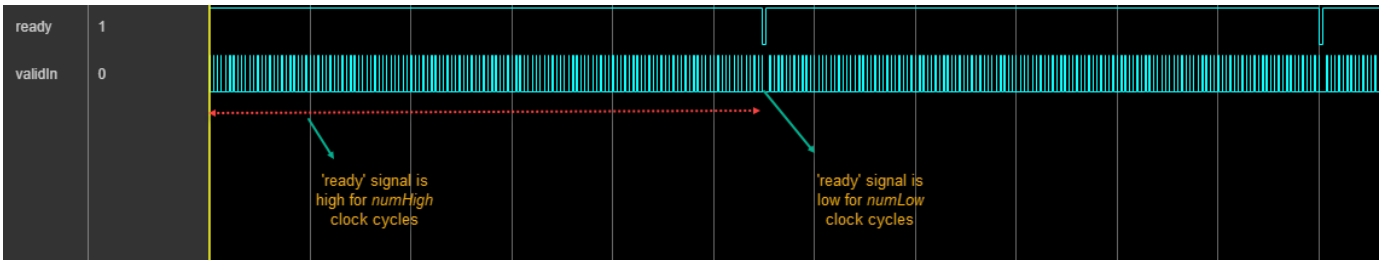
This figure shows the timing diagram of the **ready** signal generation for a continuous input when **NDLRB** is 6 and **CP prefix type** is `Normal`, having FFT length as 128 and CP length as 10.



numHigh = *FFT length* + *CP length* = 128 +10 = 138 clock cycles.

numLow = *Maximum FFT length* + *Maximum CP length* - (*numHigh*) = 2048 + 160 - (138) = 2070 clock cycles.

This figure shows the timing diagram of the **ready** signal generation for a discrete input with `1` (high) for 16 clock cycles when **NDLRB** is 6 and **CP prefix type** is `Normal`, having FFT length as 128 and CP length as 10.



numHigh = (*FFT length* + *CP length* -1) * *Maximum FFT length* / *FFT length* + 1 = (128 +10 -1) * 16 + 1 = 2193 clock cycles.

numLow = *Maximum FFT Length* + *Maximum CP Length* - (*numHigh*) = 2048 + 160 - (*2193*) = 15 clock cycles.

Cyclic Prefix Removal

This block supports windowed LTE transmission by implementing fractional cyclic prefix removal. Windowing reduces out-of-band emissions. A transmitter performs windowing by overlapping the tail

of each OFDM symbol with the head of the next OFDM symbol. A receiver must avoid these overlapped samples in the FFT calculation. Fractional CP solves this problem by removing part of the CP at the start of a symbol and the remainder of the CP at the end of the symbol. Implementing a CP-fraction algorithm also makes the LTE OFDM Demodulator block less sensitive to timing offset.

The **Cyclic prefix type** parameter controls whether the block expects normal or extended CP. When the block operates at a maximum sample rate of 30.72 MHz, it assumes that each symbol is 2048 samples plus the cyclic prefix size associated with that rate. When using normal CP, the prefix of the first symbol in each slot has 160 samples, while subsequent symbols have a prefix of 144 samples. The extended CP has 512 samples.

When the block operates at sample rates with respect to the NDLRB value, for example, if the NDLRB is 6, the block receives 128 samples plus the cyclic prefix size associated with that rate. When using normal CP, the prefix of the first symbol in each slot has 10 samples, while subsequent symbols have a prefix of nine samples. The extended CP has 32 samples.

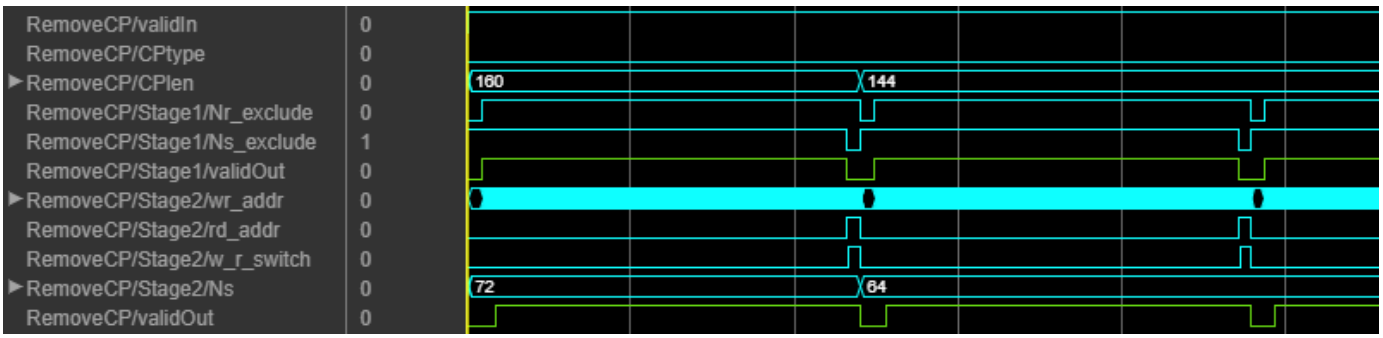
The block handles the CP in two stages. First, the block calculates the number of CP samples to remove, *N*r, and removes those samples from the input samples. Next, it calculates the number of samples to shift, *N*s, and shifts those samples to the end of OFDM symbol in the time domain. These two segments together make up the total cyclic prefix length, *N*cp = *N*s + *N*r.

The **CP fraction** parameter controls how many samples the block removes at the beginning of the symbol. The block shifts the remainder of the cyclic prefix from the start of the symbol to the end of the symbol. The block quantizes **CP fraction** to `fi(0,11,10)`. To achieve an integer number of samples, the block calculates *N*r = `floor` (*N*cp * **CP fraction**).

The waveform shows the control signals for the two stages of CP removal. The block is configured for a normal CP, so the CP of the first symbol is 160 samples. The CP for subsequent symbols is 144 samples. The **CP fraction** is `0.55`.

In stage one, the block sets the internal **valid** signal to `0` (`false`) to exclude the first *N*r samples of the symbol. For the first symbol, *N*r = 88. Manipulation of the **valid** signal also excludes the final *N*s samples, which are replaced by the shifted samples in the next stage. For the first symbol, *N*s = 72. In stage two, the block writes the *N*s samples to a RAM, and then reads and returns these samples at the end of the symbol. The block shifts the internal **valid** signal to include the shifted samples in their new location. The result is 2048 samples, properly aligned in the time domain in preparation for an FFT.

For the second symbol, with a cyclic prefix of 144 samples, *N*r = 80 and *N*s = 64.



For more information on LTE transmitter windowing, see the Algorithms section of the `lteOFDMModulate` function.

Sample Repeater

As the LTE OFDM Demodulator block uses a maximum FFT length of 2048. So, when the input samples corresponding to the actual FFT length are provided, the Sample Repeater block repeats the samples until it forms 2048 samples. For this operation, the block buffers the input samples first, and then repeats the samples based on the NDLRB value. This repetition mechanism helps avoid scaling at the FFT block input. For example, if the NDLRB is 6, each OFDM symbol consists of 128 samples. The block converts these 128 samples to 2048 samples by repeating them 16 times. After the block generates 2048 data samples, it sends **data** and **valid** signals to the next block.

Time-Domain FFT Shift

Conventionally, receivers perform FFT shift in the frequency domain. However, this method requires memory and introduces latency related to the size of the FFT. Instead, a receiver can execute the same operation in the time domain using the frequency shifting property of Fourier transforms. Shifting a function in one domain corresponds to a multiplication by a complex exponential function in the other domain. To reduce hardware resources and latency, this block performs the FFT shift by multiplying the time-domain samples by a complex exponential function.

These equations describe an FFT shift. The equation for an N-point FFT is

$$X(k) = F[x(n)] = \sum_{n=0}^{N-1} x(n)e^{-\frac{j2\pi nk}{N}}$$

For an FFT shift of $N/2$ carriers in either direction, substitute $k = k - \frac{N}{2}$, resulting in

$$X(k-\tfrac{N}{2}) = \sum_{n=0}^{N-1} x(n)e^{-\frac{j2\pi n(k-\frac{N}{2})}{N}}$$

This equation simplifies to

$$X(k-\frac{N}{2}) = \sum_{n=0}^{N-1} e^{j\pi n}x(n)e^{-\frac{j2\pi nk}{N}}$$

Since $\sum_{n=0}^{N-1} x(n)e^{-\frac{j2\pi nk}{N}}$ is equivalent to $F[x(n)]$, and $e^{j\pi} = -1$, this equation simplifies to

$$X(k-\frac{N}{2}) = F[(-1)^n x(n)]$$

The final equation shows that an FFT shift in the time domain simplifies to multiplication by $(-1)^n$. Therefore, the block implements the FFT shift by multiplying the time-domain samples by either +1 or –1.

FFT

The output of the FFT shift subsystem is fed to an FFT HDL Optimized block. The sample rate of the time-domain samples must be 30.72 MHz. The block calculates a 2048-point FFT for all NDLRB values.

The **Divide butterfly outputs by two** parameter controls whether the FFT implements an overall $1/N$ scale factor by dividing the output of each butterfly multiplication by two. This adjustment keeps

the output of the FFT in the same amplitude range as its input. When you disable scaling (default), the block avoids overflow by increasing the word length by one bit after each butterfly multiplication.

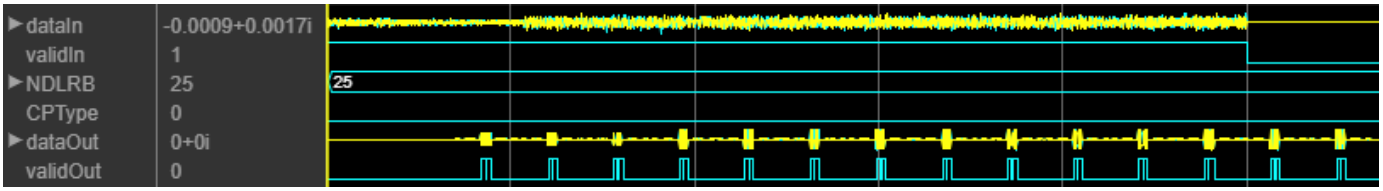
Resource Grid Selection

This part of the algorithm selects the appropriate number of subcarriers based on the NDLRB. Out of 2048 subcarriers, the block selects the center 12xNDLRB subcarriers for output.

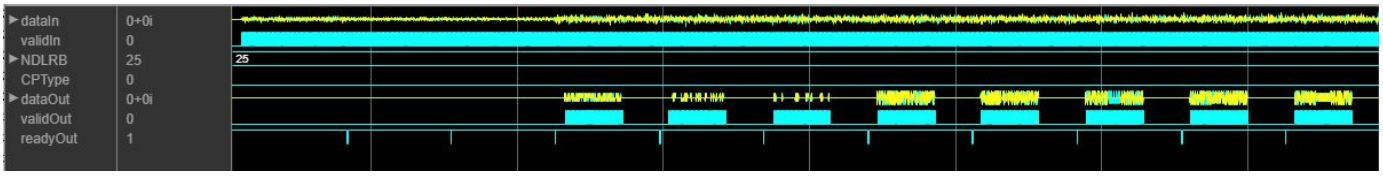
If the **Remove DC subcarrier** parameter is selected, the block excludes the DC subcarrier from the final resource grid output. The block excludes the DC subcarrier by setting the **valid** signal to `0` (false) for the center cycle of the output subcarriers.

For NDLRB 25, the block returns 12×25=300 resource grid samples. The block indicates the location of these output samples with the **validOut** signal set to `1` (true). The **validOut** signal is `0` (false) at the center of the output samples, to exclude the DC carrier.

This waveform shows the output when you set the **Input data sample rate** parameter to `Use maximum input data sample rate` and select an NDLRB of 25 with normal CP.



This waveform shows the output of the block when you set the **Input data sample rate** parameter to `Match input data sample rate to NDLRB` and select an NDLRB of 25 with normal CP. The block repeats four times at the input of FFT for computing 2048-point FFT. The actual FFT samples are taken for every one in four samples at the output of FFT. The block chooses the center (12×25 = 300) resource grid elements and outputs them along with valid signal.



Performance

The performance of the synthesized HDL code varies with your target and synthesis options. The input data type used for generating HDL code is `fixdt(1,16,14)`.

This table shows the resource and performance data synthesis results when using the block with default configuration. The generated HDL targeted to Xilinx Zynq XC7Z045I-FFG900-2L FPGA. The design achieves a clock frequency of 280 MHz.

Resource	Number Used
LUTs	6072
Registers	8291
DSPs	16

Resource	Number Used
Block RAM	23
F7 Muxes	0
F8 Muxes	0
RAMB36/FIFO	6
RAMB18	18

References

[1] 3GPP TS 36.212. "Multiplexing and channel coding." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA)*. URL: https://www.3gpp.org.

[2] Sesia, S., I. Toufik, and M. Baker, eds. *LTE - The UMTS Long Term Evolution : From Theory to Practice*. Hoboken, NJ: John Wiley & Sons Ltd., 2009.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

See Also

Blocks
LTE OFDM Modulator

Functions
`lteOFDMDemodulate` | `lteOFDMModulate`

Introduced in R2018a

Viterbi Decoder

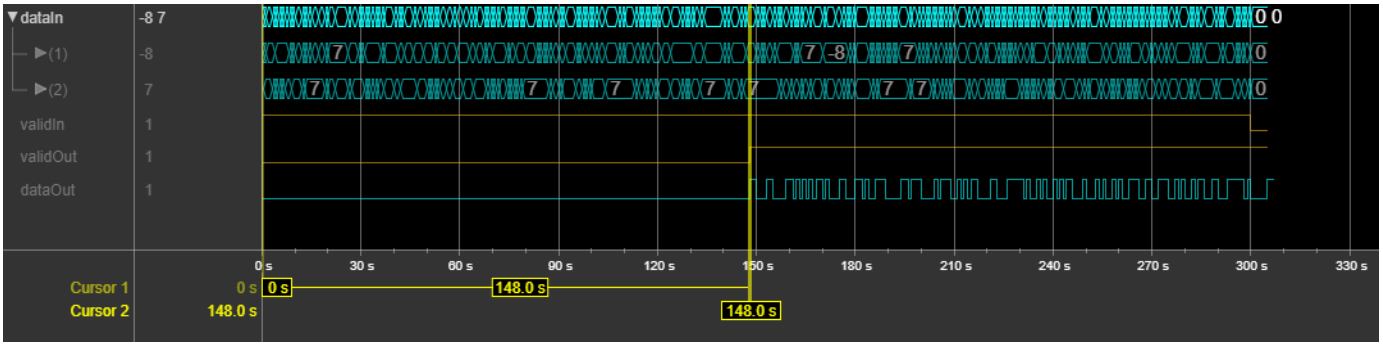
Decode convolutionally encoded data using Viterbi algorithm
Library: Wireless HDL Toolbox / Error Detection and Correction

Description

The Viterbi Decoder block decodes convolutionally encoded data using a RAM-based traceback implementation. Viterbi decoding is widely used in LTE standard TS 36.212 [1] and other forward-error-correction (FEC) applications such as wireless networks (802.11a/b/g/n/ac), digital satellite communications, digital video broadcast (DVB), IEEE 802.16, and HiperLAN. To support any of these standards, the block accepts convolution codes with constraint lengths of 3 to 9, code rates 1/2 to 1/7, and provides continuous, terminated, and truncated modes. The block provides an architecture and interface suitable for HDL code generation.

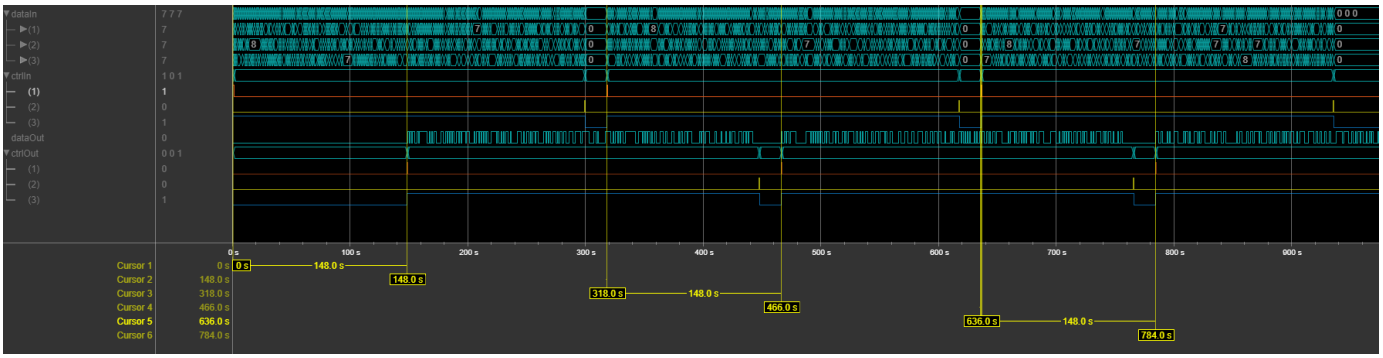
The block supports decoding of punctured codes by providing an optional **erasure** input port. You can use the Depuncturer block to insert neutral values in a punctured sample stream, and generate the **erasure** signal.

The Viterbi Decoder block accepts input samples as hard-decision binary values or soft-decision log-likelihood-ratios (LLR). Each sample is a column vector, whose length depends on the encoding scheme. The first waveform shows continuous operation mode with input samples of signed 4-bit data, using the default block parameters. The **Traceback depth** is 32. The block returns the first decoded output data sample after 148 clock cycles. The decoding latency is 4×**Traceback depth** + **Constraint length** + 13 valid input cycles.



The second waveform shows three frames in terminated operation mode. The input is unsigned 4-bit samples, and the block is using the trellis (7,[171 133 112]). The **Traceback depth** is 32. The input and output **ctrl** buses are expanded to show their three control signals. The latency from each input **ctrl**.`start` to output **ctrl**.`start` is also 148 clock cycles.

The control signals in the bus indicate the validity of each sample and the boundaries of the frame. To convert a matrix into a sample stream and corresponding control signals, use the Frame To Samples block or the `whdlFramesToSamples` function. For a full description of the streaming sample interface, see “Streaming Sample Interface”.



Ports

Input

`data` — Input sample

column vector

Input sample, specified as an *n*-by-1 column vector, where *n* is the length of the generator polynomial. The block performs soft-decision decoding when the input data type is fixed point or integer and hard-decision decoding when the input data type is `Boolean` or `fixdt(0,1,0)`. The block performs unquantized soft-decision decoding for `single` and `double` data types, but these data types are not supported for HDL code generation.

For soft-decision input samples, the block supports word lengths up to 16 bits. For HDL code generation, a word length more than 8 bits is not recommended due to hardware resource usage. The input data must have a fraction length of 0.

Data Types: `int8` | `int16` | `uint8` | `uint16` | `Boolean` | `fixdt(0,1,0)` | `fixdt(S,WL,0)` | `single` | `double`

`valid` — Validity of input samples

scalar

Control signal that indicates when the sample from the **data** input port is valid. When the **valid** input port is `1` (`true`), the block captures the values of the **data** input port. When the **valid** input port is `0` (`false`), the block ignores the input samples.

Dependencies

To enable this port, set **Operation mode** to `Continuous`.

Data Types: `Boolean`

`erasure` — Neutral symbol locations

column vector

Neutral symbol locations, specified as a column vector of binary values the same size as the input **data** vector. When an **erasure** element is `1` (`true`), the corresponding input **data** element is a depunctured neutral value, and the decoder does not update the branch metric. When an **erasure** element is `0` (`false`), the block uses the corresponding input **data** element to update the branch metric. You can use the Depuncturer block to drive this port.

Dependencies

To enable this port, select **Enable erasure input port**.

Data Types: `Boolean`

`reset` — Clear internal state

scalar

Clear internal state, specified as a `Boolean` scalar. When **reset** is `1` (`true`), after one cycle, the block stops the current calculation and clears internal branch and state metrics.

Dependencies

To enable this port, set **Operation mode** to `Continuous` and select **Enable reset input port**.

Data Types: `Boolean`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, specified as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the input samples.

Dependencies

To enable this port, set **Operation mode** to `Terminated` or `Truncated`.

Data Types: `bus`

Output

`data` — Output sample

scalar

Output sample, returned as a scalar with the same data type as the input samples.

Data Types: `int8` | `int16` | `uint8` | `uint16` | `Boolean` | `fixdt(0,1,0)` | `fixdt(S,WL,0)` | `single` | `double`

`valid` — Validity of output samples

scalar

Control signal that indicates when the sample from the **data** output port is valid. The block sets the **valid** port to `1` (`true`) when there is a valid sample on the output **data** port.

Dependencies

Tho enable this port, set **Operation mode** to `Continuous`.

Data Types: `Boolean`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, returned as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

Dependencies

To enable this port, set **Operation mode** to `Terminated` or `Truncated`.

Data Types: `bus`

Parameters

`Constraint length` — Trellis constraint length

7 (default) | integer in the range [3, 9]

Trellis constraint length, specified as an integer in the range [3, 9].

`Code generator` — Code generation polynomial

`[171, 133]` (default) | vector of octal values

Code generation polynomial, specified as a 1-by-*n* vector of octal values, where *n* is the length of the polynomial. The block accepts polynomials from 2 to 7 elements long.

`Enable erasure input port` — Use optional erasure signal

`off` (default) | `on`

Select this parameter to enable the **erasure** port.

`Traceback depth` — Number of trellis branches

32 (default) | integer in the range [3, 128]

Number of trellis branches used to construct each traceback path, specified as an integer. The block supports traceback depth in the range [3, 128]. For nonpunctured samples, the recommended depth is 5×**constraintLength**. For punctured samples, the recommended depth is 10×**constraintLength**. These values balance decoding accuracy with the amount of memory used.

`Operation mode` — End of frame behavior

`Continuous` (default) | `Truncated` | `Terminated`

End of frame behavior, specified as one of these modes:

- `Continuous` – The block does not clear the internal state metric. The input **valid** signal qualifies the input samples.
- `Truncated` – The block resets the state metrics after each frame, and the traceback path starts at the state with the best metric and ends in the all-zeros state. The input **ctrl** bus qualifies the input samples and marks the frame boundaries.

  Note This mode requires a minimum space of **Constraint length**–1 cycles between frames .
- `Terminated` – The block resets the state metrics after each frame, and the traceback path always starts and ends in the all-zeros state. The input **ctrl** bus qualifies the input samples and marks the frame boundaries.

`Enable reset input port` — Use optional reset signal

`off` (default) | `on`

Select this parameter to enable the **reset** port. When **reset** is `1` (`true`), after one cycle, the block stops the current calculation and clears internal branch and state metrics.

Dependencies

To enable this parameter, set **Operation mode** to `Continuous`.

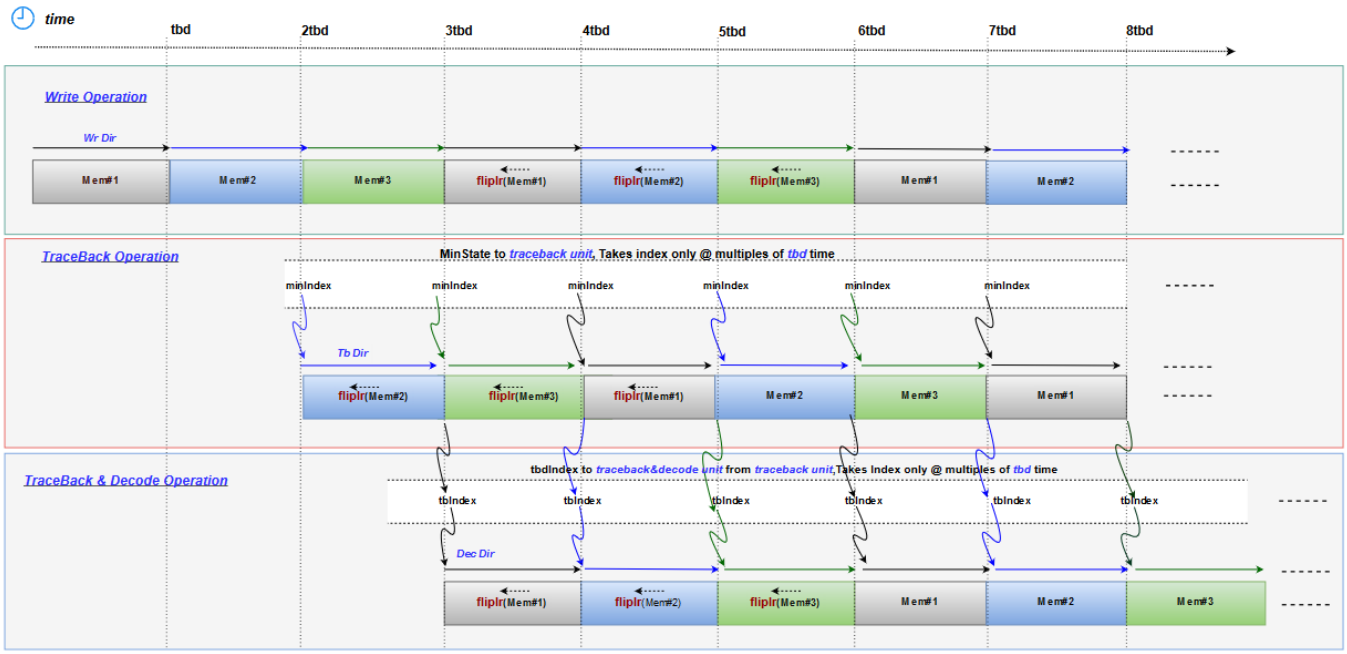
Algorithms

The Viterbi Decoder block implements a RAM-based traceback using *K*-pointer odd algorithm [2]. The parameter *K* is the number of read pointers required in the algorithm. This implementation has a *K* value of two, and accesses three memories using two read and one write pointers. The three types of operations are:

- Write (Wr): Save the survivor path information in memory.
- Traceback (TB): Read the survivor path information from the memory, and compute the previous state based on the current state and survivor branch. The earliest state traced by this process is then used as the initial state to decode the previous memory block of the data.
- Decoding (Dec): Read the survivor path information from the memory and decode the input.

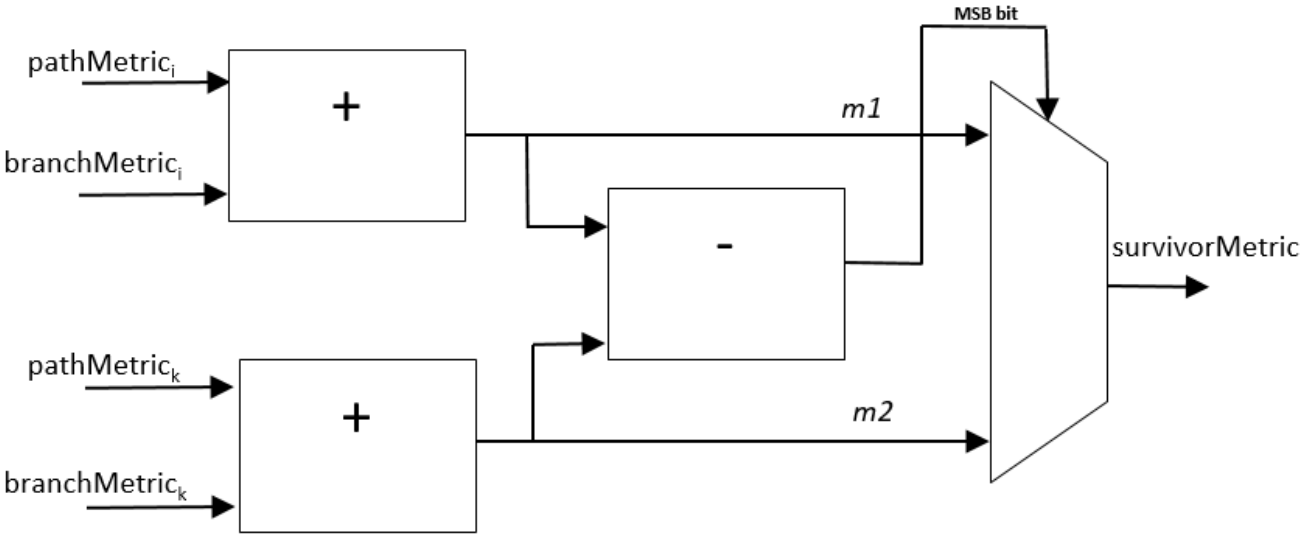
Each of the three memories is **Traceback depth** (*tbd*) size. The *K*-pointer algorithm takes 4×**Traceback depth** cycles to decode the data.

The diagram shows how the three operations use the three memory banks. For two multiples of **Traceback depth**, write the survivor metrics in forward direction to Mem#1 and Mem#2. Then, while continuing to write to Mem#3, traceback from Mem#2 to find the minimum index. Use that index as a pointer to traceback from Mem#1 and obtain the decoded value. After the decoded value is read, the block writes the new input survivor path to the same location. This write starts the next decoding cycle.



In the Viterbi algorithm, the add-compare logic can cause a state metric overflow. These overflows degrade the decoder performance. Modulo normalization is used to mitigate the effects of overflow [3]. The block implements modular arithmetic using two's complement adders and subtractors as

shown in the diagram. A subtractor is used in place of a comparator. An overflow is detected by checking the most significant bit (MSB) of *(m1-m2)*, where *m1m2* are the normalization metrics. When an overflow is detected, *m1* and *m2* provide a wrapped value.



Performance

These resource and performance data are the synthesis results from the generated HDL targeted to a XilinxZynq-7000 ZC706 board. The block is using `ufix4` input samples, in continuous mode with default settings. The design achieves a clock frequency of 260 MHz.

Resource	Number Used
LUT	3861
FFS	2521
Xilinx LogiCORE DSP48	0
Block RAM (16k)	2

The word length of the input samples affects the timing and the resources used in metric computation. Increasing the **Traceback depth** uses more RAM.

References

[1] 3GPP TS 36.212. "Multiplexing and channel coding." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA)*. URL: https://www.3gpp.org.

[2] Horwitz, M., and R. Braun. "A Generalised Design Technique for Traceback Survivor Memory Management in Viterbi Decoders." *Proceedings of the 1997 South African Symposium on Communications and Signal Processing*: 63-68. Piscataway, NJ: IEEE, 1997.

[3] Shung, C.b., P.h. Siegel, G. Ungerboeck, and H.k. Thapar. "VLSI Architectures for Metric Normalization in the Viterbi Algorithm." *IEEE International Conference on Communications, Including Supercomm Technical Sessions*: vol 4. 1726-728. New York, N.Y. : IEEE, 1990.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

See Also

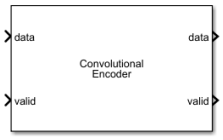
Blocks

Convolutional Encoder | Depuncturer

Introduced in R2018b

Convolutional Encoder

Encode data bits using convolution coding — optimized for HDL code generation
Library: Wireless HDL Toolbox / Error Detection and Correction



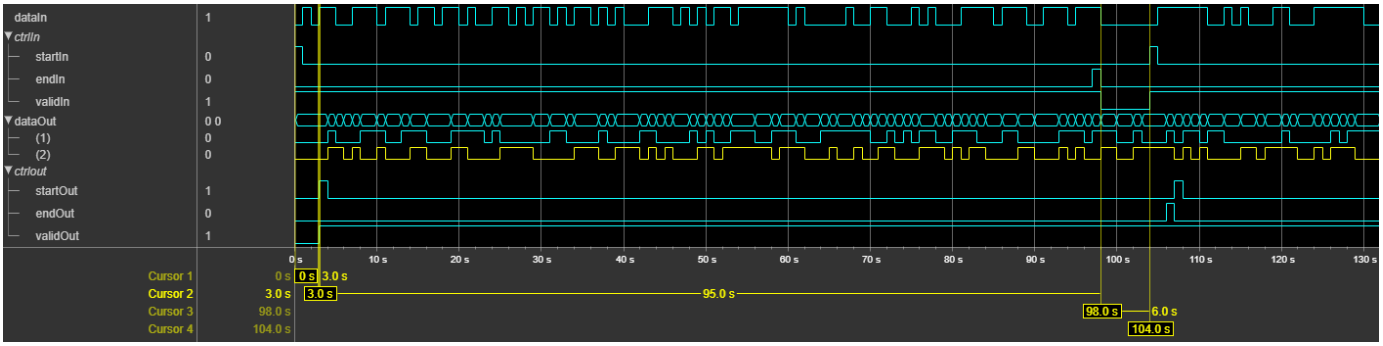
Description

The Convolutional Encoder block encodes data bits using convolution coding. The block supports code rates from 1/2 to 1/7 and constraint lengths from 3 to 9 including both recursive and nonrecursive polynomials. The block provides an architecture suitable for HDL code generation and hardware deployment.

The block operates in three modes: continuous with an optional reset port, terminated, and truncated with optional initial state and final state ports. In `Continuous` mode, the block accepts data bits, along with a valid signal, and outputs encoded bits with a valid signal. In `Terminated` and `Truncated` modes, the block accepts data bits, along with a `samplecontrol` bus and outputs encoded bits with a `samplecontrol` bus.

The block supports communication standards such as Wi-Fi (802.11a/b/g/n/ac), digital satellite communications, digital video broadcast (DVB), 3GPP2, IEEE 802.16, HIPERLAN, and other technologies. You can use this block to implement other channel codes such as turbo codes, which are used in LTE standards.

This waveform shows the encoded output of the block in `Terminated` mode, when block parameter **Constraint length** is set to 7, **Code generator** to `[133 171]`, and **Feedback connection** to `0`. The input and output **ctrl** buses are expanded to show their control signals.



The latency of the block is three clock cycles, so the block returns the first encoded output data after three clock cycles. In the `Terminated` mode, after the end of the frame, the block resets the encoded states to all zeros state by appending (**Constraint length** – 1) bits. So, the waveform shows the frame gap of six (**Constraint length** – 1) clock cycles between the end of the frame (**ctrlIn.endIn**) and the start of the next frame **ctrlIn.startIn**.

Ports

Input

`data` — Input data bits
scalar

Input data bits, specified as `Boolean` or `ufix1`.

Data Types: `Boolean` | `fixed point`

`valid` — Validity of input data
scalar

Control signal that indicates if the input data is valid. When this value is `1` (true), the block accepts the values on the **data** input port. When this value is `0` (false), the block ignores the values on the **data** input port.

Dependencies

To enable this port, set the **Operation mode** parameter to `Continuous`.

Data Types: `Boolean`

`reset` — Clears internal states
scalar

Clears internal states, specified as a `Boolean` scalar. When this value is `1` (`true`), the block stops the current calculation and clears all encoder states.

Dependencies

To enable this port, set the **Operation mode** parameter to `Continuous` and select the **Enable reset input port** parameter.

Data Types: `Boolean`

`ctrl` — Control signals accompanying sample stream
`samplecontrol` bus

Control signals accompanying the sample stream, specified as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the input samples.

- `start` — Indicates start of input frame.
- `end` — Indicates end of input frame.
- `valid` — Indicates the data on the input **data** port is valid.

Dependencies

To enable this port, set the **Operation mode** parameter to `Truncated` or `Terminated`.

Data Types: `bus`

`ISt` — Initial state at every start of frame
scalar

Initial state of block at every start of frame, specified as `fixdt(0,constraint length -1,0)`. Input state is the number of binary bits in the shift register at the frame start of the block, which is read from most significant bit (MSB) to least significant bit (LSB).

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Dependencies

To enable this port, set the **Operation mode** parameter to `Truncated` and select the **Enable initial state input port** parameter.

Data Types: `single` | `double` | `fixed point`

Output

`data` — Encoded output data
column vector

Output data, returned as 1-by-*n* column vector, if the code rate is 1/n. The *n* value ranges from 2 to 7.

Data Types: `Boolean`

`valid` — Validity of output samples
scalar

Control signal that indicates if the data from the **data** output port is valid. When this value is `1` (true), the block returns valid data on the **data** output port. When **valid** is `false` (`0`), the values on **data** output port are not valid.

Dependencies

To enable this port, set the **Operation mode** parameter to `Continuous`.

Data Types: `Boolean`

`ctrl` — Control signals accompanying sample stream
`samplecontrol` bus

Control signals accompanying the sample stream, returned as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates start of output frame.
- `end` — Indicates end of output frame.
- `valid` — Indicates the data on the output **data** port is valid.

Dependencies

To enable this port, set the **Operation mode** parameter to `Truncated` or `Terminated`.

Data Types: `bus`

`FSt` — Final state of frame at every frame end
scalar

Final state of frame at every frame end, returned as `fixdt(0,constraint length -1,0)`. Final state is the number of binary bits in the shift register at the frame end of the block, which is read from most significant bit (MSB) to least significant bit (LSB).

The block returns the same as the **ISt** data type.

Dependencies

To enable this port, set the **Operation mode** parameter to `Truncated` and select the **Enable final state output port** parameter.

Data Types: `single` | `double` | `fixed point`

Parameters

Main

`Constraint length` — Constraint length of block

`7` (default) | integer in the range [3, 9]

Constraint length of the block, specified as an integer in the range [3, 9].

`Code generator` — Code generation polynomial

`[171 133]` (default) | vector of octal values

Code generation polynomial, specified as a 1-by-*n* vector of octal values, where *n* is the length of the polynomial ranged from 2 to 7.

`Feedback connection` — Feedback connection polynomial

`0` (default) | scalar octal number

Feedback polynomial, specified as a scalar octal number. If the feedback connection is `0`, there is no feedback connection enabled.

To enable feedback connection, specify an octal value whose binary representation must be a K-bit number with MSB `1`, where *K* is the **Constraint length**. For more information on how to construct a feedback polynomial, refer to “Convolutional Codes”.

`Operation mode` — Mode of operation

`Continuous` (default) | `Terminated` | `Truncated`

Mode of operation, specified as one of these modes:

- `Continuous` — In this mode, the block starts with all zeros state and retains the encoder states at the end of each input, for use with the next input.
- `Terminated` — In this mode, the block considers each input frame independently. The encoder states of the block are reset to all-zeros state at the end of each frame by appending bits.

  Note This mode requires a minimum frame gap of **Constraint length** – 1 cycles between frames. If no sufficient frame gap is provided, the block stops processing the old frame and starts processing a new frame.
- `Truncated` — In this mode, the block considers each input frame independently. The encoder states are reset to all-zeros state at the start of each input.

Control Ports

`Enable reset input port` — Reset input signal

`off` (default) | `on`

Select this parameter to enable the **reset** port. When **reset** is 1 (`true`), the block resets the encoder state in the next clock cycle.

Dependencies

To enable this parameter, set the **Operation mode** parameter to `Continuous`.

`Enable initial state input port` — Initial state input signal
`Off` (default) | `On`

Select this parameter to enable the **ISt** port.

Dependencies

To enable this parameter, set the **Operation mode** parameter to `Truncated`.

`Enable final state output port` — Final state output signal
`Off` (default) | `On`

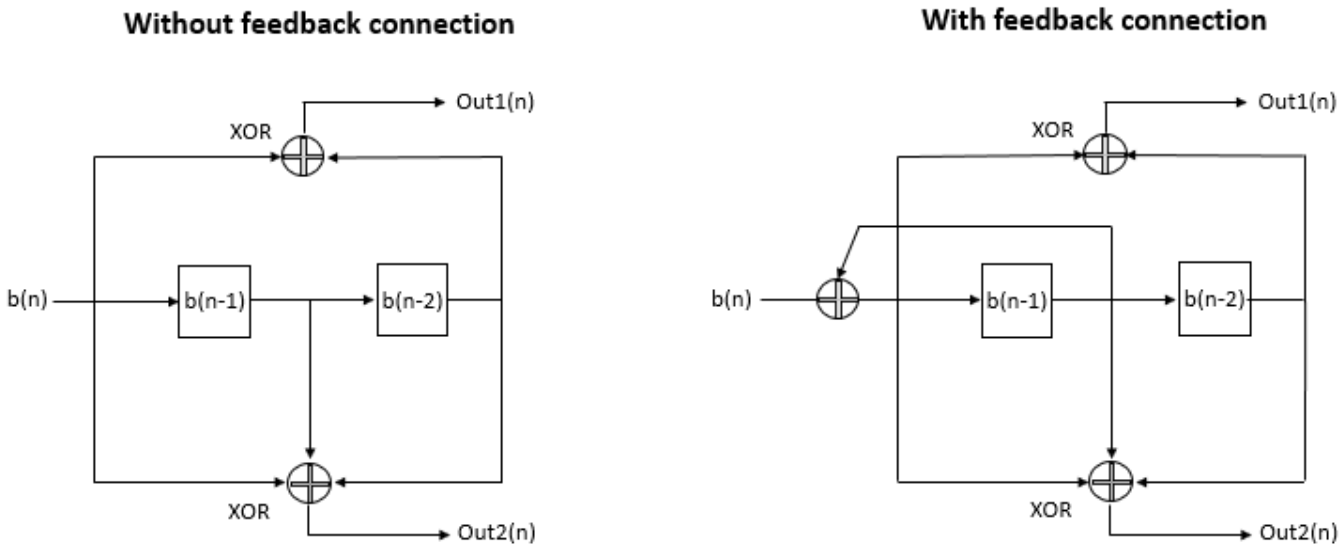
Select this parameter to enable the **FSt** port.

Dependencies

To enable this parameter, set the **Operation mode** parameter to `Truncated`.

Algorithms

A polynomial description of a Convolutional Encoder block describes the connections among shift registers and modulo 2 adders. This figure shows two sample encoding operations, one without feedback that has one input, two outputs, and two shift registers and the other with feedback that has one input, two outputs, and two shift registers.



b(n) represents input data bit stream and *b(n-1)* and *b(n-2)* represent the 2-bit shift register of the encoder. *Out1(n)* and *Out2(n)* represent the 2-bit output. From this figure, you can calculate the block mask parameters based on the Convolutional codes concepts. For more information about

Convolution codes concepts, refer to “Convolutional Codes”. So, based on the connections provided in the figure, the **Constraint length** is 3, **Code generator** value is `[5 7]`. The **Feedback connection** value is `0` for the encoder without feedback connection and `6` for the encoder with feedback connection.

Performance

The performance of the synthesized HDL code varies with your target and synthesis options. This table shows the resource and performance data synthesis results of the block with the default configuration parameters, when operated in `Terminated` mode. The generated HDL is targeted to Xilinx Zynq XC7Z045-FFG900-2 FPGA. The design achieves a clock frequency of 1223 MHz.

Resource	Number Used
Slice LUTs	13
Slice registers	25
Block RAM	0
DSPs	0

References

[1] Lin, Shu, and Daniel J. Costello. *Error Control Coding By Shu Lin, Daniel J. Costello, Second Edition*. Upper Saddle River, NJ: Prentice Hall, 2004.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).

OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

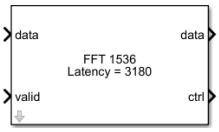
See Also

Blocks
Convolutional Encoder | Puncturer

Introduced in R2019b

FFT 1536

Computes fast-fourier-transform (FFT) for LTE standard transmission bandwidth of 15 MHz
Library: Wireless HDL Toolbox / Modulation



Description

The FFT 1536 block is designed to support LTE standard transmission bandwidth of 15 MHz. This block is used in LTE OFDM Demodulator block operation. The block accepts input data, along with a valid control signal and outputs streaming data with a `samplecontrol` bus.

The block provides an architecture suitable for HDL code generation and hardware deployment.

Ports

Input

`data` — Input data
scalar of real or complex values

Input data, specified as a scalar of real or complex values.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

The more the fractional bits you provide in the input word length, the better the accuracy you receive in the output.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `fixed point`

`valid` — Indicates valid input data
scalar

Indicates if the input data is valid. When the input **valid** is `1` (true), the block captures the value on the input **data** port. When the input **valid** is `0` (false), the block ignores the input **data** samples.

Data Types: `Boolean`

`reset` — Reset control signal
scalar

When this value is `1` (true), the block stops the current calculation and clears all internal states.

Dependencies

To enable this port, select the **Enable reset input port** parameter.

Data Types: `Boolean`

Output

`data` — Frequency channel output data

scalar of real or complex values

Frequency channel output data, returned as a scalar of real or complex values.

When the input is of `fixed point` data type, the output data type is the same as the input data type. When the input is of integer type, the output data type is of `fixed point` type.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `fixed point`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, returned as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the output frame
- `end` — Indicates the end of the output frame
- `valid` — Indicates that the data on the output **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

Parameters

Main

`Complex multiplication` — HDL implementation

`Use 3 multipliers and 5 adders` (default) | `Use 4 multipliers and 2 adders`

Specifies the complex multiplier type for HDL implementation. Each multiplication is implemented either with `Use 3 multipliers and 5 adders` or with `Use 4 multipliers and 2 adders`. The implementation speed depends on the synthesis tool and the target device that you use.

`Rounding method` — Rounding mode for internal fixed-point calculations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Zero`

Specifies the type of rounding method for internal fixed-point calculations. For more information about rounding methods, see “Rounding Modes” (DSP System Toolbox). When the input is any integer or fixed-point data type, this block uses fixed-point arithmetic for internal calculations. This parameter does not apply when the input data is `single` or `double`.

`Normalize butterfly output` — Output normalization

`off` (default) | `on`

When you select this parameter, the block divides the output by 1536. This option is useful when you want the output of the block to stay in the same amplitude range as its input. You require this option when the input is of `fixed point` type.

When you select this parameter, the output word length increases by 2 bits and when you clear this parameter the output word length increases by 11 bits.

Control Ports

`Enable reset input port` — Optional reset signal
`off` (default) | `on`

Select this parameter to enable the **reset** port.

Algorithms

To design an FFT 1536 block, radix-3 decimation-in-time (DIT) algorithm is implemented. The input sequence *x(n)* for all *n = {0,1,2....1535}* is divided into three DIT sequences, *x(3n), x(3n+1), x(3n+2)* for all *n = {0,1,2....511}*.

This equation defines FFT 1536 computation of a given sequence *x(n)*.

$$x(k) = \sum_{n=0}^{1535} x(n) W_{1536}^{nk}; k = 0, 1, 2, ..., 1535$$

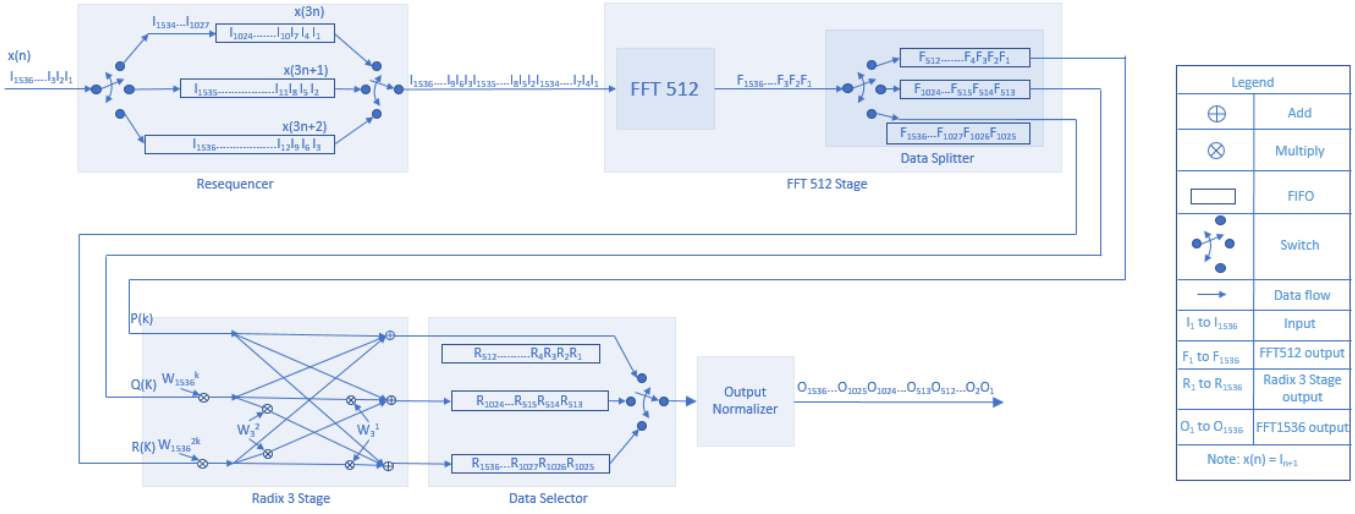
The equation can be implemented by dividing it into three parts, where *P(k), Q(k), R(k)* are the N/3 (FFT 512) point FFT of *x(3n), x(3n+1), and x(3n+2)*, respectively. Here, *N = 1536, and k = 0,1,2,.....,511*.

$$x(k) = P(k) + W_N^k Q(k) + W_N^{2k} R(k)$$

$$x(k + N/3) = P(k) + W_3^1 W_N^k Q(k) + W_3^2 W_N^{2k} R(k)$$

$$x(k + 2N/3) = P(k) + W_3^2 W_N^k Q(k) + W_3^1 W_N^{2k} R(k)$$

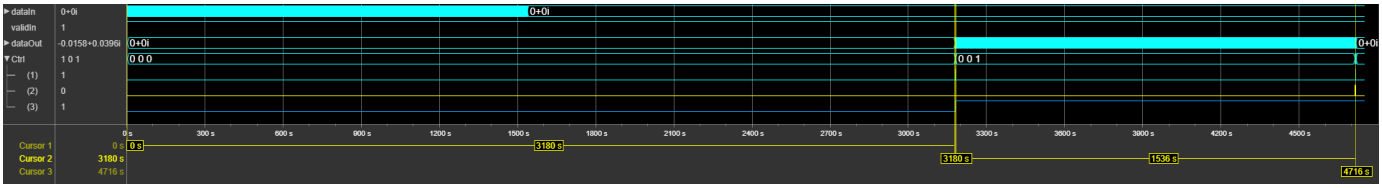
This diagram shows the internal architecture of the block and how the input sequence streams through the components of the block.



The input sequence *x(n)* is demultiplexed into three DIT sequences, *x(3n), x(3n+1), x(3n+2)*, each of length 512. Three first-input first-output (FIFO) memories store these sequences. These DIT sequences are serialized and streamed through the **FFT 512** block.

Latency

This image shows the output waveform of the block when operated with default configuration parameters. The block provides output data after a latency of 3180 clock cycles. The length of the output data between `start` (**Ctrl.(1)**) and `end` (**Ctrl.(2)**) output control signals is 1536 clock cycles.



Performance

The performance of the synthesized HDL code varies with your target and synthesis options. This table shows the resource and performance data synthesis results of the block with default configuration parameters, along with normalization feature enabled, and with an input data in `fixdt(1,17,15)` format. The generated HDL is targeted to Xilinx Zynq XC7Z045-FFG900-2 FPGA board. The design achieves a clock frequency of 355 MHz.

Resource	Number Used
LUTs	7330
Registers	9325
Block RAMs	18
DSPs	36

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).

InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

See Also

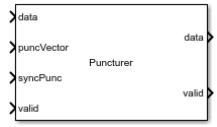
Blocks
FFT HDL Optimized

Introduced in R2019b

Puncturer

Punctures data according to puncture vector

Library: Wireless HDL Toolbox / Error Detection and Correction



Description

The Puncturer block punctures input data based on a specified puncture vector. The block accepts puncture vector either from the `Input port` or from the `Property` of the block and supports encoder rates from 1/2 to 1/7. It provides an architecture suitable for HDL code generation and hardware deployment.

The block supports `Continuous` and `Frame` mode operations and accepts both scalar and vector data. In `Continuous` mode, the block accepts input data and puncture vector, along with control signals valid and syncPunc and outputs punctured data with a valid signal. In `Frame` mode, the block accepts input data and puncture vector, along with a `samplecontrol` bus and outputs punctured data with a `samplecontrol` bus.

The block supports communication standards such as Wi-Fi (802.11a/b/g/n/ac), digital satellite communications, digital video broadcast (DVB), WiFi (IEEE 802.11a/b/g/n/ac), WiMax (IEEE 802.16), IEEE 802.16, HIPERLAN, and HiperMAN.

Ports

Input

`data` — Input data sample

scalar | column vector of size from 2 to 7

Input data sample, specified as a scalar or vector.

If input is of vector type, the size of the input data must match with the selected **Encoder rate** parameter value.

For example, if the **Encoder rate** is `1/2`, the input data size must be 2-by-1.

Data Types: `Boolean` | `fixdt(0,1,0)`

`valid` — Indicates valid input data samples

scalar

Control signal that indicates if the input data is valid. When this value is `1` (true), the block accepts the values on the **data** input port. When this value is `0` (false), the block ignores the values on the **data** input port.

Dependencies

To enable this port, set the **Operation mode** parameter to `Continuous`.

Data Types: `Boolean`

`puncVector` — Puncture vector

column vector of binary values

Puncture vector, specified as a column vector of binary values. The length of the puncture vector must be an integral multiple of *n*, where **Encoder rate** is *1/n*. For encoder rates 1/2, 1/3, 1/5, and 1/6, the maximum length of the **puncVector** is 30 and for encoder rates 1/4 and 1/7, the maximum length of the **puncVector** is 28.

You can change the **puncVector** pattern, but its length must remain constant. If the maximum puncture vector length provided is 10, the block supports all the vector lengths below 10.

Example: For an encoder rate 1/2 and its puncture rates 2/3, 3/4, and 5/6, the respective vector lengths are 4, 6, and 10. To achieve these multiple rates, set the **Puncture vector source** parameter to `Input port`. To support the largest vector size, the vector length must be 10 for all rates. For 2/3 and 3/4 rates, pad the **puncVector** input with zeros to create a 10-element vector. The puncture vector for rate 3/4 is `[1 1 0 1 1 0]'`. For a vector length of 10, use `[0 0 0 0 1 1 0 1 1 0]'` as the input **puncVector**.

When the **Operation mode** parameter is set to `Continuous`, the block captures the value of **puncVector** when both **syncPunc** and input **valid** port signals are `1` (true).

When the **Operation mode** parameter is set to `Frame`, the block captures the value of **puncVector** when both `ctrl.start` and `ctrl.valid` signals are `1` (true).

Dependencies

To enable this port, set the **Puncture vector source** parameter to `Input port`.

Data Types: `Boolean`

`syncPunc` — Puncture synchronization signal

scalar

Puncture synchronization signal, specified as a `Boolean` scalar value. This input is a control signal that synchronizes the puncture vector input with the input sample. When both **syncPunc** and **valid** are `1` (true), the block aligns the puncture vector to begin puncturing. The block captures the vector from either the **puncVector** input port or from the **Puncture vector** parameter.

The block ignores the **puncVector** signal values when **syncPunc** signal value is `0` (`false`).

Dependencies

To enable this port, set the **Operation mode** parameter to `Continuous`.

Data Types: `Boolean`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, specified as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the input samples.

- `start` — Indicates start of input frame.

- `end` — Indicates end of input frame.
- `valid` — Indicates that the data on the input **data** port is valid.

Dependencies

To enable this port, set the **Operation mode** parameter to `Frame`. In this mode, the block synchronizes the puncture vector using control signals in the input `samplecontrol` bus.

Data Types: `bus`

Output

`data` — Punctured output data
n-by-1 column vector

Punctured output data, returned as an n-by-1 column vector, where *n* value ranges from 1 to 7.

Data Types: `Boolean` | `fixdt(0,1,0)`

`valid` — Validity of output data samples
scalar

Control signal that indicates when the sample from the **data** output port is valid. The block sets the **valid** port to `1` (true) when there is a valid sample on the output **data** port.

Dependencies

To enable this port, set the **Operation mode** parameter to `Continuous`.

Data Types: `Boolean`

`ctrl` — Control signals accompanying sample stream
`samplecontrol` bus

Control signals accompanying the sample stream, returned as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates start of output frame.
- `end` — Indicates end of output frame.
- `valid` — Indicates that the data on the output **data** port is valid.

Dependencies

To enable this port, set the **Operation mode** parameter to `Frame`.

Data Types: `bus`

Parameters

`Operation mode` — Mode of operation
`Continuous` (default) | `Frame`

Mode of operation, specified as one of these modes:

- `Continuous` — Allows changes to **puncVector** at any time. To force the block to capture the new puncture vector, set the **syncPunc** parameter to `1` (true).

- `Frame` — Allows changes to **puncVector** only at the start of a frame, indicated by `ctrl.start`.

`Encoder rate` — Rate of encoder

`1/2` (default) | range from 1/2 to 1/7

Select the encoder rate for puncturing the data.

`Puncture vector source` — Source of puncture vector

`Input port` (default) | `Property`

Source of puncture vector, specified as:

- `Input port` — Specify the puncture vector using the **puncVector** port.
- `Property` — Specify the puncture vector using the **Puncture vector** parameter.

`Puncture vector` — Location of data to be punctured

`[1;1;1;0;0;1]` (default) | column vector of binary values

The length of the puncture vector must be an integral multiple of *n*, where **Encoder rate** is *1/n*. For encoder rates 1/2, 1/3, 1/5, and 1/6, the maximum length of the puncture vector is 30 and for encoder rates 1/4 and 1/7 the maximum length of the puncture vector is 28.

Dependencies

To enable this port, set the **Puncture vector source** parameter to `Property`.

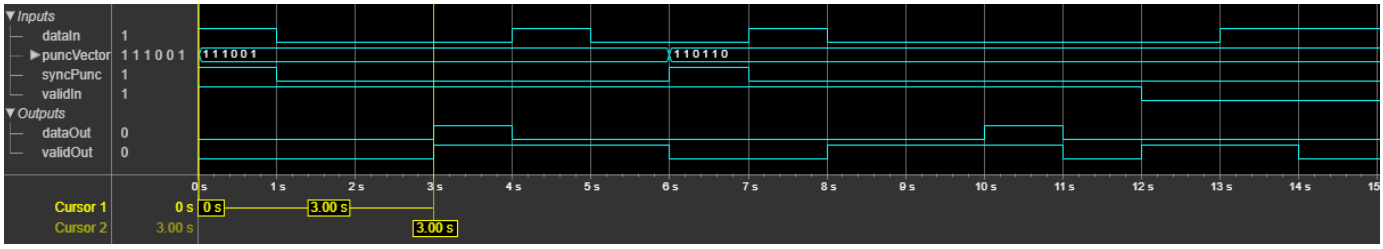
Algorithms

The puncturing algorithm checks every *n* elements of a puncture vector, with an **Encoder rate** *1/n*, until it reaches a nonzero combination. Then, it punctures the input data and provides the punctured output data.

For example, if the **Encoder rate** is `1/3` and the puncture vector is `[0;0;0;1;0;1]`, the block checks every 3 elements until it reaches a nonzero combination in the puncture vector and then punctures the input data based on the type of inputs (scalar or vector) and operation modes (`Continuous` or `Frame`).

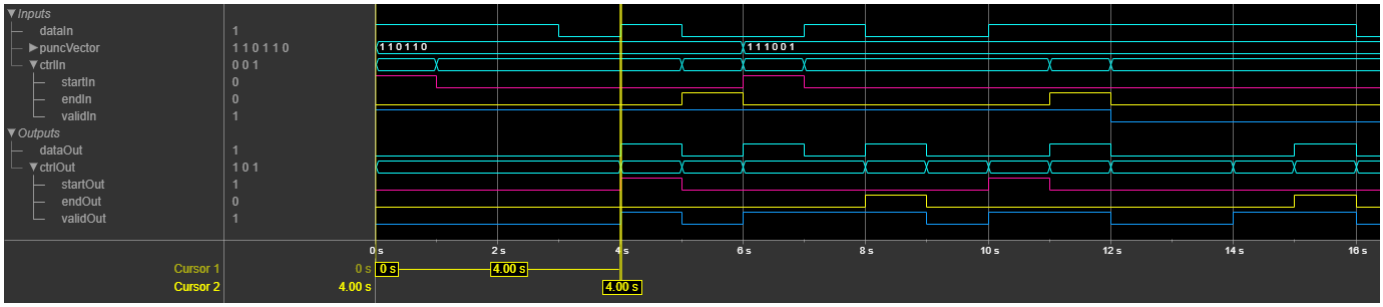
Scalar Input

- `Continuous` mode — When the puncture vector element is `0`, the block punctures the input data and provides no output. When the puncture vector element is `1`, the block provides the corresponding input data as output.



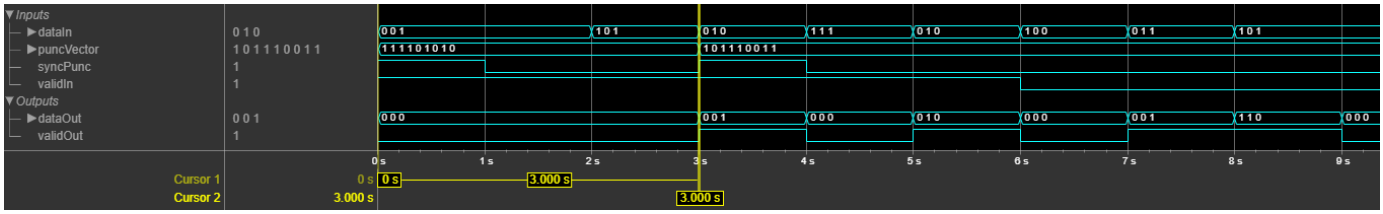
- `Frame` mode — When the puncture vector element is `1`, the block stores the corresponding input data in a buffer. It waits till it encounters the next `1` in the puncture vector and then provides the previous buffered data as output.

When the puncture vector element is `0`, the block punctures the input data and provides no output. But, if the **endIn** signal is `1` (high), the block provides the previous buffered data as output. The block repeats the similar process throughout the frame.

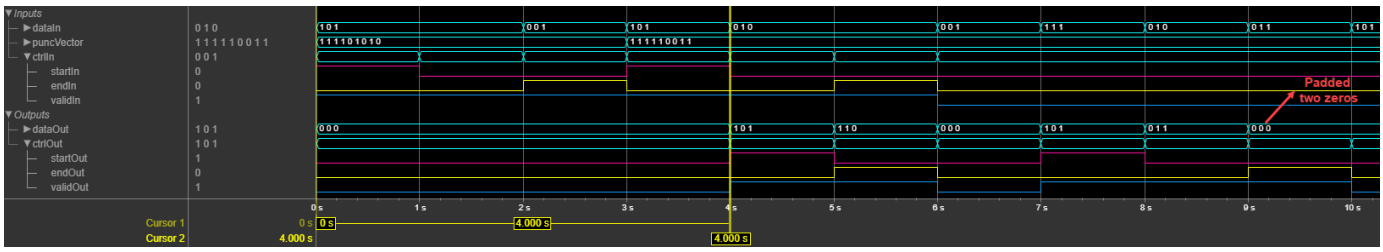


Vector Input

- `Continuous` mode — For a 3-by-1 vector input data with **Encoder rate** `1/3`, the block selects 3 elements of the puncture vector at a time. When the puncture vector element is `0`, the block punctures the data and provides no output. When the puncture vector element is `1`, the block stores the corresponding input data. The block provides the output only when the stored data count reaches 3.



- `Frame` mode — The block behaves similarly as when in `Continuous` mode. But, when the **endIn** signal is `1` (high) and the stored data count is less than 3, the block pads zeros and then outputs the data.



Latency

The latency of the block varies with the puncture vector and encoder rate. The above waveforms show the latency of the block for a sample scalar and vector input data with different puncture vectors.

Performance

These resource and performance data are the synthesis results from the generated HDL targeted to a XilinxZynq-7000 ZC706 board. The block is using `Boolean` input samples, in continuous mode with default settings with puncture vector length 6. The design achieves a clock frequency of 559 MHz.

Resource	Number Used
LUT	50
FFS	40

If you set the **Puncture vector source** parameter to `Property`, the design uses fewer LUT and FFS resources with more frequency. The hardware resources and frequencies vary based on the encoder rate and the puncture vector size.

References

[1] *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.* IEEE Std 802.11™- 2016 Part 11.

[2] EN 300 421 V1.1.2 *Digital Video Broadcasting (DVB); Framing structure, Channel coding and modulation for 11/12 GHz satellite services*.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

This block does not have any HDL Block Properties.

See Also

Blocks

Convolutional Encoder | Depuncturer

Introduced in R2019b

OFDM Demodulator

Demodulate time-domain OFDM samples and return subcarriers for custom communication protocols
Library: Wireless HDL Toolbox / Modulation

Description

The OFDM Demodulator block demodulates time-domain orthogonal frequency division multiplexing (OFDM) samples and outputs subcarriers based on the OFDM parameters. The block supports 5G new radio (NR) standard, long term evolution (LTE) [1], wireless local area network (WLAN 802.11a/g/n/ac) [2], WiMAX, digital video broadcast (DVB), and digital audio broadcast (DAB) standards.

The block accepts input data along with a valid control signal and these OFDM parameters: FFT length, CP length, and the number of right and left guard subcarriers. The block outputs demodulated data along with valid and ready controls signals. The block enables the **ready** output port only when these OFDM parameters are provided to the block through input ports. The block samples the corresponding OFDM parameters only when the **ready** port is `1` (high) and the first **valid** port of each OFDM symbol is `1` (high).

The block supports scalar and vector inputs. You can use a vector input to increase the data throughput and achieve a giga-sample-per-second (GSPS) throughput. The block provides an interface and architecture suitable for HDL code generation and hardware deployment.

Ports

Input

`data` — Input data

scalar | column vector

Input data, specified as a scalar or column vector of real or complex values. The vector size must be a power of 2, in the range from 1 to 64, and less than or equal to the FFT length.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `signed fixed point`

`valid` — Indicates valid input data

scalar

Indicates valid input data, specified as a scalar.

This port is a control signal that indicates when the sample from the **data** input port is valid. When this value is `1`, the block captures the values on the **data** input port. When this value is `0`, the block ignores the values on the **data** input port.

Data Types: `Boolean`

FFTLen — Length of FFT

scalar

Length of the FFT, specified as a scalar. The FFT length must be power of 2 and in the range from 8 to 65,536. This value must be less than or equal to the **Maximum FFT length** parameter value.

To support the minimum FFT length of 8, the **FFTLen** data type must be `fixdt(0,k,0)`, where *k* is greater than or equal to 4.

Dependencies

To enable this port, set the **OFDM parameters source** parameter to `Input port`.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `unsigned fixed point`

CPLen — Length of cyclic prefix

scalar

Length of the cyclic prefix, specified as a scalar in the range from 0 to **FFTLen**.

To support the minimum FFT length of 8, the **CPLen** data type must be `fixdt(0,k,0)`, where *k* is greater than or equal to 4.

Dependencies

To enable this port, set the **OFDM parameters source** parameter to `Input port`.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `unsigned fixed point`

numLgSc — Number of left guard carriers of OFDM symbol

scalar

Number of left guard carriers of OFDM symbol, specified as a scalar in the range from 0 to (**FFTLen**/2) – 1.

To support the minimum FFT length of 8, the **numLgSc** data type must be `fixdt(0,k,0)`, where *k* is greater than or equal to 2.

Dependencies

To enable this port, set the **OFDM parameters source** parameter to `Input port`.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `unsigned fixed point`

numRgSc — Number of right guard carriers of OFDM symbol

scalar

Number of right guard carriers of OFDM symbol, specified as a scalar in the range from 0 to (**FFTLen**/2) – 1.

To support the minimum FFT length of 8, the **numRgSc** data type must be `fixdt(0,k,0)`, where *k* is greater than or equal to 2.

Dependencies

To enable this port, set the **OFDM parameters source** parameter to `Input port`.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `unsigned fixed point`

`reset` — Clear internal states

scalar

Clear internal states, specified as a scalar. When this value is `1` (true), the block stops the current calculation and clears all internal states.

Dependencies

To enable this port, select the **Enable reset input port** parameter.

Data Types: `Boolean`

Output

`data` — Demodulated output data

scalar | column vector

Demodulated output data, returned as a complex-valued scalar or column vector. Output data type is dependent on the data type of the input **data** port.

- When you set the **OFDM parameters source** parameter to `Property` and clear the **Divide butterfly outputs by two** parameter, the output word length increases by $\log_2$(**FFT length**) bits.
- When you set the **OFDM parameters source** parameter to `Input port` and clear the **Divide butterfly outputs by two** parameter, the output word length increases by $\log_2$(**Maximum FFT length**) bits.

To avoid overflow, select the **Divide butterfly outputs by two** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `signed fixed point`

`valid` — Indicates valid output data

scalar

Indicates valid input data, returned as a scalar.

This port is a control signal that indicates when the **data** output port is valid. The block sets this value to `1` when the data samples are available on the **data** output port. When you select the **Remove DC subcarrier** parameter, this value is set to `0` at the center of the output samples to exclude the DC carrier.

Data Types: `Boolean`

`ready` — Indicates block is ready

scalar

Control signal that indicates when the block is ready for new input data. When this value is `1`, the block accepts input data in the next time step. When this value is `0`, the block ignores input data in the next time step.

Dependencies

To enable this port, set the **OFDM parameters source** parameter to `Input port`.

Data Types: `Boolean`

Parameters

Main

`OFDM parameters source` — Source of OFDM parameters
`Property` (default) | `Input port`

You can set OFDM parameters with an input port or by selecting a value for the parameter.

Select `Property` to enable the **FFT length**, **Cyclic prefix length**, **Number of left guard subcarriers**, and **Number of right guard subcarriers** parameters.

Select `Input port` to enable the **FFTLen**, **CPLen**, **numLgSc**, **numRgSc** input ports and the **Maximum FFT length** parameter. The **Maximum FFT length** parameter sets the upper bound of the range of valid values for the **FFTLen** input port.

`Maximum FFT length` — Maximum length of FFT length
64 (default) | power of 2 in range from 8 to 65,536

Specify the maximum length of the FFT.

Dependencies

To enable this parameter, set the **OFDM parameters source** parameter to `Input port`.

`FFT length` — Length of FFT
64 (default) | power of 2 in range from 8 to 65,536

Specify the FFT length. When you set the **OFDM parameters source** parameter to `Property`, the block uses this FFT length value as the maximum FFT length.

Dependencies

To enable this parameter, set the **OFDM parameters source** parameter to `Property`.

`Cyclic prefix length` — Length of cyclic prefix
16 (default) | integer in range from 0 to **FFT length**

Specify the length of the cyclic prefix.

Dependencies

To enable this parameter, set the **OFDM parameters source** parameter to `Property`.

`Number of left guard subcarriers` — Number of guard band subcarriers in left extreme of OFDM symbol
6 (default) | integer in range from 0 to (**FFT length**/2) – 1

Specify the number of left guard subcarriers.

Dependencies

To enable this parameter, set the **OFDM parameters source** parameter to `Property`.

`Number of right guard subcarriers` — Number of guard band subcarriers in right extreme of OFDM symbol
5 (default) | integer in range from 0 to (**FFT length**/2) – 1

Specify the number of right guard subcarriers.

Dependencies

To enable this parameter, set the **OFDM parameters source** parameter to `Property`.

`Enable CP Fraction` — CP fraction enabler
`off` (default) | `on`

Select this parameter to enable the **CP Fraction** parameter on the block mask.

`CP Fraction` — Percent of cyclic prefix to remove
`0.55` (default) | range from 0 to 1

Cyclic prefix fraction, specified as a value from 0 to 1, inclusive. This parameter specifies the percentage of CP samples that the block removes from the start of the OFDM symbol. The block shifts the remaining CP samples to the end of the OFDM symbol.

When this parameter is `0.55`, the block removes 55% of the CP from the beginning of the symbol, and shifts 45% to the end of the symbol. When you set this parameter to `1`, the block removes 100% of the CP from the start of the OFDM symbol, and does not shift any samples to the end.

Dependencies

To enable this parameter, select the **Enable CP Fraction** parameter.

`Remove DC subcarrier` — Exclude or include DC subcarrier
`on` (default) | `off`

When you select this parameter, the block excludes the DC subcarrier in the output by setting the output valid signal to `0` for the center of the output subcarriers.

`Enable reset input port` — Reset signal
`off` (default) | `on`

Select this parameter to enable the **reset** input port.

FFT Parameters

`Divide butterfly outputs by two` — Divide FFT butterfly outputs by two
`off` (default) | `on`

This parameter controls the scaling option of the FFT HDL Optimized block inside the OFDM Demodulator block.

When you select this parameter, the FFT implements an overall 1/*N* scale factor by dividing the output of each butterfly multiplication by two. This adjustment keeps the output of the FFT in the same amplitude range as its input. If you clear this parameter, the block avoids overflow by increasing the word length by one bit after each butterfly multiplication.

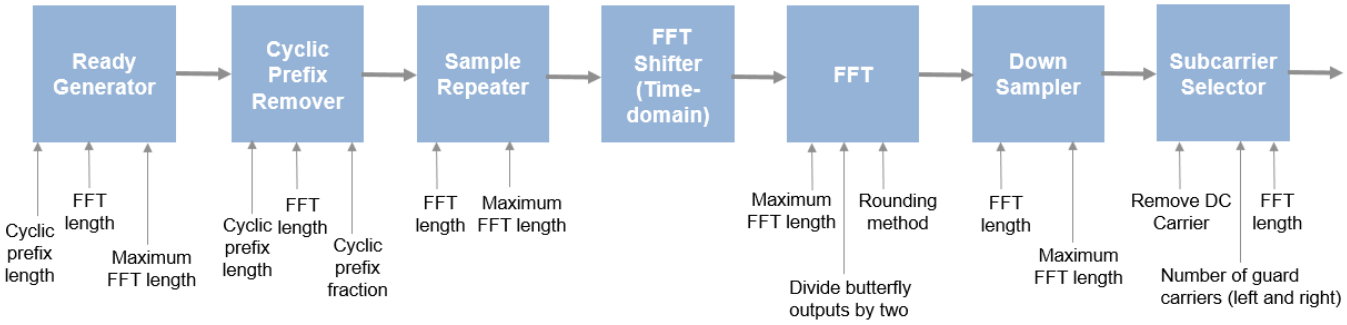
`Rounding Method` — Rounding mode for internal fixed-point calculations
`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Zero`

This parameter specifies the type of rounding mode for internal fixed-point calculations. For more information about rounding modes, see Rounding Modes (DSP System Toolbox). When the input is any integer data type or fixed-point data type, the FFT algorithm uses fixed-point arithmetic for

internal calculations. This parameter does not apply when the input is of data type `single` or `double`. Rounding applies to twiddle-factor multiplication and scaling operations.

Algorithms

The OFDM Demodulator block operation sequence is implemented using these blocks: Ready Generator, Cyclic Prefix Remover, Sample Repeater, FFT Shifter, FFT, Down Sampler, and Subcarrier Selector. The parameters shown in this figure configure the behavior of the block.



Ready Generator

This block enables a **ready** port when you set the **OFDM parameters source** parameter to `Input port`. This **ready** port controls the input samples based on the maximum FFT length.

The following equations apply.

- N_h = `ceil`((N_r + **FFTLen** + **CPLen**)/*vecLen*)
- N_l = `ceil`((N_r + **Maximum FFT length** + **CPLen**)/*vecLen*) – N_h

In these equations,

- N_h is the number of high ready clock cycles
- N_l is the number of low ready clock cycles
- N_r is the number of remaining samples from the previous OFDM symbol. Initially, this value is `0`. In the subsequent operations, the block calculates N_r using the equation, (N_r + **FFTLen** + **CPLen**) - (`floor`((N_r + **FFTLen** + **CPLen**) / *vecLen*) x *vecLen*)
- *vecLen* is the length of the vector

Cyclic Prefix Remover

This block removes CP samples from an OFDM symbol for extracting constellation symbols. The block performs CP removal based on these parameters: **CP length**, **CP fraction** (when enabled), and the **FFT length**.

This block supports windowed transmission by implementing fractional cyclic prefix removal. Windowing reduces out-of-band emissions. A transmitter performs windowing by overlapping the tail of each OFDM symbol with the head of the next OFDM symbol. A receiver must avoid these overlapped samples in the FFT calculation. Fractional CP solves this problem by removing part of the CP at the start of a symbol and the remainder of the CP at the end of the symbol. Implementing a CP-fraction algorithm also makes this block less sensitive to timing offset.

The block handles the CP in two stages. First, the block calculates the number of CP samples to remove, N_r, and removes those samples from the input samples. In this case, N_r = *CP fraction* x *CP length*.

Next, the block calculates the number of samples to shift, N_s, and shifts those samples to the end of OFDM symbol in the time domain. Where, N_s = *CP length* – (*CP fraction* x *CP length*).

These two segments together make up the total cyclic prefix length, $N_{cp} = N_s + N_r$. The **CP fraction** parameter controls how many samples the block removes at the beginning of the symbol. The block shifts the remainder of the cyclic prefix from the start of the symbol to the end of the symbol. The block quantizes the **CP fraction** parameter as `fi(0,11,10)`. To achieve an integer number of samples, the block calculates N_r = `floor` (N_{cp} x **CP fraction**).

For example, if the FFT length is 128 and CP length is 10, the block receives 128 samples plus the cyclic prefix size.

Sample Repeater

This block repeats FFT-length number of samples until it forms the maximum FFT length. For this operation, the block buffers the input samples first and then repeats the samples based on the maximum FFT length value. This repetition mechanism helps to avoid scaling at the FFT block input. This block is optional and available only when you set the **OFDM parameters source** parameter to `Input port`. When you set the **OFDM parameters source** parameter to `Property`, the FFT length value provided in the block mask is set as the maximum FFT length. The block does not need to repeat the samples in this context.

For example, if the FFT length is 128 and the maximum FFT length is 2048, each OFDM symbol consists of 128 samples. The block converts these 128 samples to 2048 samples by repeating the 128 samples 16 times. After the block generates 2048 data samples, it sends data and valid input signals to the next block.

Time-Domain FFT Shifter

Conventionally, receivers perform the FFT shift in the frequency domain. However, this method requires memory and introduces latency related to the size of the FFT. Instead, a receiver can execute the same operation in the time domain by using the frequency shifting property of Fourier transforms. Shifting a function in one domain corresponds to a multiplication by a complex exponential function in the other domain. To reduce hardware resources and latency, this block performs the FFT shift by multiplying the time-domain samples by a complex exponential function.

These equations describe an FFT shift. The equation for an N-point FFT is

$$X(k) = F[x(n)] = \sum_{n=0}^{N-1} x(n) e^{-\frac{j2\pi nk}{N}}$$

For an FFT shift of $N/2$ carriers in either direction, substitute $k = k - \frac{N}{2}$, resulting in

$$X(k - \tfrac{N}{2}) = \sum_{n=0}^{N-1} x(n) e^{-\frac{j2\pi n(k - \frac{N}{2})}{N}}$$

This equation simplifies to

$$X(k-\frac{N}{2}) = \sum_{n=0}^{N-1} e^{j\pi n} x(n) e^{-\frac{j2\pi nk}{N}}$$

Since $\sum_{n=0}^{N-1} x(n)e^{-\frac{j2\pi nk}{N}}$ is equivalent to $F[x(n)]$, and $e^{j\pi} = -1$, this equation simplifies to

$$X(k-\frac{N}{2}) = F[(-1)^n x(n)]$$

The final equation shows that an FFT shift in the time domain simplifies to multiplication by $(-1)^n$. As a result, the block implements the FFT shift by multiplying the time-domain samples by either +1 or –1.

FFT

This block converts a time-domain signal to a frequency-domain signal based on the maximum FFT length provided for the block. You can provide the FFT length value either through a parameter or through an input port. The output of the FFT shift subsystem is fed to an FFT HDL Optimized block. The block calculates the maximum FFT for all the FFT length and CP length values.

The **Divide butterfly outputs by two** parameter sets whether the FFT implements an overall 1/*N* scale factor by dividing the output of each butterfly multiplication by two. This adjustment keeps the output of the FFT in the same amplitude range as its input. When you clear the **Divide butterfly outputs by two** parameter, the block avoids overflow by increasing the word length by one bit after each butterfly multiplication.

Down Sampler

This block down samples maximum-FFT-length number of samples to FFT-length number of samples. This block is optional and available only when you set the **OFDM parameters source** parameter to `Input port`. When you set the **OFDM parameters source** parameter to `Property`, the FFT length value provided in the block mask sets the maximum FFT length. The block does not need to downsample the samples in this context.

For example, if the FFT length is 128 and the maximum FFT length is 2048, the input is 2048 samples and must be downsampled with respective to the FFT length of 128. In this case, the block samples 1 sample for every 16 samples.

Subcarrier Selector

The output subcarriers are categorized into data, DC, and guard subcarriers. Data subcarriers contains useful data. This block selects subcarriers by removing the number of left guard subcarriers and right guard subcarriers provided for the block. The number of guard subcarriers to set varies with standards.

If you select the **Remove DC subcarrier** parameter, the block excludes the DC subcarrier from output. The block excludes the DC subcarrier by setting the **valid** port to `0` (false) for the center cycle of the output subcarriers.

Latency

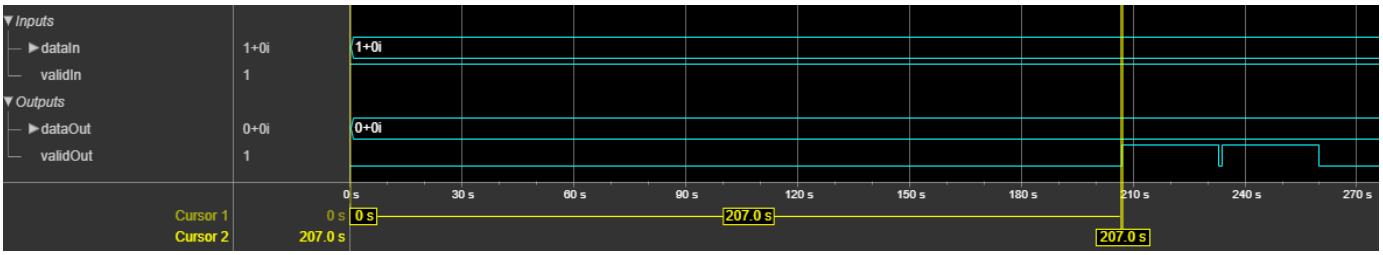
The block captures output data at valid cycles based on the type of input: scalar or vector.

Scalar Input

This figure shows a sample output and latency of the OFDM Demodulator block when you specify a scalar input, set the **OFDM parameters source** parameter to `Property` and use default settings for the other block parameters. In this example, the **FFTLen** parameter is set to 64, **Cyclic prefix length** parameter is set to 16, **Number of left guard subcarriers** parameter is set to 6, and **Number of right guard subcarriers** parameter is set to 5.

In this example, the latency of the block is calculated using this formula: **Cyclic prefix length** + *FFTLatency* + **Number of left guard subcarriers** + 12, where *FFTLatency* is the latency of FFT block for the specified FFT length, and 12 is the number of pipeline delays.

After calculation, the latency of the block is 207 clock cycles, as shown in the following figure.



This figure shows a sample output and latency of the block when you specify a scalar input and set the **OFDM parameters source** parameter to `Input port`. In this example, the **FFTLen** port is set to 64, **CPLen** port is set to 16, **numLgSc** port is set to 6 and **numRgSc** port is set to 5, and **Maximum FFT length** parameter is set to 128.

The latency of the block is calculated using the formula **CPLen** + **FFTLen** + *FFTLatency* + **numLgSc** x (**Maximum FFT length**/**FFTLen**) + 25, where *FFTLatency* is the latency of FFT block for the specified maximum FFT length, and 25 is the number of pipeline delays.

After calculation, the latency of the block is 424 clock cycles, as shown in this figure.

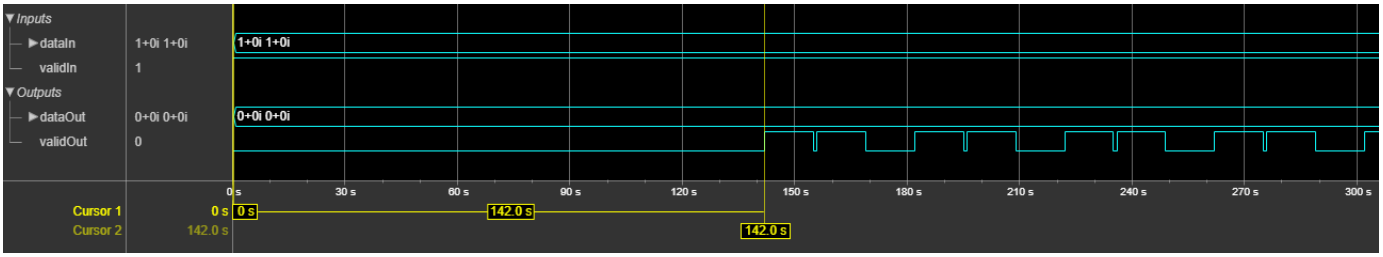
The block accepts input only when the **ready** is 1 (high). In this case, the block captures parameters on the first cycle when the input **valid** port is 1 (high).

Vector Input

This figure shows a sample output and latency of the OFDM Demodulator block when you specify a two-element column vector input and set the **OFDM parameters source** parameter to `Property` and use default settings for the other block parameters. **FFTLen** is set to 64, **Cyclic prefix length** is set to 16, and **Number of left guard subcarriers** and **Number of right guard subcarriers** are set to 6 and 5, respectively.

In this example, the latency of the block is calculated using this formula: `floor`(**Cyclic prefix length**/*vecLen*) + *vecFFTLatency* + `floor` (**Number of left guard subcarriers**/*vecLen*) + 12, where *vecFFTLatency* is the latency of FFT block for the specified FFT length and vector length, *vecLen* is the length of the vector, and 12 is the number of pipeline delays.

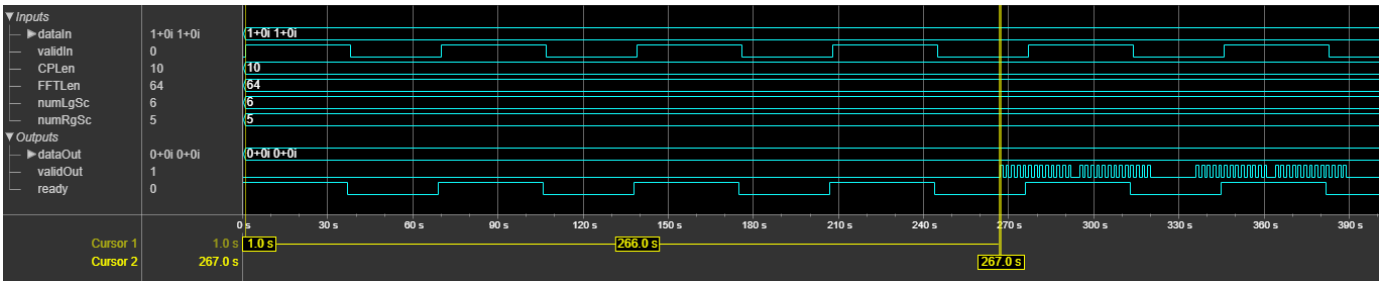
This calculation shows that the latency of the block is 142 clock cycles, as shown in this figure.



This figure shows a sample output and latency of the block when you specify a two-element column vector input and set the **OFDM parameters source** parameter to `Input port`. For this example, **FFTLen** is set to 64, **CPLen** is set to 16, **numLgSc** is set to 6, **numRgSc** is set to 5, and **Maximum FFT length** is set to 128.

In this example, the latency of the block is calculated using this formula: `floor`(**CPLen**/*vecLen*) + **FFTLen**/*vecLen* + *vecFFTLatency* + `floor`(**numLgSc**/*vecLen*) x (**Maximum FFT length**/**FFTLen**) + 26, where *vecFFTLatency* is the latency of FFT block for the specified maximum FFT length and vector length, *vecLen* is the length of the vector, and 26 is the number of pipeline delays.

After calculation, the latency of the block is 266 clock cycles, as shown in this figure.



The block accepts input only when the **ready** is 1 (high). In this case, the block captures parameters on the first cycle when the input **valid** port is 1 (high).

Performance

The performance of the synthesized HDL code varies with your target and synthesis options. The input data type used in this example for generating HDL code is `fixdt(1,16,14)`.

This table shows the resource and performance data synthesis results when using the block with a scalar or two-element column vector input for default configuration values. The generated HDL is targeted to the Xilinx Zynq- 7000 ZC706 evaluation board.

Input Data	Slice LUTs	Slice Registers	DSPs	Block RAM	Maximum Frequency in MHz
Scalar	2434	4161	8	1	340
Vector	4890	7764	16	0	235

References

[1] 3GPP TS 36.211 version 14.2.0 Release 14. "Physical channels and modulation." *LTE - Evolved Universal Terrestrial Radio Access (E-UTRA)*.

[2] "Wireless LAN Medium Access Control (MAC) and Physical layer (PHY) Specifications." IEEE Std 802.11 – 2012.

[3] Stefania Sesia, Issam Toufik, and Matthew baker. *LTE - THE UMTS Long Term Evolution from theory to practice*.

[4] Erik Dahlman, Stefan Parkvall, and Johan Skold. *4G - LTE/LTE - Advanced for Mobile broadband Second edition*.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

This block does not have any HDL Block Properties.

See Also

Blocks

OFDM Demodulator Baseband | OFDM Modulator

Objects

`comm.OFDMDemodulator`

Introduced in R2019b

RS Decoder

Decode and recover message from RS codeword

Library: Wireless HDL Toolbox / Error Detection and Correction

Description

The RS Decoder block decodes and recovers a message from a Reed-Solomon (RS) codeword. The block accepts codeword data and a `samplecontrol` bus and outputs a decoded message data, a `samplecontrol` bus, whether the received data is corrupted, a block ready indicator, and (optionally) the number of detected errors. The block provides an architecture suitable for HDL code generation and hardware deployment and supports shortened message lengths.

Because, the latency of the block varies, the block provides output port **nextFrame** that indicates when the block is ready to accept new input codeword data. For more details about latency, see the “Algorithms” on page 1-165 section.

You can use this block to model many communication system forward error correcting (FEC) codes. The block supports digital subscriber line (DSL), WiMAX (802.16 m and e), digital video broadcast handheld (DVB-H) terminals, digital video broadcast satellite (DVB-S) services, and digital video broadcast satellite services to handheld (DVB-SH) devices below 3 MHz.

Ports

Input

`data` — Input codeword data

scalar

Input codeword data, specified as a scalar representing one symbol.

The length of the codeword in symbols specified by the **Codeword length (N)** parameter must be an integer equal to 2^M – 1, where M is an integer in the range from 3 to 16.

The input word length must be an unsigned integer equal to `ceil`($\log_2$(**Codeword length (N)**)). For a codeword length of 7, the input data word length must be 3.

`double` and `single` data types are allowed for simulation, but not for HDL code generation.

Data Types: `double` | `single` | `uint8` | `uint16` | `fixed point`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, specified as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the input frame
- `end` — Indicates the end of the input frame
- `valid` — Indicates that the data on the input **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

Output

`data` — Decoded message data
scalar

Decoded message data, returned as a scalar. This output data width is the same as the input data width.

Data Types: `double` | `single` | `uint8` | `uint16` | `fixed point`

`ctrl` — Control signals accompanying sample stream
`samplecontrol` bus

Control signals accompanying the sample stream, returned as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the output frame
- `end` — Indicates the end of the output frame
- `valid` — Indicates that the data on the output **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

`err` — Indication of corruption of received data
scalar

Indication of corruption of the received data, returned as a scalar.

When this value is `1`, the message contains at least one error. When this value is `0`, the message contains zero errors.

Data Types: `Boolean`

`nextFrame` — Block ready indicator
scalar

Block ready indicator, returned as a scalar.

The block sets this signal to `1` (`true`) when the block is ready to accept the start of the next frame. If the block receives an input **ctrl.start** signal while **nextFrame** is `0` (`false`), the block discards the frame in progress and begins processing the new data.

Data Types: `Boolean`

`numErrors` — Number of detected errors
nonnegative scalar

Number of detected errors, returned as a nonnegative scalar.

Dependencies

To enable this port, select the **Output number of corrected symbol errors** parameter.

Data Types: `uint8`

Parameters

`Codeword length (N)` — Length of codeword

7 (default) | range from 7 to 65,535

Specify the codeword length.

The codeword length `N` must be an integer equal to $2^M - 1$, where M is an integer in the range from 3 to 16. For more information on representing data for RS codes, see "Integer Format (Reed-Solomon Only)".

`Message length (K)` — Length of message

3 (default) | integer in the range from 3 to (**Codeword length (N)** – 2)

Specify the message length.

For more information on representing data for RS codes, see "Integer Format (Reed-Solomon Only)".

`Source of primitive polynomial` — Primitive polynomial source

`Auto` (default) | `Property`

Specify the source of the primitive polynomial.

- Select `Auto` to specify the primitive polynomial based on the **Codeword length (N)** parameter value. The degree of the primitive polynomial is calculated as M = `ceil`($\log_2$(**Codeword length (N)**)).
- Select `Property` to specify the primitive polynomial using the **Primitive polynomial** parameter.

`Primitive polynomial` — Primitive polynomial

`[1 0 1 1]` (default) | binary row vector

Specify a binary row vector representing the primitive polynomial in descending order of powers.

For more information on how to specify a primitive polynomial, see "Primitive Polynomials and Element Representations".

Dependencies

To enable this parameter, set the **Source of primitive polynomial** parameter to `Property`.

`Source of B, the starting power for roots of the primitive polynomial` — Source of starting power for roots of primitive polynomial

`Auto` (default) | `Property`

Specify the source of the starting power for roots of the primitive polynomial.

- Select `Auto`, to use the default **B value** parameter value, 1.

- Select `Property` to enable the **B value** parameter.

`B value` — Starting power of roots
1 (default) | positive integer

Specify the starting power for roots of the primitive polynomial.

Dependencies

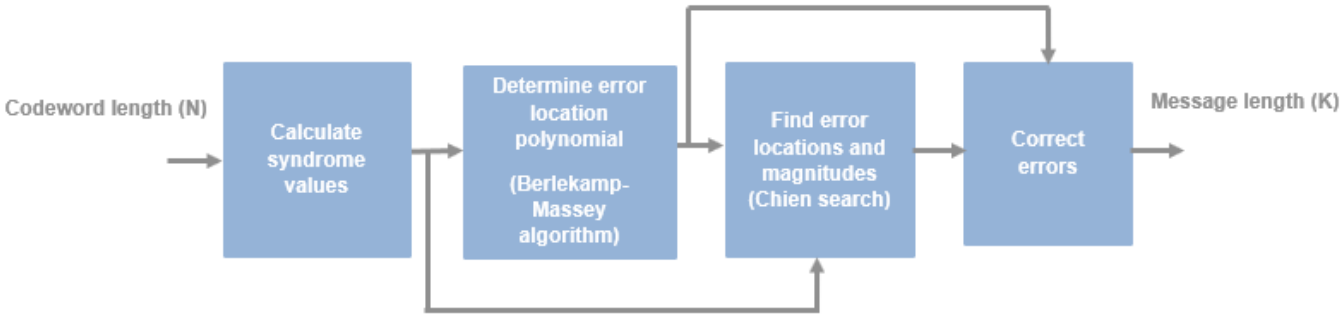
To enable this parameter, set the **Source of B, the starting power for roots of the primitive polynomial** parameter to `Property`.

`Output number of corrected symbol errors` — Number of corrected symbol errors
`off` (default) | `on`

Select this parameter to enable the **numErrors** output port. This port outputs the detected symbol error count.

Algorithms

This figure shows the different stages of operations performed by the RS Decoder block. The block calculates syndrome values, determines the error location polynomial, finds error locations and magnitudes, and corrects the errors.



This block uses the Berlekamp-Massey decoding algorithm. For information about this algorithm, see "Algorithms for BCH and RS Errors-only Decoding".

Latency

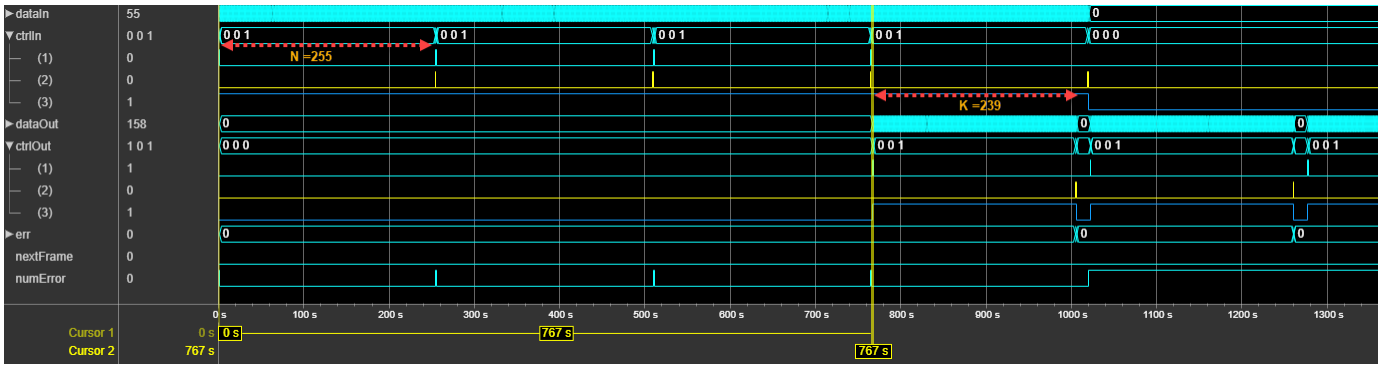
The latency between valid input data and the corresponding valid output data depends on the length of the codeword and the time the block takes to calculate error locating polynomials and find error locations and magnitudes. The time for which the **nextFrame** port value remains `0` depends on the processing time of the block. The processing time of the block is equal to the sum of the time the block takes to compute error locating polynomial (*ELPTime*) and find error locations and error magnitudes (*ConvTime*). The processing time is calculated as

$$\begin{aligned}\text{Processing\_time} &= ELPTime + ConvTime \\ &= 4 \times t + t(2t \times t + 1) + 10\ (Pipelining\ delays)\end{aligned}$$

t is the number of errors the RS code can correct and is equal to (**Codeword length (N)** – **Message length (K)**) /2.

The latency of the block is $2^{\text{ceil}(\log_2 Processing\_time)}$ + 2 x **Codeword length (N)**.

This figure shows a sample output of the RS Decoder block with latency according to the DVB-S standard configuration, and **Codeword length (N)** and **Message length (K)** parameter values specified as 255 and 239, respectively. In this case, when the processing time is less than the **Codeword length (N)**, the block provides support for a continuous input. The latency of the block is 768 clock cycles.



Performance

The performance of the synthesized HDL code varies with your target and synthesis options. The input data type used for generating HDL code is `fixdt(0,8,0)`.

This table shows the resource and performance data synthesis results when using the block with **Codeword length (N)** and **Message length (K)** parameter values specified as 255 and 239, respectively. The generated HDL is targeted to the Xilinx Zynq- 7000 ZC706 evaluation board. The design achieves a clock frequency of 161.4 MHz.

Resource	Number Used
LUTs	4042
Registers	2694
DSPs	0
Block RAMs	19.5

References

[1] Wicker, Stephen B. *Error Control Systems for Digital Communication and Storage*. Englewood Cliffs, NJ: Prentice Hall, 1995.

[2] Berlekamp, Elwyn R. *Algebraic Coding Theory*. Revised edition. McGraw-Hill Series in Systems Science. New Jersey: World Scientific, 2015.

[3] Clark, George C., and J. Bibb Cain. *Error-Correction Coding for Digital Communications*. Applications of Communications Theory. New York: Plenum Press, 1981.

[4] Moon, Todd K. *Chapter 6, Error Correction Coding: Mathematical Methods and Algorithms*. Hoboken, N.J: Wiley-Interscience, 2005.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

See Also

Blocks

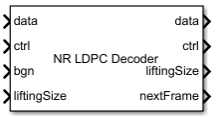
Integer-Input RS Encoder | Integer-Output RS Decoder | RS Encoder

Introduced in R2020a

NR LDPC Decoder

Decode LDPC code using layered belief propagation with min-sum or normalized min-sum approximation algorithm

Library: Wireless HDL Toolbox / Error Detection and Correction



Description

The NR LPDC Decoder block implements a low-density parity-check (LDPC) decoder with hardware-friendly control signals. The block accepts punctured log-likelihood ratio (LLR) values, a stream of control signals, a base graph number, and lifting sizes. The block outputs decoded bits, a stream of control signals, lifting sizes, and a signal that indicates when the block is ready to accept new inputs.

This block provides an option to implement layered belief propagation with either the normalized min-sum approximation algorithm or the min-sum approximation algorithm. This implementation matches that of the function `nrLDPCDecode`. You can use this block for channel coding of downlink and uplink shared channels and paging channel according to 5G new radio (NR) standard TS 38.212 [1].

The NR LDPC Decoder block supports scalar and 64-element column vector inputs. The block supports the early termination feature to help improve decoding performance and faster convergence speeds at high signal noise ratio (SNR) conditions. The block enables decoding of multiple code rates to help achieve high throughput efficiency with a high degree of code rate flexibility. The block provides an architecture suitable for HDL code generation and hardware deployment. For more information, see “Algorithms” on page 1-174.

Ports

Input

`data` — Input LLR values

scalar | vector

Input log-likelihood ratio (LLR) values, specified as a scalar or a column vector of size 64.

The data type of this input must be a signed fixed-point data type with a word length from 4 to 16 bits. For more information on how to specify vector input data, see “Specifying Vector Input” on page 1-172.

Data Types: `int8` | `int16` | `fixed point`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, specified as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the input frame
- `end` — Indicates the end of the input frame
- `valid` — Indicates that the data on the input **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

`bgn` — Base graph number

scalar

Base graph number, specified as a scalar. When this value is `0`, the block applies *bgn 1*. When this value is `1`, the block applies *bgn 2*. For more information about *bgn 1* and *bgn 2*, see section 5.3.2, of TS 38.212 [1].

Data Types: `Boolean`

`liftingSize` — Input lifting size

scalar

Input lifting size, specified as a scalar.

For an invalid **liftingSize** value, the block discards the current frame and waits for the new frame. For more information about the supported lifting size values, see section 5.3.2, of TS 38.212 [1].

Data Types: `uint16`

`iter` — Number of iterations

scalar

Number of iterations, specified as a integer in the range from 1 to 63.

If you specify **iter** as a value greater than 63, the block automatically sets the **iter** value to `8` and performs the decoding operation.

Dependencies

To enable this port, set the **Decoding termination criteria** parameter to `Max` and the **Source for number of iterations** parameter to `Input port`.

Data Types: `uint8`

`numRows` — Number of rows

scalar

Number of rows, specified as a scalar.

When you set the **bgn** value to `0` the block supports the number of rows in the range from 4 to 46. When you set the **bgn** value to `1`, the block supports the number of rows in the range from 4 to 42.

Dependencies

To enable this port, select the **Enable multiple code rates** parameter.

Data Types: `fixdt(0,6,0)`

Output

`data` — Decoded output data bits

scalar | vector

Decoded output data bits, returned as a scalar or a column vector of size 64.

The block outputs data bits in a similar format as the input LLR values. Extract these output data bits in a similar format for further processing.

Data Types: `Boolean`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, returned as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the output frame
- `end` — Indicates the end of the output frame
- `valid` — Indicates that the data on the output **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

`liftingSize` — Output lifting size

scalar

Output lifting size, returned as a scalar.

Data Types: `uint16`

`nextFrame` — Ready for new inputs

scalar

The block sets this signal to `1` when the block is ready to accept the start of the next frame. 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.

For more information, see “Using the nextFrame Output Signal”.

Data Types: `Boolean`

`actIter` — Actual number of iterations

scalar

Actual number of iterations the block takes to decode the output, returned as a scalar.

Dependencies

To enable this port, set the **Decoding termination criteria** parameter to `Early`.

Data Types: `uint8`

`parityCheck` — Parity check status indicator

scalar

Parity check status indicator, returned as a Boolean scalar. The port indicates the status of the parity check after the decoding operation.

- `0` — Indicates that the parity check failed
- `1` — Indicates that the parity check passed

Dependencies

To enable this port, select the **Enable parity check output port** parameter.

Data Types: `Boolean`

Parameters

`Algorithm` — Type of algorithm

`Min-sum` (default) | `Normalized min-sum`

Select the type of algorithm. For more information, see "Algorithm" (5G Toolbox).

`Scaling factor` — Scaling factor

`0.75` (default) | values in the range from 0.5 to 1, incremented by 0.0625

Specify the scaling factor.

Dependencies

To enable this parameter, set the **Algorithm** parameter to `Normalized min-sum`.

`Decoding termination criteria` — Termination criteria

`Max` (default) | `Early`

Select the decoding termination criteria.

- `Max` — Terminates decoding when the block reaches the number of iterations specified in the block mask or through the **iter** input port
- `Early` — Terminates decoding when all of the parity checks are met or when the block reaches the maximum number of iterations provided in the block mask

`Source for number of iterations` — Source selection for number of iterations

`Property` (default) | `Input port`

Select the source for specifying the number of iterations.

You can set the number of iterations by using either an input port or a parameter.

- Select `Property` to enable the **Number of iterations** parameter.
- Select `Input port` to enable the **iter** port.

Dependencies

To enable this parameter, set the **Decoding termination criteria** parameter to `Max`.

`Number of iterations` — Number of iterations

8 (default) | integer in the range from 1 to 63

Specify the number of iterations.

Dependencies

To enable this parameter, set the **Source for number of iterations** parameter to `Property`.

`Maximum number of iterations` — Maximum number of iterations
`8` (default) | integer in the range from 1 to 63

Specify the maximum number of iterations.

Dependencies

To enable this parameter, set the **Decoding termination criteria** parameter to `Early`.

`Enable multiple code rates` — Multiple code rates
`off` (default) | `on`

Select this parameter to enable the **numRows** input port to support multiple code rates. For more information about multiple code rates, see “Multiple Code Rates” on page 1-173.

`Enable parity check output port` — Parity check status
`off` (default) | `on`

Select this parameter to enable the **parityCheck** output port to view the status of the parity check.

More About

Specifying Vector Input

Vector input data for the block must be specified as a column vector of size 64. You must provide inputs as an integer number of `ceil`(**liftingSize**/64) clock cycles.

The total number of clock cycles that the block requires to receive a frame of LLR values for decoding is equal to *n* x `ceil`(**liftingSize**/64), where *n* is the number of columns in the parity check matrix. *n* depends on the base graph number, specified by the **bgn** input port. When the **bgn** port value is `0`, the block sets *n* to 66. When the **bgn** port value is `1`, the block sets *n* to 50.

These sections show how the block accepts input LLR values based on the **liftingSize** and **bgn** port values.

liftingSize input value is less than 64 and bgn value is 0

For a **liftingSize** input value of `2` and **bgn** input value of `0`, the block can accept 132 LLRs. In this case, the block accepts the first two LLR input bits in each clock cycle and ignores the remaining 62 elements in that clock cycle. The total number of clock cycles the block requires to receive a frame of LLR values is 66.

The L_n elements represent LLR bits, and the X elements represent ignored values.

Input LLR Values	Number of Clock Cycles					
	1 Clock Cycle	2 Clock Cycles	3 Clock Cycles	4 Clock Cycles	...	66 Clock Cycles
`data[0]`	L_0	L_2	L_4	L_6	...	L_{130}
`data[1]`	L_1	L_3	L_5	L_7	...	L_{131}

Input LLR Values	Number of Clock Cycles					
	1 Clock Cycle	2 Clock Cycles	3 Clock Cycles	4 Clock Cycles	...	66 Clock Cycles
...	X	X	X	X	X	X
`data[63]`	X	X	X	X	X	X

liftingSize input value is greater than 64 and bgn value is 0

For a **liftingSize** input value of `104` and **bgn** input value of `0`, the block can accept 6,864 LLRs. In this case, the block accepts 104 LLR values in two clock cycles: 64 LLRs in the first clock cycle and 40 LLRs in the second clock cycle. The block ignores the remaining 24 elements in the second clock cycle. The total number of clock cycles the block requires to receive input LLR values is 132.

The L_n elements represent LLR bits, and the X elements represent ignored values.

Input LLR Values	Number of Clock Cycles							
	1 Clock Cycle	2 Clock Cycles	3 Clock Cycles	4 Clock Cycles	...	...	131 Clock Cycles	132 Clock Cycles
`data[0]`	L_0	L_{64}	L_{104}	L_{168}	...	...	L_{6760}	L_{6824}
`data[1]`	L_1	L_{65}	L_{105}	L_{169}	...	...	L_{6761}	L_{6825}
	...	...	...		...	...	...	...
...	...	L_{103}	...	L_{207}	...	...	...	L_{6863}
...	...	X	...	X	...	...	...	X
`data[63]`	L_{63}	X	L_{167}	X	...	...	L_{6823}	X

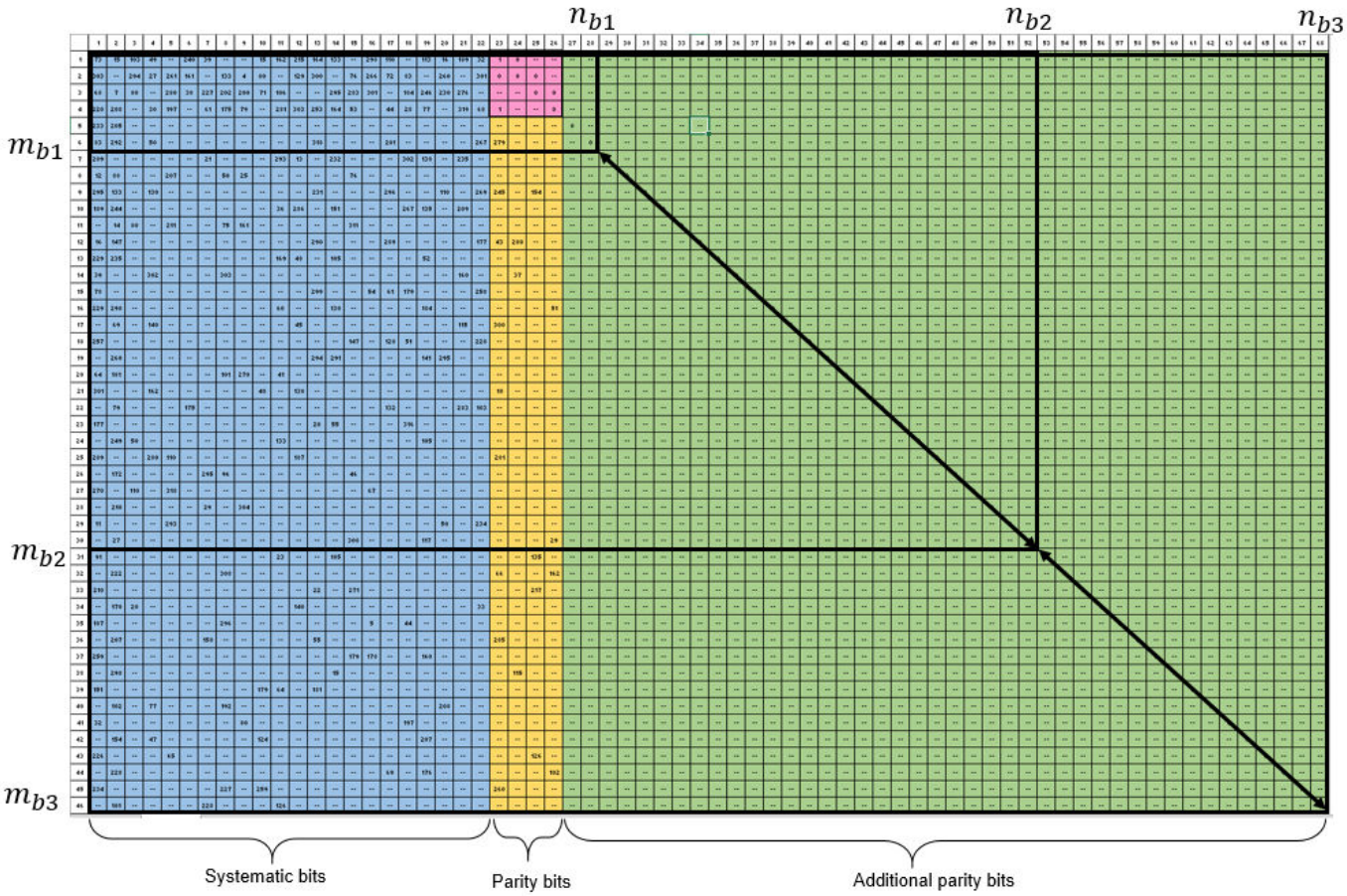
Multiple Code Rates

NR LDPC codes can support flexible code rates based on the parity check matrix (PCM) extension to achieve high throughputs and meet low latency requirements. The block supports multiple code rates by varying the number of rows of the parity check matrix.

For LDPC codes, the base parity check matrix (H_b) is a product of the number of rows (m_b) and the number of columns (n_b) of the matrix. The output (K) of the block is calculated as k_b x Z, where Z is the expansion factor or lifting size that can be in the range from 2 to 384, and k_b is equal to 22 for **bgn** value `0` and `10` for **bgn** value `1` as defined in the standard [1].

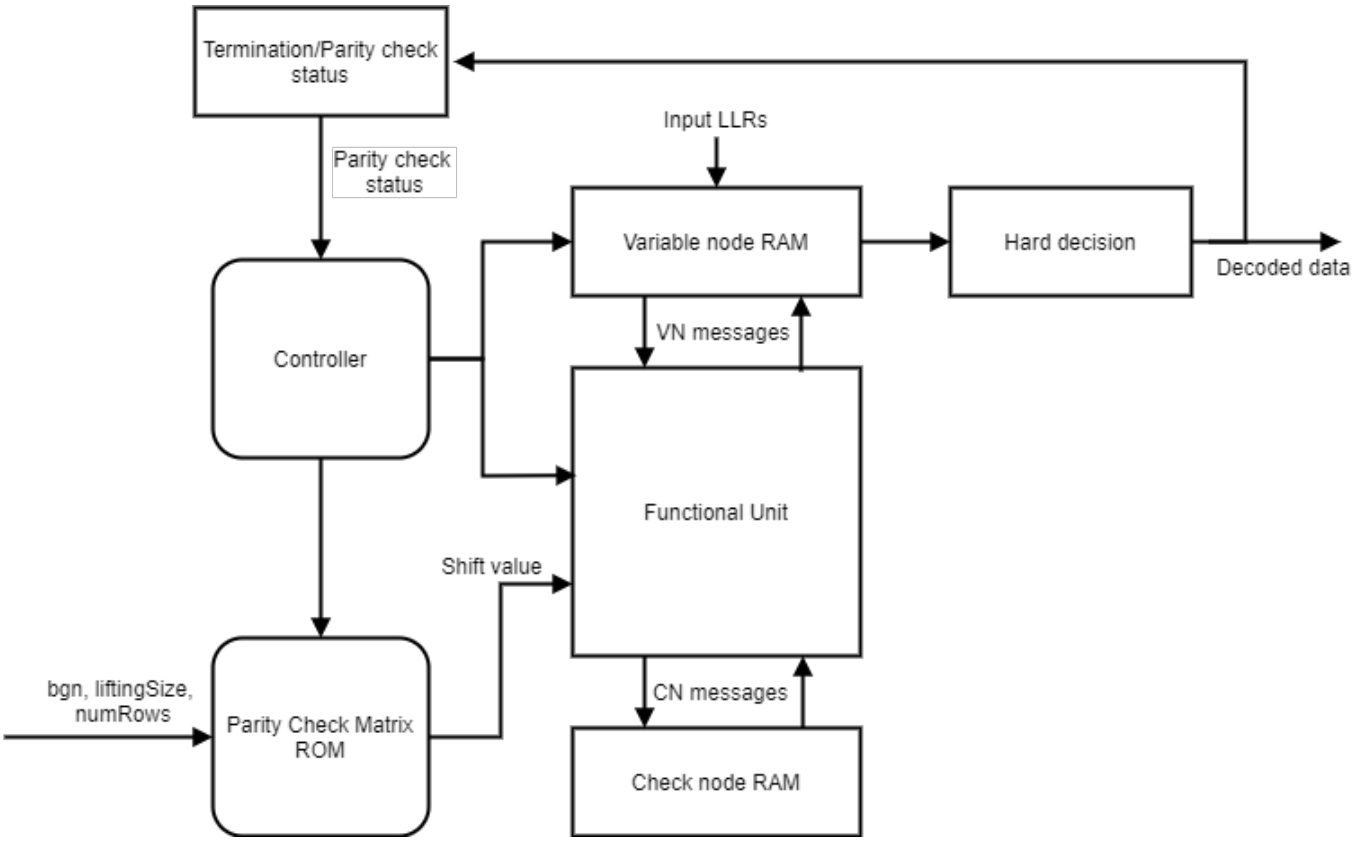
The input size (N) is calculated as, n_b x Z, where n_b is equal to m_b + k_b.

This figure shows a parity check matrix marked with a specified number of rows and columns, which you can use to calculate the code rates of the block. The code rate R is calculated as, k_b / (k_b – 2 + m_b) for the specified **bgn** value. In this figure, the values n_{b1}, n_{b2}, and n_{b3} indicate the number of columns for the specified **bgn** value and values m_{b1}, m_{b2}, and m_{b3} indicate the number of rows for the specified **bgn** value.



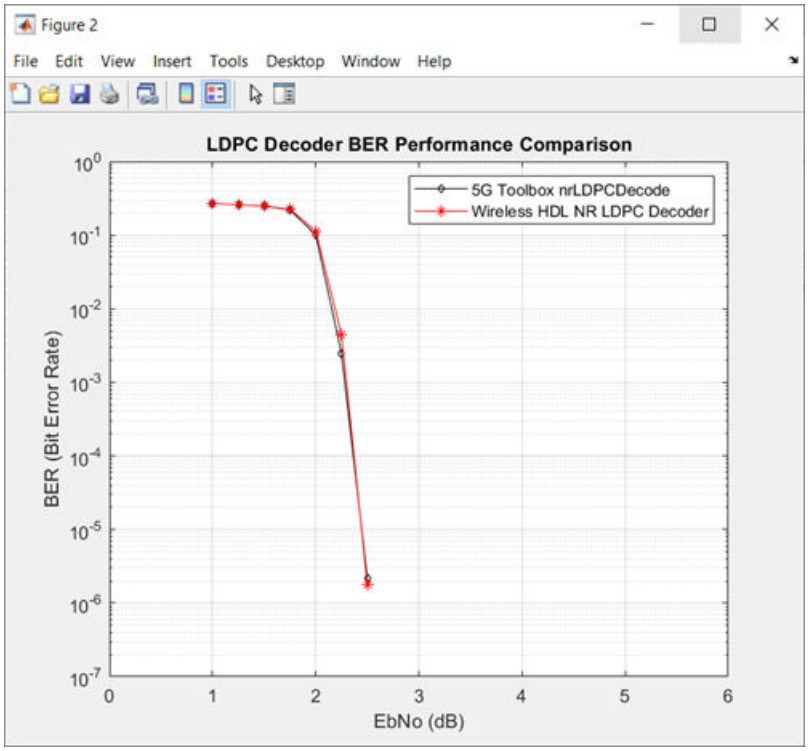
Algorithms

This figure shows the architecture block diagram of the NR LDPC Decoder block. The Controller block controls the layer and iteration count of the decoding process. The Variable node RAM block stores the variable node (VN) messages, and Check node RAM block stores the check node messages (CN). The Functional Unit block calculates the variable node (VN) messages and check node (CN) messages based on the layered belief propagation and either the normalized min-sum approximation algorithm or the min-sum approximation algorithm. The Termination/Parity check status block calculates the parity checks and provides the parity check status after each iteration. For more information about decoding algorithms, see “Algorithm” (5G Toolbox).

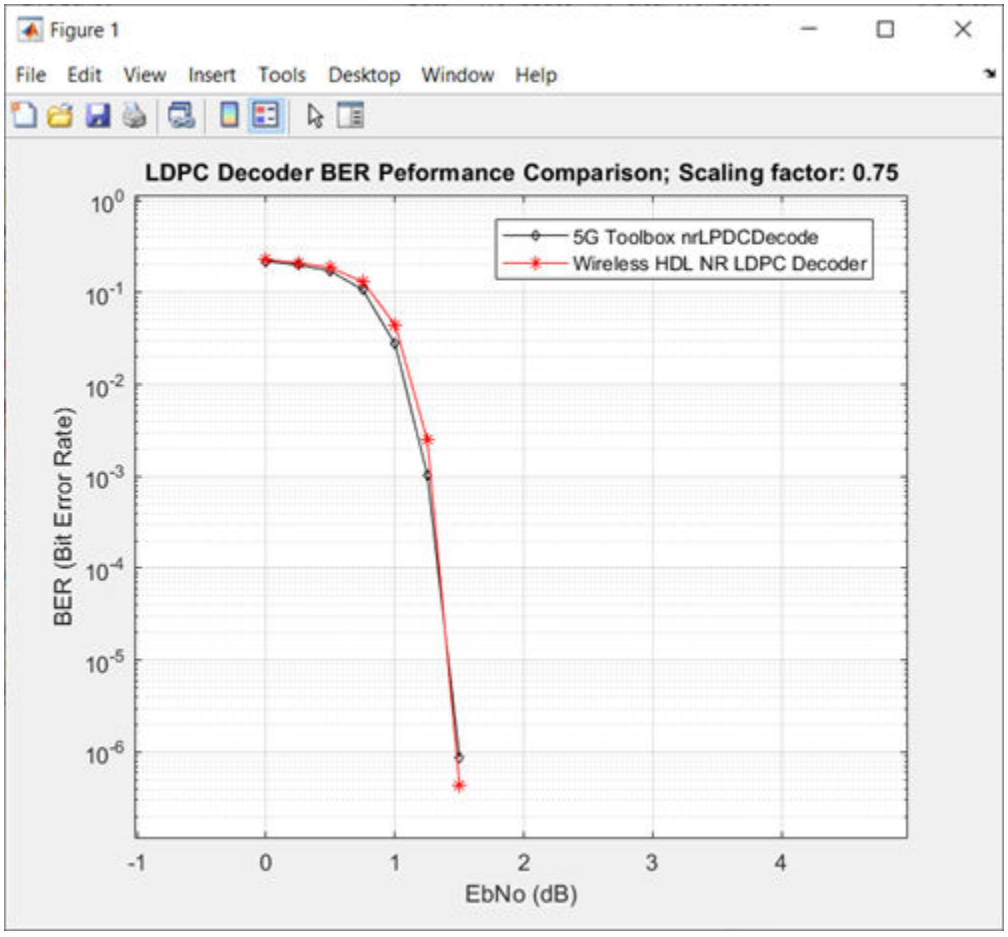


The implementation of the block matches the performance of the function `nrLDPCDecode`.

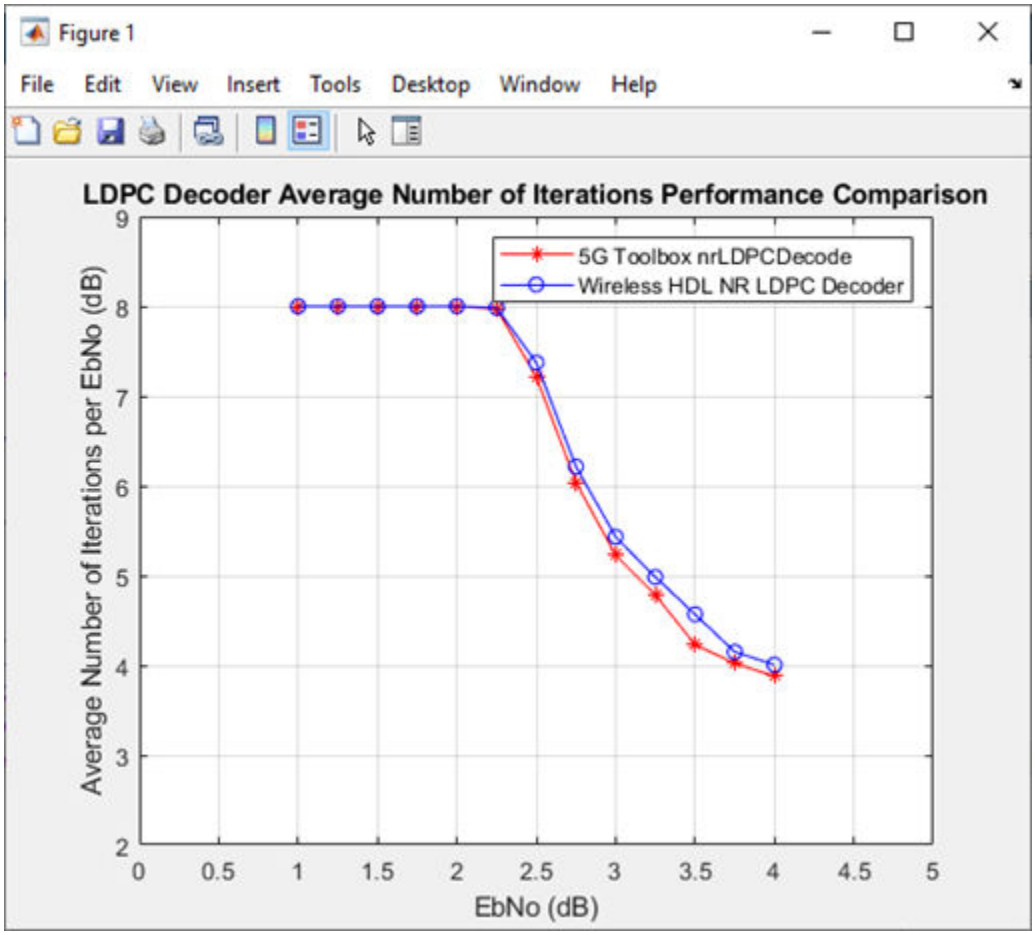
This plot shows the performance of the block for a 4-bit LLR input when you set the **Algorithm** parameter to `Min-sum`.



This plot shows the performance of the block for a 4-bit LLR input when you set the **Algorithm** parameter to `Normalized min-sum` .



This plot shows the average number of iterations taken to decode the data per EbNo for a 4-bit LLR input when you set the **Algorithm** parameter to `Min-sum` and the **Decoding termination criteria** parameter to `Early`.



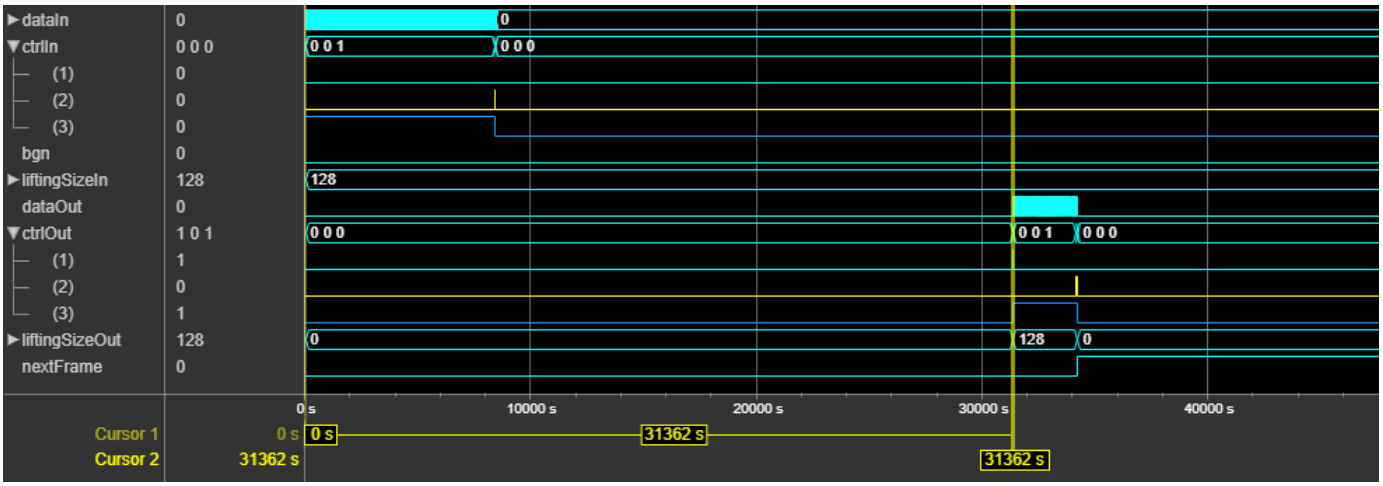
Latency

The latency of the block varies based on the values of the **bgn**, **liftingSize**, and **numRows** input ports and the number of iterations. Because the latency varies, use the **nextFrame** control signal output port to determine when the block is ready for a new input frame.

Scalar Input

The latency of the block is equal to r x (t + (m x 8) x `ceil`(**liftingSize**/64) + t + m x (7 – `ceil`(**liftingSize**/64))) + (n x **liftingSize**) + 18. In this calculation, r is the number of iterations, n is the number of columns in the parity check matrix, t is twice the total number of non -1 elements in the parity check matrix, m is the number of rows in the parity check matrix, and d is the pipeline delays. When you select the **Enable multiple code rates** parameter, d is 26. Otherwise, d is 18.

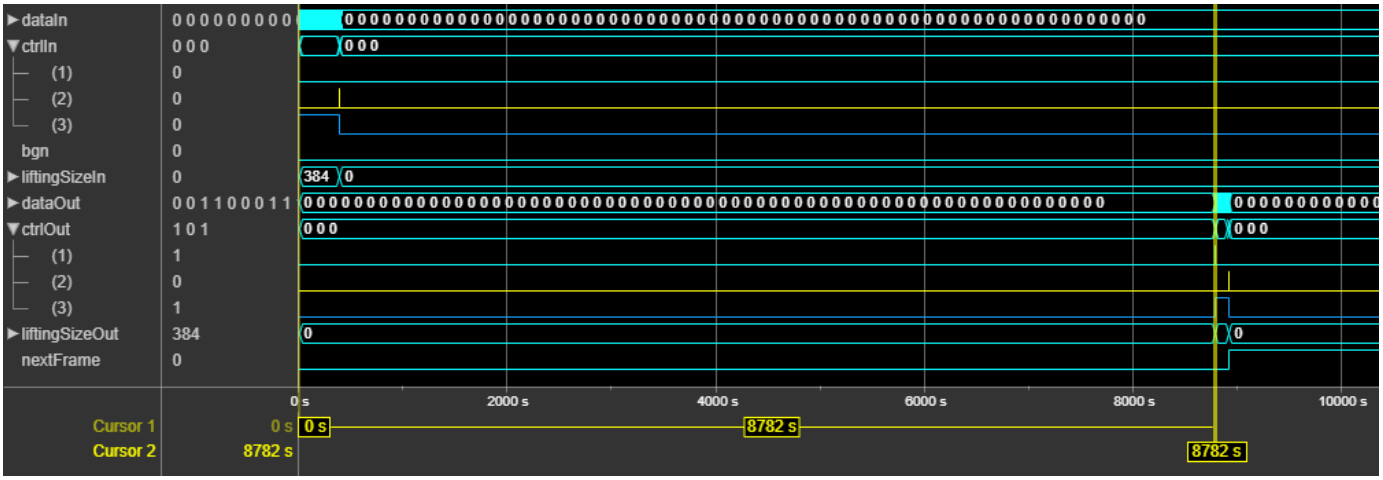
This figure shows a sample output of the NR LDPC Decoder block with latency. In this case, the **bgn** and **liftingSize** input port values are set to `0` and `128`, respectively, and the **Number of iterations** parameter is set to `8`. The latency of the block is 31,362 clock cycles.



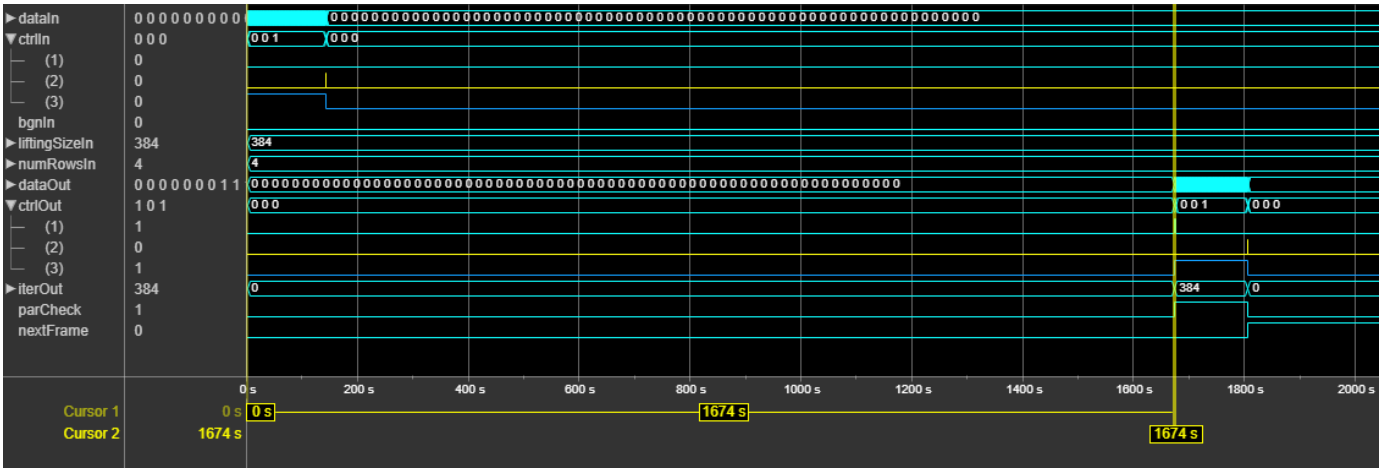
Vector Input

For vector inputs, the latency of the block is equal to r x (t + (m x 9)) + n x (`ceil`(**liftingSize**/64)) + d. In this calculation, r is the number of iterations, n is the number of columns in the parity check matrix, t is twice the total number of non -1 elements in the parity check matrix, m is the number of rows in the parity check matrix, and d is the pipeline delays. When you select the **Enable multiple code rates** parameter, d is 26. Otherwise, d is 18.

This figure shows a sample output of the NR LDPC Decoder block with latency. In this case, the **bgn** and **liftingSize** input port values are set to `0` and `384`, respectively, and the **Number of iterations** parameter is set to `8`. The latency of the block is 8,782 clock cycles.



This figure shows a sample output of the NR LDPC Decoder block with latency. In this case, the **bgn**, **liftingSize**, and **numRows** input port values are set to `0`, `384`, and `4`, respectively, and the **Number of iterations** parameter is set to `8`. The latency of the block is 1,674 clock cycles.



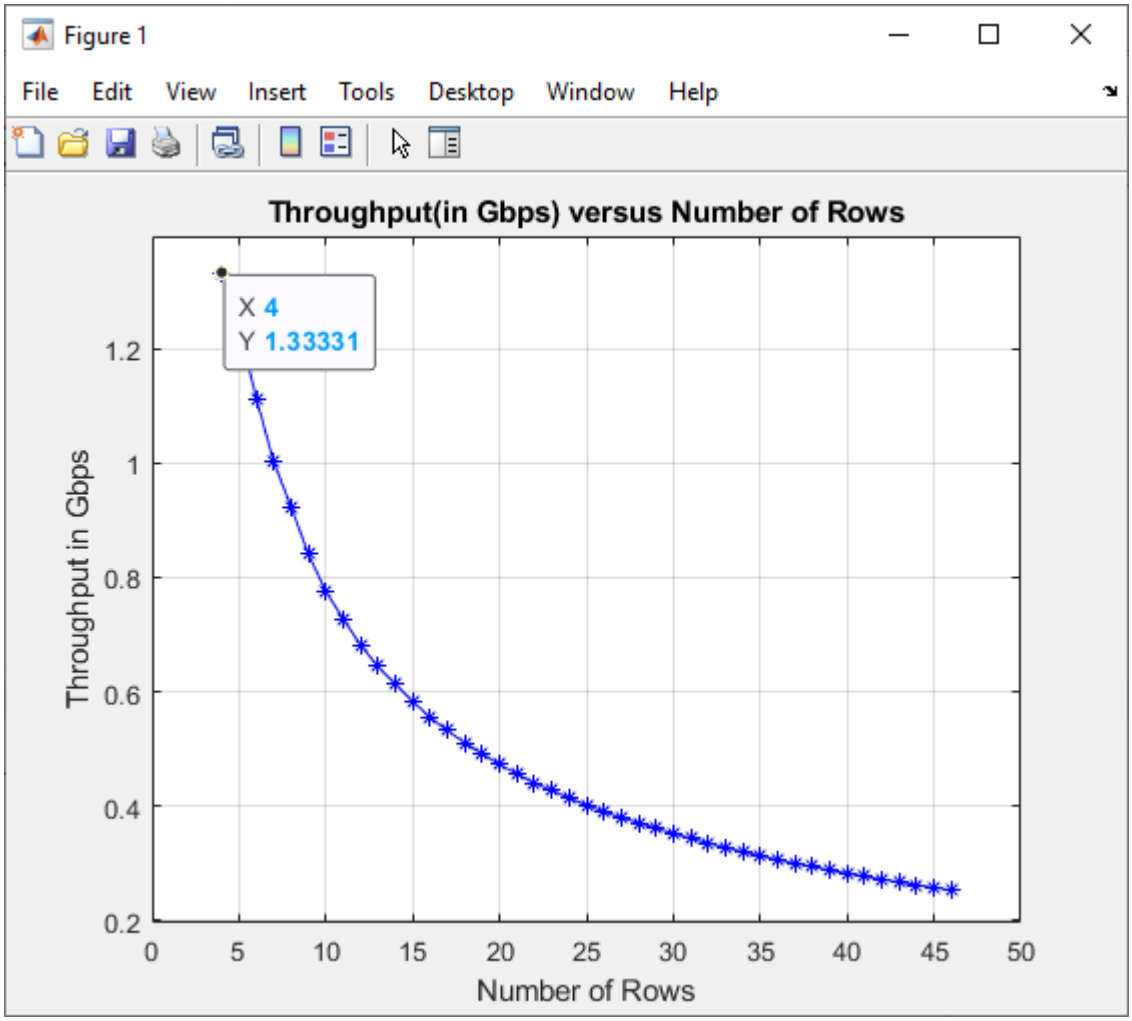
Throughput

The throughput of the block is calculated as (*cwLen* / *latency*) x $f_{\max}$. In this calculation:

- *cwLen* is the code word length which is equal to k_b x Z, where k_b is `22` for the **bgn** value `0` and `10` for the **bgn** value `1`.
- *latency* is the latency of the block for the specified configuration
- f_{max} is the maximum operating frequency

For more information about the latency calculation, see “Latency” on page 1-178. For more information about the maximum operating frequency, see “Performance” on page 1-181.

This plot shows the throughput versus the number of rows specified at the block input, when you set the **Algorithm** parameter to `Min-sum`, the **Number of iterations** parameter to `8`, and the **bgn** input port to `0`.



Performance

The performance of the synthesized HDL code varies with your target and synthesis options. It also varies based on the type of algorithm and the word length of the input LLR values.

This table shows the resource and performance data synthesis results of the block, when you set the **Algorithm** parameter to `Min-sum`, set the **Number of iterations** parameter to `8`, and specify the input LLR values of data type `fixdt(1,4,0)`. The generated HDL is targeted to the Xilinx ZynqUltrascale+™ RFSoC evaluation board.

Input Data	Slice LUTs	Slice Registers	Block RAMs	Maximum Frequency in MHz
Scalar	45461	58331	192.5	291
Vector	67410	75217	128.5	291.5

This table shows the resource and performance data synthesis results of the block for the vector input when you set the **Algorithm** parameter to `Min-sum`, set the **Decoding termination criteria** parameter to `Max`, set the **Number of iterations** parameter to `8`, select the **Enable multiple code**

rates parameter, and specify the input LLR values of data type `fixdt(1,4,0)`. The generated HDL is targeted to the Xilinx ZynqUltrascale+ RFSoC evaluation board.

Slice LUTs	Slice Registers	Block RAMs	Maximum Frequency in MHz
72527	76002	128.5	264.2

References

[1] 3GPP TS 38.212. “NR; Multiplexing and Channel Coding.” *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*.

[2] Gallager, R. “Low-Density Parity-Check Codes.” *IEEE Transactions on Information Theory* 8, no. 1 (January 1962): 21–28. www.doi.org/10.1109/TIT.1962.1057683.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

See Also

Blocks
NR LDPC Encoder

Functions
`nrLDPCDecode` | `nrLDPCEncode`

Introduced in R2020a

NR LDPC Encoder

Perform LDPC encoding according to 5G NR standard
Library: Wireless HDL Toolbox / Error Detection and Correction

Description

The NR LPDC Encoder block implements a low-density parity-check (LDPC) encoder with hardware-friendly control signals. The block accepts data bits, a stream of control signals, a base graph number, and lifting sizes. The block outputs encoded bits, a stream of control signals, lifting sizes, and a signal that indicates when the block is ready to accept new inputs.

The block functionality matches that of the function `nrLDPCEncode`. You can use this block for channel coding of downlink and uplink shared channels and paging channel according to 5G new radio (NR) standard TS 38.212 [1].

The block supports scalar and vector inputs. The block provides an architecture suitable for HDL code generation and hardware deployment. For more information, see “Algorithms” on page 1-187.

Ports

Input

`data` — Input data bits

scalar | vector

Input data bits, specified as a scalar or a column vector of size 64.

For more information on how to specify vector input data, see “Specifying Vector Input” on page 1-186.

Data Types: `Boolean`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, specified as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the input frame
- `end` — Indicates the end of the input frame
- `valid` — Indicates that the data on the input **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

`bgn` — Base graph number

scalar

Base graph number, specified as a scalar. When this value is `0`, the block applies *bgn 1*. When this value is `1`, the block applies *bgn 2*. For more information about *bgn 1* and *bgn 2*, see section 5.3.2, of TS 38.212 [1].

Data Types: `Boolean`

`liftingSize` — Input lifting size

scalar

Input lifting size, specified as a scalar.

For an invalid **liftingSize** value, the block discards the current frame and waits for the new frame.

For more information about the supported lifting size values, see section 5.3.2, of TS 38.212 [1].

Data Types: `uint16`

Output

`data` — Encoded output data bits

scalar | vector

Encoded output data bits, returned as a scalar or a column vector of size 64.

The block outputs data bits in a similar format as the input data bits.

Data Types: `Boolean`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, returned as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the output frame
- `end` — Indicates the end of the output frame
- `valid` — Indicates that the data on the output **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

`liftingSize` — Output lifting size

scalar

Output lifting size, returned as a scalar.

Data Types: `uint16`

`nextFrame` — Ready for new inputs

scalar

The block sets this signal to `1` when the block is ready to accept the start of the next frame. 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.

For more information, see “Using the nextFrame Output Signal”.

Data Types: `Boolean`

More About

Specifying Vector Input

Vector input data for the block must be specified as a column vector of size 64. You must provide inputs as an integer number of `ceil`(**liftingSize**/64) clock cycles.

The total number of clock cycles that the block requires to receive a frame of data bits for encoding is equal to *n* x `ceil`(**liftingSize**/64), where *n* is the number of columns in the parity check matrix. *n* depends on the base graph number, specified by the **bgn** input port. When the **bgn** port value is `0`, the block sets *n* to 22. When the **bgn** port value is `1`, the block sets *n* to 10.

These sections show how the block accepts input data bits based on the **liftingSize** and **bgn** port values.

liftingSize input value is less than 64 and bgn value is 0

For a **liftingSize** input value of 2, the block accepts the first two data input bits in each clock cycle and ignores the remaining 62 elements in that clock cycle. The total number of clock cycles the block requires to receive input data bits is 22.

The D_n elements represent data bits, and the X elements represent ignored values.

Input data bits	Number of Clock Cycles						
	1 Clock Cycle	2 Clock Cycles	3 Clock Cycles	4 Clock Cycles	...	21 Clock Cycles	22 Clock Cycles
`data[0]`	D_0	D_2	D_4	D_6	...	...	D_{42}
`data[1]`	D_1	D_3	D_5	D_7	...	...	D_{43}
...	X	X	X	X	X	X	X
...	X	X	X	X	X	X	X
`data[63]`	X	X	X	X	X	X	X

liftingSize input value is greater than 64 and bgn value is 0

For a **liftingSize** input value of `104`, the block accepts 104 data bits in two clock cycles: 64 data bits in the first clock cycle and 40 data bits in the second clock cycle. The block ignores the remaining 24 elements in the second clock cycle. The total number of clock cycles the block requires to receive input data bits is 44.

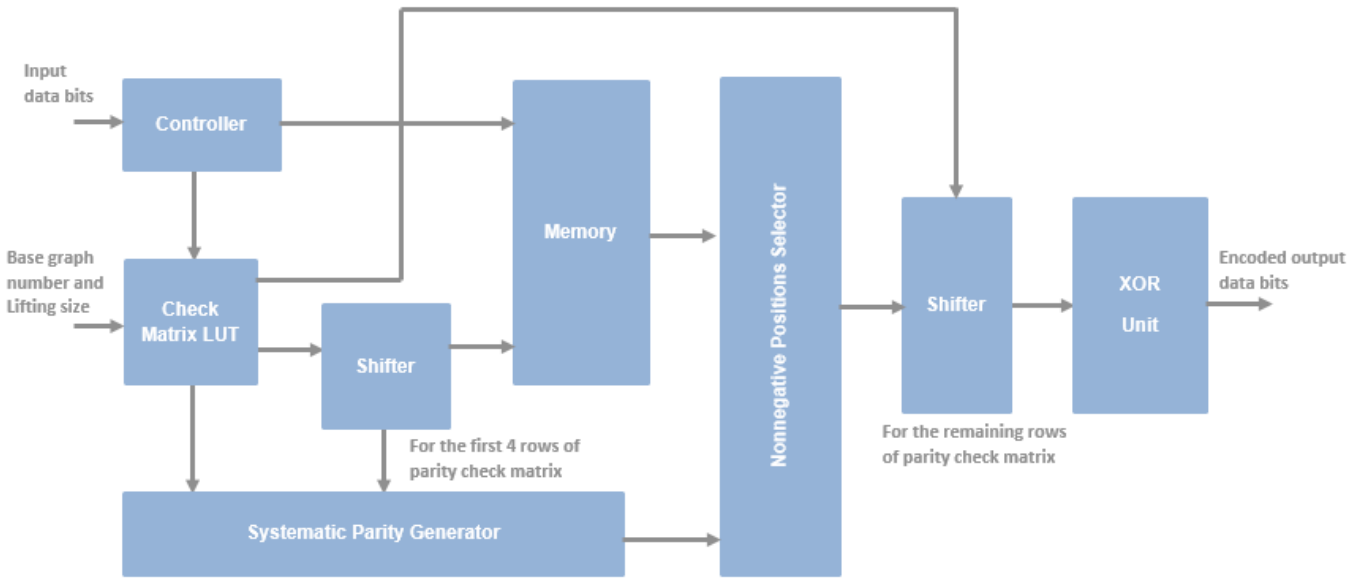
The D_n elements represent data bits, and the X elements represent ignored values.

Input data bits	Number of Clock Cycles							
	1 Clock Cycle	2 Clock Cycles	3 Clock Cycles	4 Clock Cycles	...	...	43 Clock Cycles	44 Clock Cycles
`data[0]`	D_0	D_{64}	D_{104}	D_{168}	...	...	D_{2184}	D_{2248}
`data[1]`	D_1	D_{65}	D_{105}	D_{169}	...	...	D_{2185}	D_{2249}
	...	...	...		...	...	...	...
...	...	D_{103}	...	D_{207}	...	...	...	D_{2287}
...	...	X	...	X	...	...	...	X
`data[63]`	D_{63}	X	D_{167}	X	...	...	D_{2247}	X

Algorithms

This figure shows the architecture block diagram of the NR LDPC Encoder block.

The architecture consists of Controller, Check Matrix LUT, Shifter, Memory, Nonnegative Position Selector, and XOR Unit blocks. The Controller block controls the data flow to and from the Memory block and provides control signals to control the functionality of all of these blocks. The Check Matrix LUT block consists of 5G NR LDPC standard [1] parity check matrix values. Based on the **bgn** and **liftingSize** input port values, the Check Matrix LUT block provides input to the Shifter block. The Systematic Parity Generator block generates parity bits for the first four rows of the parity check matrix and uses those generated parity bits to calculate the parity bits for the remaining rows of the parity check matrix. The Nonnegative Position Selector block selects the nonnegative positions of the parity check matrix. The XOR Unit block performs the modulo operation by completing the encoding operation.

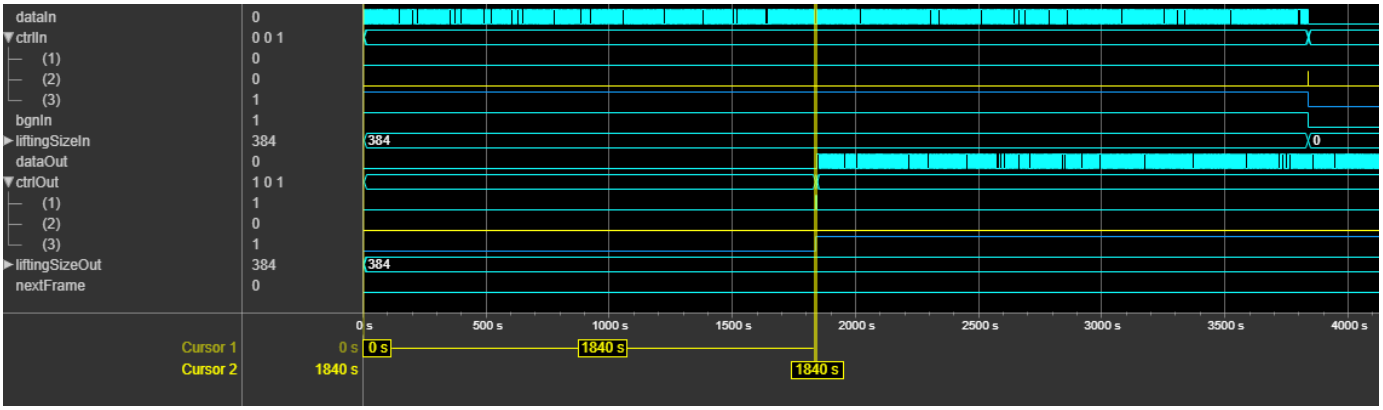


Latency

The latency of the block varies based on the values of the **bgn** and **liftingSize** input ports. Because the latency varies, use the **nextFrame** control signal to determine when the block is ready for a new input frame.

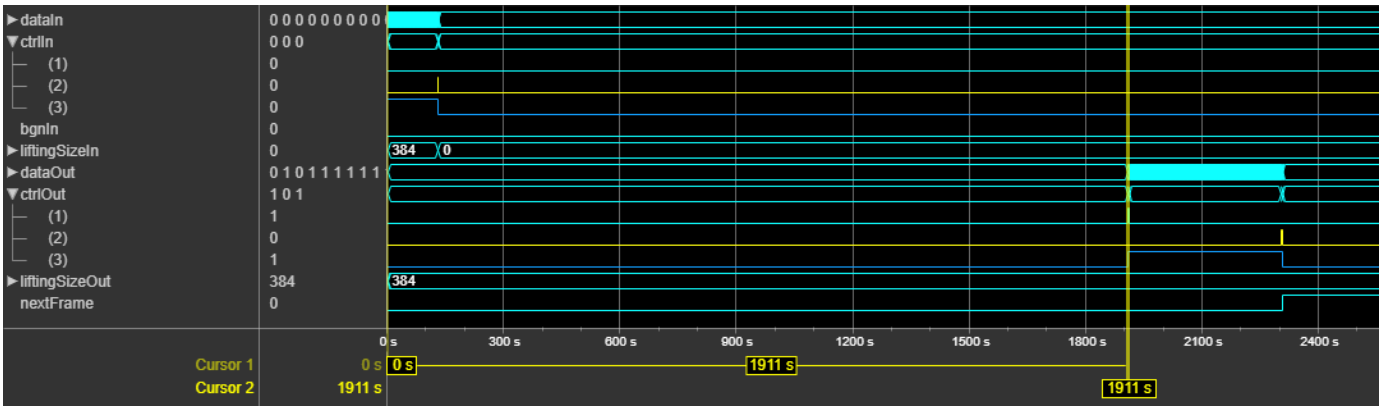
Scalar Input

This figure shows a sample output of the NR LDPC Encoder block with latency. In this case, the **bgn** and **liftingSize** input port values are set to `1` and `384`, respectively. The latency of the block is 1,840 clock cycles.



Vector Input

This figure shows a sample output of the NR LDPC Encoder block with latency. In this case, the **bgn** and **liftingSize** input port values are set to `0` and `384`, respectively. The latency of the block is 1,911 clock cycles.



Performance

The performance of the synthesized HDL code varies with your target and synthesis options.

This table shows the resource and performance data synthesis results. The generated HDL is targeted to the Xilinx Zynq- 7000 ZC706 evaluation board.

Input Data	Slice LUTs	Slice Registers	Block RAMs	Maximum Frequency in MHz
Scalar	6748	7084	2.5	431
Vector	7951	8504	3.5	430

References

[1] 3GPP TS 38.212. “NR; Multiplexing and Channel Coding.” *3rd Generation Partnership Project; Technical Specification Group Radio Access Networ*k.

[2] Gallager, R. “Low-Density Parity-Check Codes.” *IEEE Transactions on Information Theory* 8, no. 1 (January 1962): 21–28. www.doi.org/10.1109/TIT.1962.1057683.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

See Also

Blocks
NR LDPC Decoder

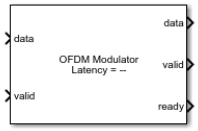
Functions
`nrLDPCDecode` | `nrLDPCEncode`

Introduced in R2020a

OFDM Modulator

Modulate frequency-domain OFDM subcarriers to time-domain samples for custom communication protocols

Library: Wireless HDL Toolbox / Modulation



Description

The OFDM Modulator block modulates frequency-domain orthogonal frequency division multiplexing (OFDM) subcarriers to time-domain samples based on the OFDM parameters. The block supports 5G new radio (NR) standard, long term evolution (LTE) [1], wireless local area network (WLAN 802.11a/g/n/ac) [2], WiMAX, digital video broadcast (DVB), and digital audio broadcast (DAB) standards.

The block accepts input data along with a valid control signal and these OFDM parameters: FFT length, CP length, and the number of right and left guard subcarriers. The block outputs modulated data along with valid and ready controls signals. The block samples the corresponding OFDM parameters only when the **ready** port is 1 (high) and when the first **valid** port of each OFDM symbol is 1 (high).

The block supports scalar and vector inputs. You can use a vector input to increase the data throughput and achieve a giga-sample-per-second (GSPS) throughput. The block supports windowing for scalar and vector inputs to reduce the spectral regrowth, or adjacent channel leakage ratio (ACLR), of an OFDM signal. The block provides an interface and architecture suitable for HDL code generation and hardware deployment.

Ports

Input

`data` — Input data

scalar | column vector

Input data, specified as a scalar or column vector of real or complex values. The vector size must be a power of 2 and in the range from 1 to 64, and less than or equal to the FFT length. For more information on how to specify vector inputs, see “Specifying Vector Input” on page 1-196.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `signed fixed point`

`valid` — Indicates valid input data

scalar

Indicates valid input data, specified as a scalar.

This port is a control signal that indicates when the sample from the **data** input port is valid. When this value is `1`, the block captures the values on the **data** input port. When this value is `0`, the block ignores the values on the **data** input port.

Data Types: `Boolean`

FFTLen — Length of FFT

scalar

Length of the FFT, specified as a scalar. The FFT length must be power of 2 and in the range from 8 to 65,536. This value must be less than or equal to the **Maximum FFT length** parameter value.

To support the minimum FFT length of 8, the **FFTLen** data type must be `fixdt(0,k,0)`, where *k* is greater than or equal to 4.

Dependencies

To enable this port, set the **OFDM parameters source** parameter to `Input port`.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `unsigned fixed point`

CPLen — Length of cyclic prefix

scalar

Length of the cyclic prefix, specified as a scalar in the range from 0 to **FFTLen**.

To support the minimum FFT length of 8, the **CPLen** data type must be `fixdt(0,k,0)`, where *k* is greater than or equal to 4.

Dependencies

To enable this port, set the **OFDM parameters source** parameter to `Input port`.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `unsigned fixed point`

numLgSc — Number of left guard carriers of OFDM symbol

scalar

Number of left guard carriers of the OFDM symbol, specified as a scalar in the range from 0 to (**FFTLen**/2) – 1.

To support the minimum FFT length of 8, the **numLgSc** data type must be `fixdt(0,k,0)`, where *k* is greater than or equal to 2.

Dependencies

To enable this port, set the **OFDM parameters source** parameter to `Input port`.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `unsigned fixed point`

numRgSc — Number of right guard carriers of OFDM symbol

scalar

Number of right guard carriers of the OFDM symbol, specified as a scalar in the range from 0 to (**FFTLen**/2) – 1.

To support the minimum FFT length of 8, the **numRgSc** data type must be `fixdt(0,k,0)`, where *k* is greater than or equal to 2.

Dependencies

To enable this port, set the **OFDM parameters source** parameter to `Input port`.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `unsigned fixed point`

`reset` — Clear internal states

scalar

Clear internal states, specified as a Boolean scalar. When this value is `1`, the block stops the current calculation and clears all internal states.

Dependencies

To enable this port, select the **Enable reset input port** parameter.

Data Types: `Boolean`

`winLen` — Length of window

scalar

Length of the window for overlap-adding of adjacent OFDM symbols, specified as a scalar in the range from 1 to **Maximum window length**.

Dependencies

To enable this port, set the **OFDM parameters source** parameter to `Input port`.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `unsigned fixed point`

Output

`data` — Modulated output data

scalar | column vector

Modulated output data, returned as a complex-valued scalar or column vector. The data type this output is dependent on the data type of the input **data** port.

- When you set the **OFDM parameters source** parameter to `Property` and clear the **Divide butterfly outputs by two** parameter, the output word length increases by $\log_2$(**FFT length**) bits.
- When you set the **OFDM parameters source** parameter to `Input port` and clear the **Divide butterfly outputs by two** parameter, the output word length increases by $\log_2$(**Maximum FFT length**) bits.

To avoid overflow, select the **Divide butterfly outputs by two** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `signed fixed point`

`valid` — Indicates valid output data

scalar

Indicates valid output data, returned as a scalar.

This port is a control signal that indicates when the **data** output port is valid. The block sets this value to `1` when the data samples are available on the **data** output port.

Data Types: `Boolean`

`ready` — Indicates block is ready

scalar

Indicates block is ready, returned as a scalar.

This is a control signal that indicates when the block is ready for new input data. When this value is `1`, the block accepts input data in the next time step. When this value is `0`, the block ignores input data in the next time step.

Data Types: `Boolean`

Parameters

Main

`OFDM parameters source` — Source of OFDM parameters

`Property` (default) | `Input port`

You can set OFDM parameters with an input port or by selecting a value for the parameter.

Select `Property` to enable the **FFT length**, **Cyclic prefix length**, **Number of left guard subcarriers**, and **Number of right guard subcarriers** parameters.

Select `Input port` to enable the **FFTLen**, **CPLen**, **numLgSc**, and **numRgSc** input ports and the **Maximum FFT length** parameter. The **Maximum FFT length** parameter sets the upper bound of the range of valid values for the **FFTLen** input port.

`Maximum FFT length` — Maximum length of FFT length

64 (default) | power of 2 in range from 8 to 65,536

Specify the maximum length of the FFT.

Dependencies

To enable this parameter, set the **OFDM parameters source** parameter to `Input port`.

`FFT length` — Length of FFT

64 (default) | power of 2 in range from 8 to 65, 536

Specify the FFT length.

When you set the **OFDM parameters source** parameter to `Property`, the block uses the **FFT length** value as the maximum FFT length.

Dependencies

To enable this parameter, set the **OFDM parameters source** parameter to `Property`.

`Cyclic prefix length` — Length of cyclic prefix

16 (default) | integer in range from 0 to **FFT length**

Specify the length of the cyclic prefix.

Dependencies

To enable this parameter, set the **OFDM parameters source** parameter to `Property`.

`Number of left guard subcarriers` — Number of guard band subcarriers in left extreme of OFDM symbol
6 (default) | integer in range from 0 to (**FFT length**/2) – 1

Specify the number of left guard subcarriers.

Dependencies

To enable this parameter, set the **OFDM parameters source** parameter to `Property`.

`Number of right guard subcarriers` — Number of guard band subcarriers in right extreme of OFDM symbol
5 (default) | integer in range from 0 to (**FFT length**/2) – 1

Specify the number of right guard subcarriers.

Dependencies

To enable this parameter, set the **OFDM parameters source** parameter to `Property`.

`Insert DC Null` — Option to insert DC null
`on` (default) | `off`

Select this parameter to insert a null on the DC subcarrier.

`Enable reset input port` — Reset signal
`off` (default) | `on`

Select this parameter to enable the **reset** input port.

`Windowing` — Spectral growth reduction
`off` (default) | `on`

Select this parameter to perform a windowing operation that reduces spectral growth based on the specified window length. Clear this parameter to disable the windowing operation. For more information about windowing, see “Windowing” on page 1-202.

`Window length` — Length of window
4 (default) | even integer in range from 1 to **Cyclic prefix length**

Specify the window length to overlap-add adjacent OFDM symbols.

Dependencies

To enable this parameter, set the **OFDM parameters source** parameter to `Property` and select the **Windowing** parameter.

`Maximum window length` — Maximum length of window
8 (default) | integer in the range from 1 to **CPLen**

Specify the maximum window length.

Dependencies

To enable this parameter, set the **OFDM parameters source** parameter to `Input port` and select the **Windowing** parameter.

IFFT Parameters

`Divide butterfly outputs by two` — Divide FFT butterfly outputs by two
`on` (default) | `off`

This parameter controls the scaling option of the IFFT HDL Optimized block inside the OFDM Modulator block.

When you select this parameter, the FFT implements an overall 1/*N* scale factor by dividing the output of each butterfly multiplication by two. This adjustment keeps the output of the IFFT in the same amplitude range as its input. If you clear this parameter, the block avoids overflow by increasing the word length by one bit after each butterfly multiplication.

`Rounding Method` — Rounding mode for internal fixed-point calculations
`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Zero`

This parameter specifies the type of rounding mode for internal fixed-point calculations. For more information about rounding modes, see Rounding Modes (DSP System Toolbox). When the input is any integer data type or fixed-point data type, the FFT algorithm uses fixed-point arithmetic for internal calculations. This parameter does not apply when the input is of data type `single` or `double`. Rounding applies to twiddle-factor multiplication and scaling operations.

More About

Specifying Vector Input

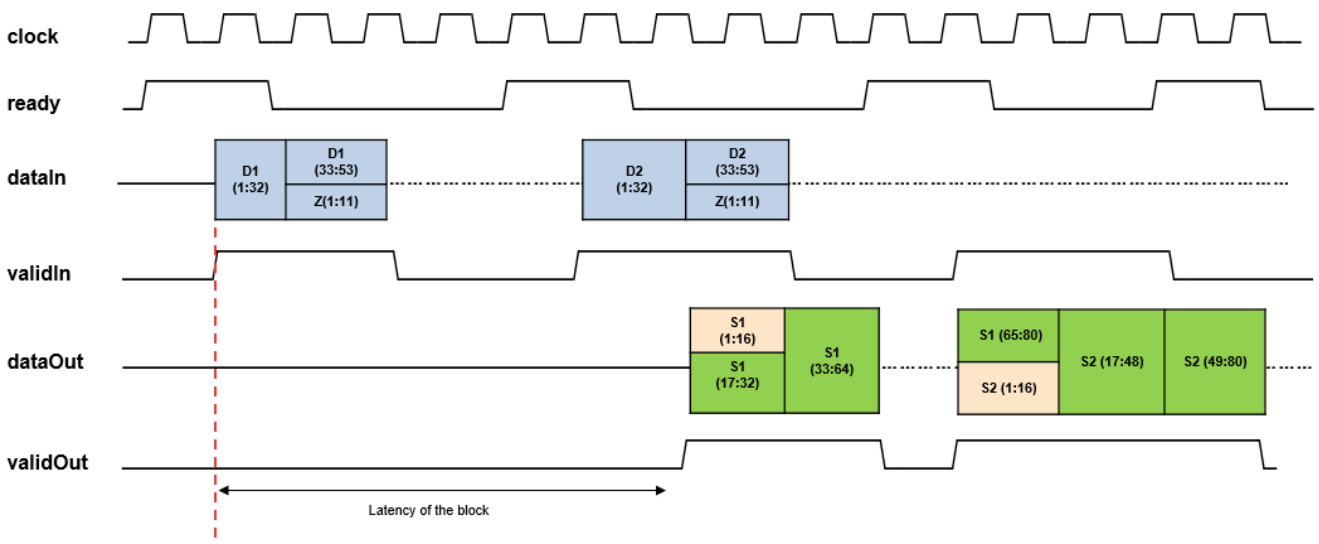
The OFDM Modulator block accepts active input subcarriers, which are calculated using the formula **FFT length** - (**Number of left guard subcarriers** + **Number of right guard subcarriers** + **Insert DC Null**). When you specify a vector input, if the number of active input subcarriers are insufficient to accommodate the vector length of data samples, you must pad zeros to the input to make it a complete vector. For example, if the number of active data subcarriers is 62, and the vector length is 16, you can specify the 48 data samples in 3 valid cycles and the remaining 14 data samples in the 4th valid cycle by padding two zeros to match the vector length of 16. If the number of active data subcarriers is 64, and the vector length is 16, you can specify the complete data samples in 4 valid cycles.

The block outputs **Cyclic prefix length** + **FFT length** number of samples. If the output data samples returned are insufficient to accommodate the vector length, the block stores the remaining samples and outputs them along with the data samples in the first valid cycle of the next data symbol. For example, if the number of data samples to be returned is 62, and the specified vector length is 16, the block returns 48 data samples in 3 valid cycles, stores the remaining 14 data samples and outputs them in the first output valid cycle of the next data symbol. If the number of data samples to be returned is 64, and the specified vector length is 16, the block returns data samples in 4 valid cycles.

Example 1

For a vector input of size 32 with block parameter **FFT length** set to `64`, **Cyclic prefix length** set to `16`, **Number of left guard subcarriers** set to `6`, **Number of right guard subcarriers** set to `5`, and **Insert DC Null** set to `off`, the block accepts 53 active data subcarriers.

In the figure, D1 and D2 indicate the active data subcarriers, Z indicates padded zeros, and S1 and S2 indicate modulated output data symbols.



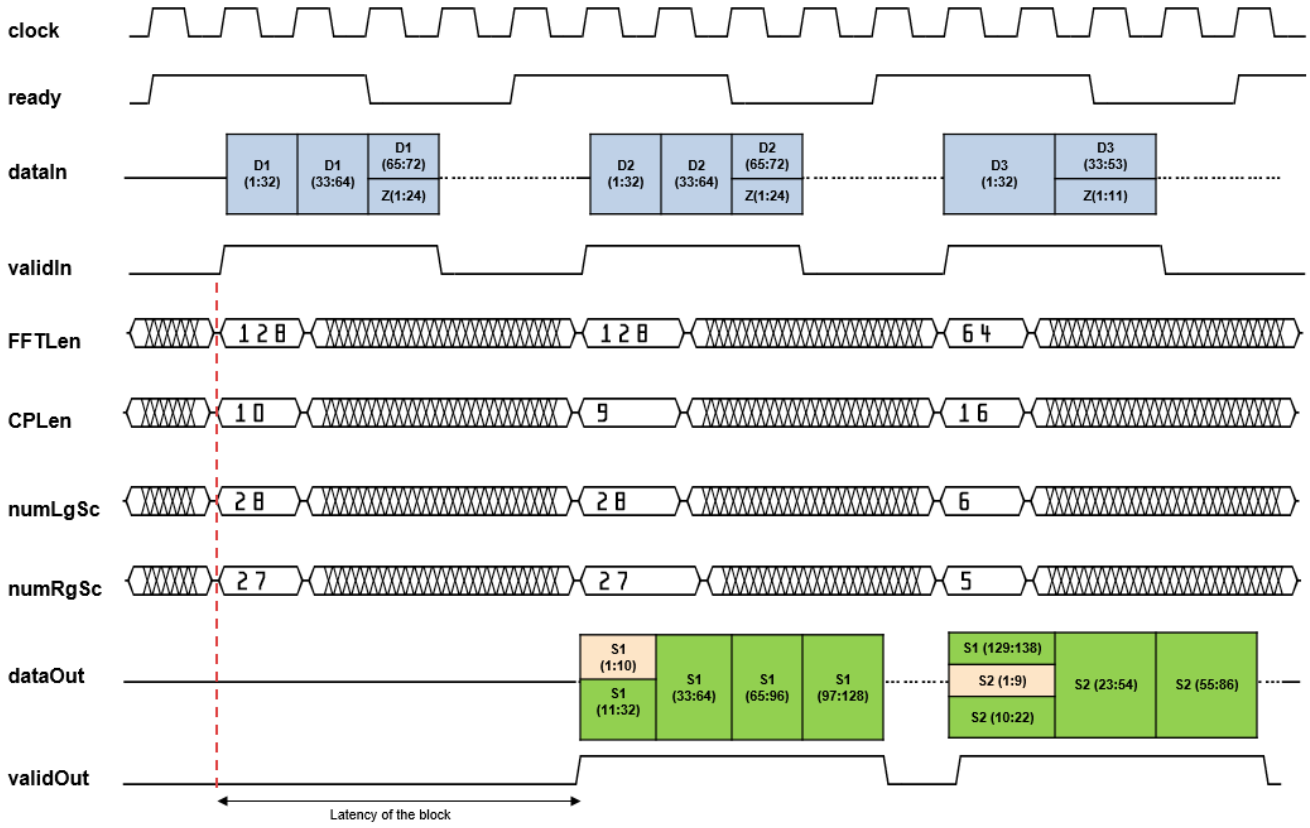
To provide 53 active data subcarriers of vector length 32, two cycles are needed. In this case, you must send the first set of data samples as D1(1:32), send the second set of data samples as D1(33:53), and pad the remaining samples with zeros Z(1:11) to make the complete vector. Similarly, the block processes D2 based on the block parameter values.

The block outputs **Cyclic prefix length** + **FFT length** number of samples, which means 80 data samples in this example. In the figure, S1(1:16) contains the added cyclic prefix, and S1(17:80) contains the data samples of the first data symbol S1, which is returned as S1(17:32), S1(33:64), and the remaining 16 data samples S1(65:80) stored and returned at the first valid cycle of the second data symbol S2. Similarly, the block outputs S2 based on the block parameter values as shown in the figure.

Example 2

For a vector input of size 32 with block inputs **FFTLen**, **CPLen**, **numLgSc**, and **numRgSc** specified as `128`, `10`, `28`, and `27`, respectively, at the first valid high cycle and with the **Insert DC Null** parameter cleared, the block accepts 73 active data subcarriers.

In the figure, D1 and D2 indicate the active data subcarriers, Z indicates padded zeros, and S1 and S2 indicate modulated output data symbols.

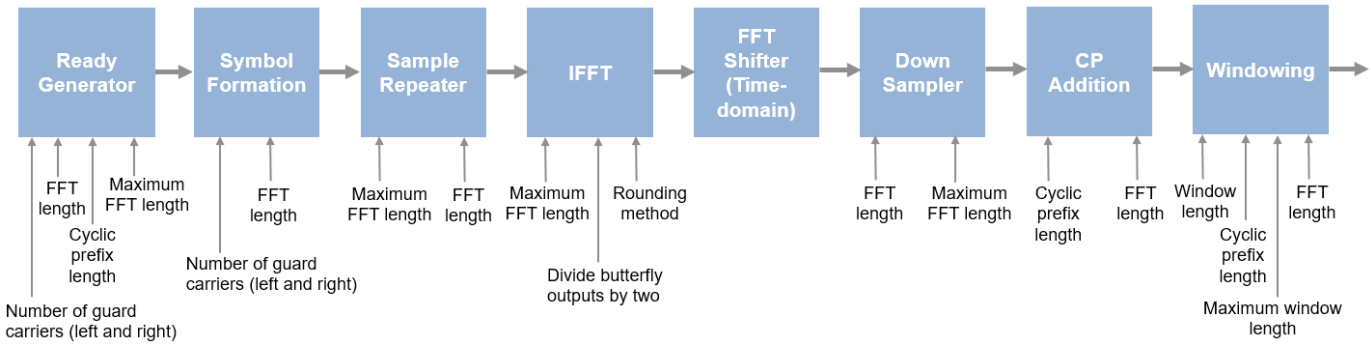


To provide 73 active data subcarriers of vector length 32, three cycles are needed. In this case, you must send the first set of data samples as D1(1:32), send the second set of data samples as D1(65:96), send the third set of data samples as D1(65:72), and pad the remaining samples with zeros Z(1:24) to make the complete vector. Similarly, the block processes D2 based on the port values.

The block outputs **Cyclic prefix length** + **FFT length** number of samples, which means 138 data samples in this example. In the figure, the output S1(1:10) contains the added cyclic prefix, and S1(11:138) contains the data samples of the first data symbol S1, which is returned as S1(11:32), S1(33:64), S1(65:96), S1(97:128), and the remaining 10 data samples S1(129:138) stored and returned at the first valid cycle of the second data symbol S2. Similarly, the block outputs S2 based on the port values shown in the figure.

Algorithms

The OFDM Modulator block operation sequence is implemented using these blocks: Ready Generator, Symbol Formation, Sample Repeater, IFFT, FFT Shifter, Down Sampler, CP Addition, and Windowing. The windowing feature is supported for scalar and vector inputs. The parameters shown in this figure configure the behavior of the block.



Ready Generator

The Ready Generator subsystem controls input data samples by calculating the number of active data subcarriers based on these input OFDM parameters: FFT length, CP length, number of left and right guard carriers, and DC null insertion status.

When you set the **OFDM parameters source** parameter to `Property`, these equations apply.

- N_h = `ceil`((**FFT length** – (**Number of left guard subcarriers** + **Number of right guard subcarriers** + **Insert DC Null**))/*vecLen*)
- N_l = `ceil`((**FFT length** + **Cyclic prefix length**))/*vecLen*) – N_h

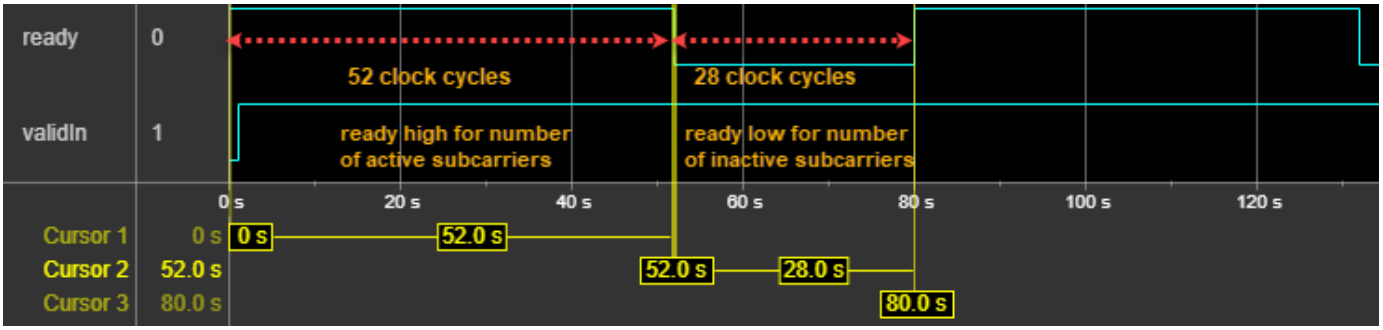
When you set the **OFDM parameters source** parameter to `Input port`, these equations apply.

- N_h = `ceil`((**FFTLen** – (**numLgSc** + **numRgSc** + **Insert DC Null**))/*vecLen*)
- N_l = `ceil`((**Maximum FFT length** + **CPLen**))/*vecLen*) – N_h

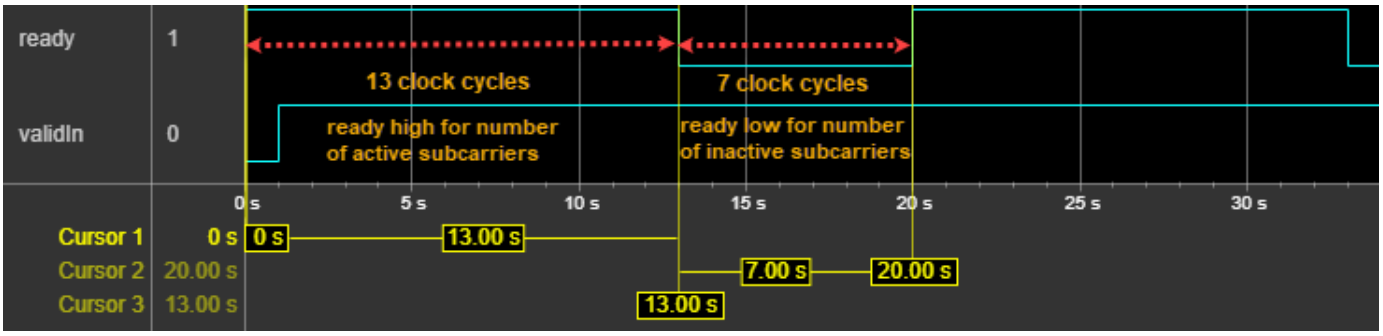
In these equations,

- N_h is the number of high ready clock cycles
- N_l is the number of low ready clock cycles
- *vecLen* is the length of the vector

This figure shows the ready signal generation for the default block configuration (**FFT length** = 64, **Cyclic prefix length** = 16, **Number of left guard subcarriers** = 6, **Number of right guard subcarriers** = 5 and **Insert DC Null** = `on`) with a scalar input.



This figure shows the ready signal generation for the default block configuration (**FFT length** = 64, **Cyclic prefix length** = 16, **Number of left guard subcarriers** = 6, **Number of right guard subcarriers** = 5 and **Insert DC Null** = `on`) with a four-element column vector input.



Symbol Formation

The block stores input valid active subcarrier data, reads it, and forms a symbol of FFT length by placing the data at the center, and guard subcarriers at the edges of the symbol based on the number of left and right guard subcarrier values provided.

Sample Repeater

This block repeats FFT-length number of samples until it forms the maximum FFT length. For this operation, the block buffers the input samples first and then repeats the samples based on the maximum FFT length value. This repetition mechanism helps to avoid scaling at the FFT block input. This block is optional and available only when you set the **OFDM parameters source** parameter to `Input port`. When you set the **OFDM parameters source** parameter to `Property`, the FFT length value provided in the block mask is set as the maximum FFT length. The block does not need to repeat the samples in this context.

For example, if the FFT length is 128 and the maximum FFT length is 2048, each OFDM symbol consists of 128 samples. The block converts these 128 samples to 2048 samples by repeating the 128 samples 16 times. After the block generates 2048 data samples, it sends data and valid input signals to the next block.

IFFT

The IFFT block converts a frequency-domain signal to a time-domain signal. The block supports the FFT length as a power of 2, in the range from 8 to 65, 536.

The **Divide butterfly outputs by two** parameter sets whether the FFT implements an overall 1/*N* scale factor by dividing the output of each butterfly multiplication by two. This adjustment keeps the output of the IFFT in the same amplitude range as its input. When you clear the **Divide butterfly outputs by two** parameter, the block avoids overflow by increasing the word length by 1 bit after each butterfly multiplication.

Time-Domain FFT Shifter

Conventionally, transceivers perform an FFT shift in the frequency domain. However, this method requires memory and introduces latency related to the size of the FFT. Instead, a transceiver can execute the same operation in the time domain by using the frequency shifting property of Fourier transforms. Shifting a function in one domain corresponds to a multiplication by a complex exponential function in the other domain. To reduce hardware resources and latency, this block performs the FFT shift by multiplying the time-domain samples by a complex exponential function.

These equations describe an FFT shift. The equation for an *N*-point FFT is

$$X(k) = F[x(n)] = \sum_{n=0}^{N-1} x(n)e^{-\frac{j2\pi nk}{N}}$$

For an FFT shift of $N/2$ carriers in either direction, substitute $k = k - \frac{N}{2}$, resulting in

$$X(k - \frac{N}{2}) = \sum_{n=0}^{N-1} x(n)e^{-\frac{j2\pi n(k-\frac{N}{2})}{N}}$$

This equation simplifies to

$$X(k - \frac{N}{2}) = \sum_{n=0}^{N-1} e^{j\pi n}x(n)e^{-\frac{j2\pi nk}{N}}$$

Since $\sum_{n=0}^{N-1} x(n)e^{-\frac{j2\pi nk}{N}}$ is equivalent to $F[x(n)]$, and $e^{j\pi} = -1$, this equation simplifies to

$$X(k - \frac{N}{2}) = F[(-1)^n x(n)]$$

The final equation shows that an FFT shift in the time domain simplifies to multiplication by $(-1)^n$. Therefore, the block implements the FFT shift by multiplying the time-domain samples by either +1 or –1.

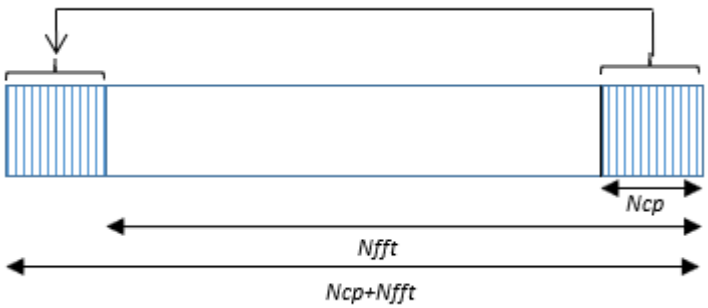
Down Sampler

This block down samples the maximum FFT length number of samples to FFT length number of samples. This block is optional and available only when the **OFDM parameters source** parameter is set to `Input port`. When **OFDM parameters source** is set to `Property`, the FFT length value provided in the block mask is considered as the maximum FFT length. So, there is no need to downsample the samples in this context.

For example, the block is operating with FFT length as 128 and the maximum FFT length is 2048. Here, the input is 2048 samples and it must be downsampled with respective to the FFT length 128. So, the block samples 1 sample for every 16 samples.

CP Addition

Cyclic prefix addition is the process of adding the last samples of an OFDM symbol as a prefix to each OFDM symbol. This figures shows CP addition for an OFDM symbol with *Nfft* samples and CP samples N_{CP}.



When the OFDM Modulator block operates through `Input port`selection, it uses the **Maximum FFT length** parameter to avoid multiple IFFTs.

Windowing

Windowing reduces the spectral regrowth, or adjacent channel leakage ratio (ACLR), of an OFDM signal. Windowing is optional and supports scalar and vector inputs. To enable windowing, select the **Windowing** parameter.

The blocks performs windowing on the CP-added OFDM symbols. For more information about windowing, see the OFDM Modulator Baseband block.

Latency

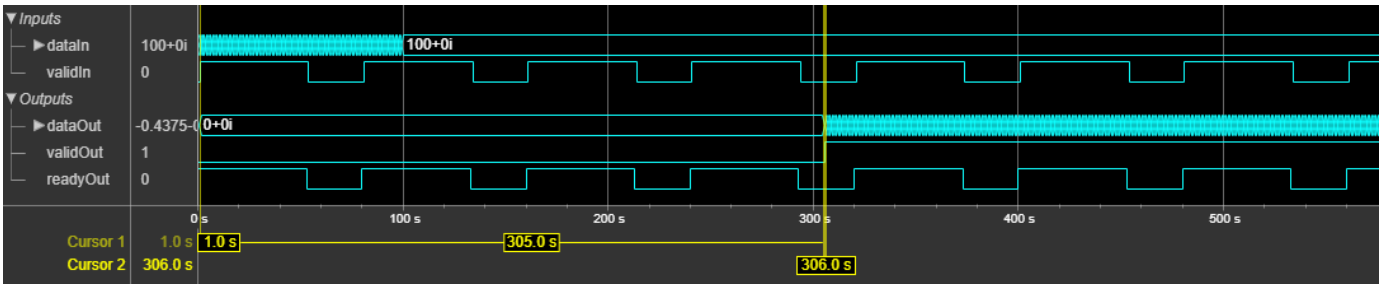
The latency of the block varies with the type of input: scalar or vector.

Scalar Input

This figure shows a sample output and latency of the OFDM Modulator block when you specify a scalar input, set the **OFDM parameters source** parameter to `Property` and use default settings for the other block parameters. **FFT length** is set to `64`, **Cyclic prefix length** is set to `16`, **Insert DC null status** is set to `on`, and **Number of left guard subcarriers** and **Number of right guard subcarriers** are set to `6` and `5`, respectively.

In this example, the latency of the block is calculated using this formula: **FFT length** – (**Number of left guard subcarriers** + **Number of right guard subcarriers** + **Insert DC null status**) + *IFFTLatency* + **FFT length** + 22, where *IFFTLatency* is the latency of IFFT block for the specified FFT length, and 22 is the number of pipeline delays.

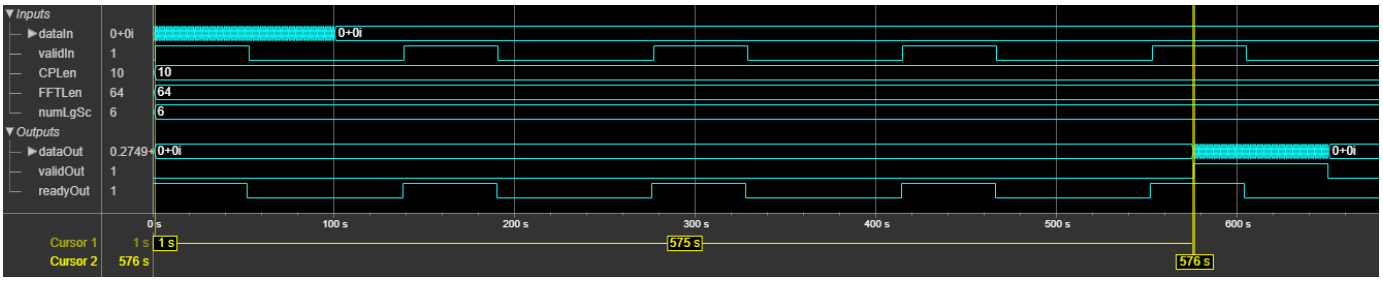
This calculation shows that the latency of the block is 311 clock cycles, as shown in this figure.



This figure shows a sample output and latency of the block when you specify a scalar input and set the **OFDM parameters source** parameter to `Input port`. For this example, **FFTLen** is set to `64`, **CPLen** is set to `16`, **Insert DC null status** is set to `on`, **numLgSc** and **numRgSc** are set to `6` and `5`, respectively, and **Maximum FFT length** is set to `128`.

In this example, the latency of the block is calculated using this formula: **FFTLen** – (**numLgSc** + **numRgSc** + **Insert DC null status**) + **FFT length** + *IFFTLatency* + **Maximum FFT length** + (**Maximum FFT length**/**FFTLen** – 1) + 32, where *IFFTLatency* is the latency of IFFT block for the specified maximum FFT length, and 32 is the number of pipeline delays.

This calculation shows that the latency of the block is 582 clock cycles, as shown in this figure.



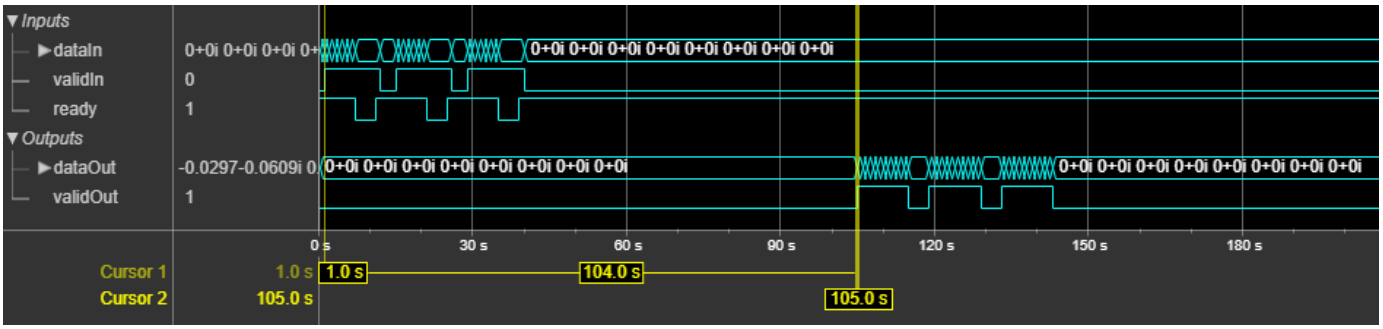
The block accepts input only when the **ready** signal is 1 (high). In this case, the block captures parameters on the first cycle when the input **valid** signal is 1 (high).

Vector Input

This figure shows a sample output and latency of the OFDM Modulator block when you specify an eight-element column vector input, set the **OFDM parameters source** parameter to `Property` and use default settings for the other block parameters. **FFT length** is set to 64, **Cyclic prefix length** is set to 16, **Insert DC null status** is set to `on`, and **Number of left guard subcarriers** and **Number of right guard subcarriers** are set to 6 and 5, respectively.

In this example, the latency of the block is calculated using this formula: `ceil`((**FFT length** – (**Number of left guard subcarriers** + **Number of right guard subcarriers** + **Insert DC null status**))/*vecLen*) + *vecIFFTLatency* + `ceil`(**FFT length**/*vecLen*) + 22, where *vecIFFTLatency* is the latency of IFFT block for the specified FFT length and vector length, *vecLen* is the length of the vector, and 22 is the number of pipeline delays.

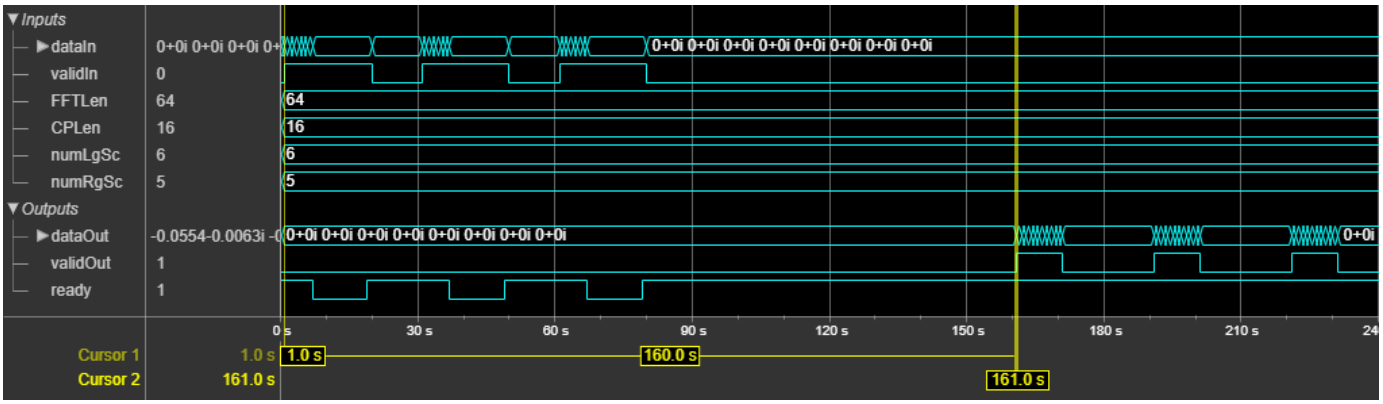
This calculation shows that the latency of the block is 104 clock cycles, as shown in this figure.



This figure shows a sample output and latency of the OFDM Modulator block when you specify an eight-element column vector input and set the **OFDM parameters source** parameter to `Input port`. For this example, **FFTLen** is set to 64, **CPLen** is set to 16, **Insert DC null status** is set to `on`, **numLgSc** and **numRgSc** are set to 6 and 5, respectively, and **Maximum FFT length** is set to 128.

In this example, the latency of the block is calculated using this formula: `ceil`((**FFTLen** – (**numLgSc** + **numRgSc** + **Insert DC null status**))/*vecLen*) + **FFTLen**/*vecLen* + *vecIFFTLatency* + `floor`((**Maximum FFT length**/**FFTLen**) * (*vecLen* – 1)/*vecLen*) + `ceil`(**Maximum FFT length**/*vecLen*) – (**Maximum FFT length**/**FFTLen** – 1) + 32, where *vecIFFTLatency* is the latency of IFFT block for the specified maximum FFT length and vector length, *vecLen* is the length of the vector, and 32 is the number of pipeline delays.

This calculation shows that the latency of the block is 160 clock cycles, as shown in this figure.



The block accepts input only when the **ready** signal is 1 (high). In this case, the block captures parameters on the first cycle when the input **valid** signal is 1 (high).

Performance

The performance of the synthesized HDL code varies with your target and synthesis options. The input data type used in this example for generating HDL code is `fixdt(1,16,14)`.

This table shows the resource and performance data synthesis results when using the block with a default configuration for a scalar input and an eight-element column vector input. The generated HDL is targeted to the Xilinx Zynq- 7000 ZC706 evaluation board.

Input Data	Slice LUTs	Slice Registers	DSPs	Block RAMs	Maximum Frequency in MHz
Scalar	2389	4103	8	3	263.4
Vector	12311	21705	56	16	236.3

References

[1] 3GPP TS 36.211 version 14.2.0 Release 14. "Physical channels and modulation." *LTE - Evolved Universal Terrestrial Radio Access (E-UTRA)*.

[2] "Wireless LAN Medium Access Control (MAC) and Physical layer (PHY) Specifications." IEEE Std 802.11 – 2012.

[3] Stefania Sesia, Issam Toufik, and Matthew baker. *LTE - THE UMTS Long Term Evolution from theory to practice*.

[4] Erik Dahlman, Stefan Parkvall, and Johan Skold. *4G - LTE/LTE - Advanced for Mobile broadband Second edition*.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

This block does not have any HDL Block Properties.

See Also

Blocks

OFDM Modulator Baseband | OFDM Demodulator

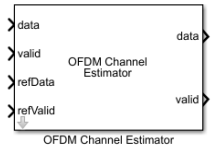
Objects

`comm.OFDMModulator`

Introduced in R2020a

OFDM Channel Estimator

Estimate channel using input data and reference subcarriers
Library: Wireless HDL Toolbox / Modulation



Description

The OFDM Channel Estimator block estimates a channel using input data and reference subcarriers. The block accepts data subcarriers, a **valid** control port, and **refData** and **refValid** reference ports. The block outputs channel estimates and a **valid** control port. The block allows you to specify the number of subcarriers to estimate for each output symbol.

You can use this block to estimate multipath faded channels on the receiver side in different communications standards, such as long term evolution (LTE) [1] and wireless local area network (WLAN) [4]. To perform proper channel estimation, the **refData** and **refValid** ports must be synchronized with the **data** and **valid** ports, respectively. For more information about channel estimation and reference data, see “Channel Estimation” (LTE Toolbox).

This block provides an interface and architecture suitable for HDL code generation and hardware deployment.

Ports

Input

`data` — Input data subcarriers

scalar

Input data subcarriers, specified as a scalar of real or complex values.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `signed fixed point`

`valid` — Indicates valid input data

scalar

Indicates valid input data, specified as a scalar.

This port is a control signal that indicates when the sample from the **data** input port is valid. When this value is `1`, the block captures the values from the **data** input port. When this value is `0`, the block ignores the values from the **data** input port.

Data Types: `Boolean`

`refData` — Reference data subcarriers

scalar

Reference data subcarriers, specified as a scalar of real or complex values.

Reference data must be a sequence of unimodular values. In a sequence of values, r_1, r_2, r_3..., r_n, the values are unimodular if r_j x r_{j^*} = 1,

where:

- j = 1, 2, 3, ..., n.
- r_{j^*} is the complex conjugate of r_j.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `signed fixed point`

`refValid` — Indicates valid reference data
scalar

Indicates valid reference data, specified as a scalar.

This port is a control signal that indicates when the sample from the **refData** input port is valid. When this value is `1`, the block captures the values from the **refData** input port. When this value is `0`, the block ignores the values from the **refData** input port.

refValid port values must be synchronized with **valid** port values.

Data Types: `Boolean`

`numScPerSym` — Number of valid subcarriers per OFDM symbol
scalar

Number of valid subcarriers per OFDM symbol, specified as a scalar in the range from 2 to 65,536.

To support the minimum number of subcarriers per symbol, **numScPerSym** must be of data type `fixdt(0,k,0)`, where k is greater than or equal to 2.

Dependencies

To enable this port, select the **Enable averaging** parameter or **Enable interpolation** parameter.

Data Types: `uint8` | `uint16` | `uint32` | `unsigned fixed point`

`reset` — Clear internal states
scalar

Clear internal states, specified as a scalar. When this value is `1`, the block stops the current calculation and clears all internal states.

Dependencies

To enable this port, select the **Enable reset input port** parameter.

Data Types: `Boolean`

Output

`data` — Output channel estimates
scalar

Output channel estimates, returned as a scalar. The output data type is the same as the input data.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `signed fixed point`

`valid` — Indicates valid output data
scalar

Indicates valid output data, returned as a scalar.

This port is a control signal that indicates when the **data** output port is valid. The block sets this value to 1 when the data samples are available from the **data** output port.

Data Types: `Boolean`

Parameters

`Enable averaging` — Average LS estimates
`off` (default) | `on`

Select this parameter to enable averaging.

`Number of symbols to be averaged` — Number of symbols to be averaged
2 (default) | integer in range from 2 to 14

Specify the number of symbols to be averaged.

Dependencies

To enable this parameter, select the **Enable averaging** parameter.

`Enable interpolation` — Interpolate LS estimates
`off` (default) | `on`

Select this parameter to enable interpolation.

`Interpolation factor` — Interpolation factor
3 (default) | integer in range from 2 to 12

Specify the interpolation factor.

Dependencies

To enable this parameter, select the **Enable interpolation** parameter.

`Maximum number of subcarriers per symbol` — Maximum number of subcarriers per symbol
52 (default) | integer in range from 2 to 65, 536

Specify the maximum number of subcarriers per symbol.

Dependencies

To enable this parameter, select the **Enable averaging** parameter or **Enable interpolation** parameter.

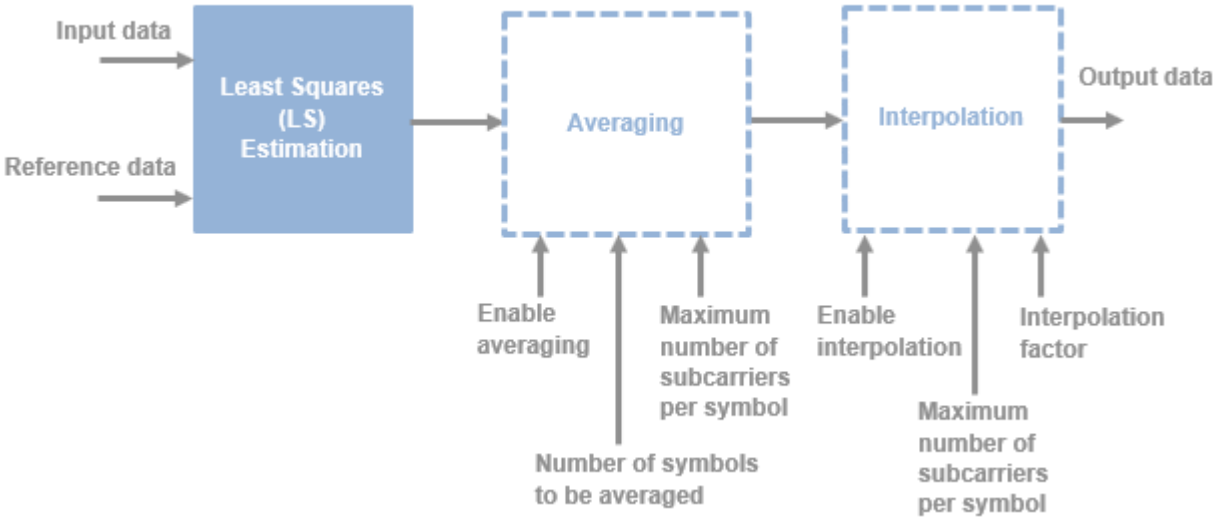
`Enable reset input port` — Reset signal
`off` (default) | `on`

Select this parameter to enable the **reset** port on the block icon.

Algorithms

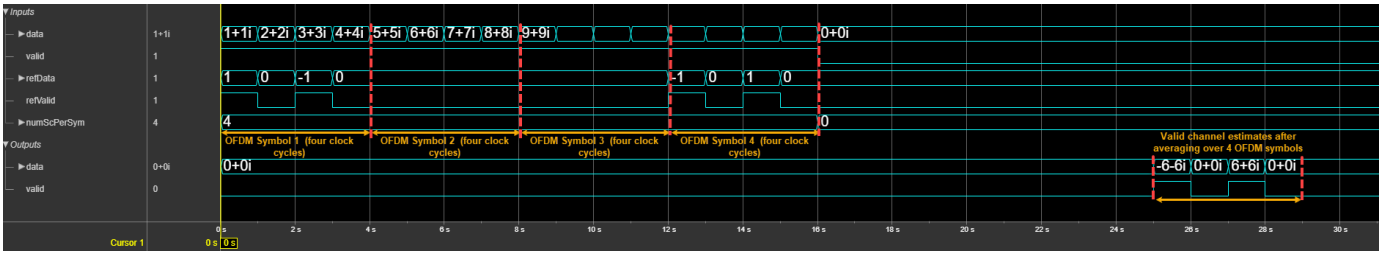
This figure shows the architecture block diagram of the OFDM Channel Estimator block. The block implements least squares (LS) estimation for the channel estimation. To improve the accuracy of LS estimation, the block uses an averaging technique and provides an interpolation feature if the number of known reference signals are limited to certain subcarriers for a particular OFDM symbol. The Least Squares (LS) Estimation block calculates the least-squares estimates using the input data and the reference data.

The Averaging and Interpolation blocks are optional. To perform averaging, select the **Enable averaging** parameter. To perform interpolation, select the **Enable interpolation** parameter. The parameters shown in this figure configure the behavior of the block.



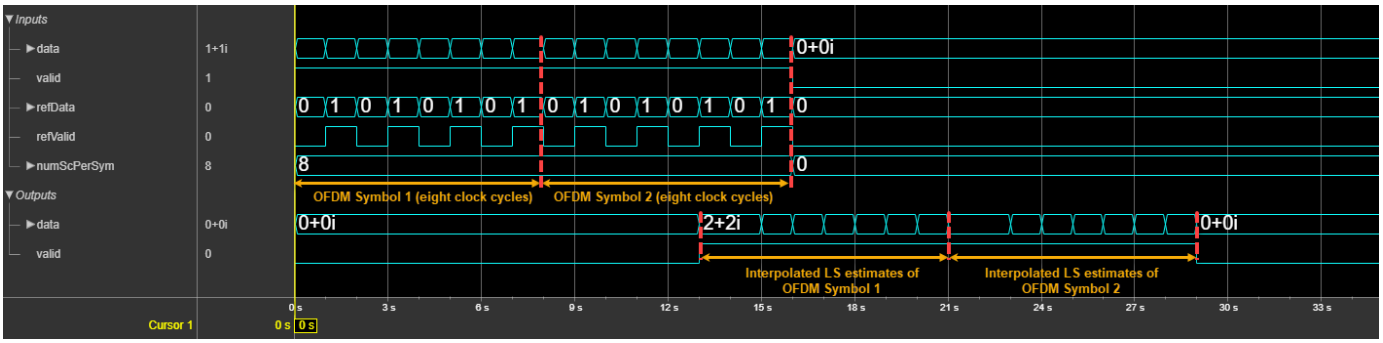
The Averaging block accepts the LS estimates and averages the corresponding subcarriers with valid LS estimates over the number of OFDM symbols to be averaged provided in the block mask. This figure shows a sample output of the OFDM Channel Estimator block when only averaging is enabled. In this case, the **Number of symbols to be averaged** parameter is set to 4, **Maximum number of subcarriers per symbol** parameter is set to 16, and the **numScPerSym** port is set to 4.

The block samples the **numScPerSym** port value at the first valid clock cycle. After that, the block samples this value at the every first valid clock cycle, after completing the valid number of **Number of symbols to be averaged** x **numScPerSym** clock cycles. As the number of OFDM symbols to be averaged is 4, the output valid shows the valid channel estimates obtained by averaging over four OFDM symbols.



The Interpolation block accepts the LS estimates and performs linear interpolation to calculate the missing channel information between two consecutive valid LS estimates. This figure shows a sample output of the OFDM Channel Estimator block when only interpolation is enabled. In this case, the **Interpolation factor** parameter is set to 2, **Maximum number of subcarriers per symbol** parameter is set to 16, and the **numScPerSym** port is set to 8.

The block samples the **numScPerSym** port value at the first valid clock cycle. After that, the block samples this value at the every first valid clock cycle, after completing the valid number of subcarriers per symbol clock cycles. The output valid shows the interpolated LS estimates for two OFDM symbols.

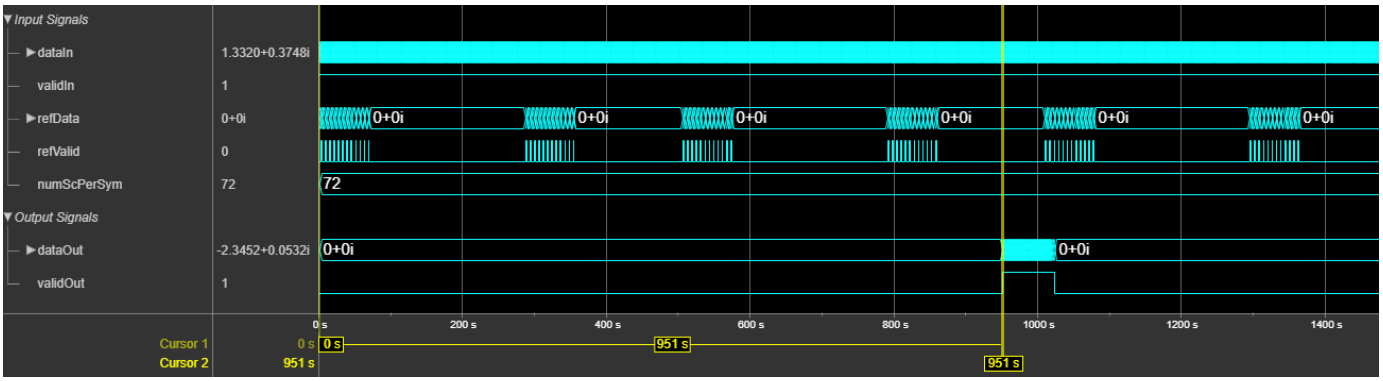


Latency

The latency of the block varies with the block parameter values and **numScPerSym** port values. This table provides the latency calculations of the block for different conditions.

Enable Averaging Value	Enable Interpolation Value	Latency Value (in Clock Cycles)
`Off`	`Off`	12
`On`	`Off`	[(**Number of symbols to be averaged** – 1) x **numScPerSym**] + 13
`Off`	`On`	**Interpolation factor** + 11
`On`	`On`	[(**Number of symbols to be averaged** – 1) x **numScPerSym**] + **Interpolation factor** + 12

This figure shows a sample output of the OFDM Channel Estimator block in an LTE standard configuration. In this case, the **Number of symbols to be averaged** parameter is set to 14, **Interpolation factor** parameter is set to 3, **Maximum number of subcarriers per symbol** parameter is set to 72, and **numScPerSym** port is set to 72. The latency of the block is 951 clock cycles.



Performance

This table shows the resource and performance data synthesis results of the block when you set the **Number of symbols to be averaged** parameter to 14, **Interpolation factor** parameter to 3, **Maximum number of subcarriers per symbol** parameter to 180, and **numScPerSym** port to 180. The input data provided is of data type `fixdt(1,16,13)`. The generated HDL is targeted to the Xilinx Zynq- 7000 ZC706 evaluation board. The design achieves a clock frequency of 244.6 MHz.

Resource	Number Used
Slice LUTs	2684
Slice Registers	1184
DSPs	6
Block RAMs	1.5

References

[1] 3GPP TS 36.211 version 14.2.0 Release 14. "Physical channels and modulation." *LTE - Evolved Universal Terrestrial Radio Access (E-UTRA)*.

[2] Sesia, Stefania, Issam Toufik, and Matthew Baker, eds. *LTE - The UMTS Long Term Evolution: From Theory to Practice*. Chichester, UK: John Wiley & Sons, Ltd, 2011. https://doi.org/10.1002/9780470978504.

[3] Dahlman, Erik, Stefan Parkvall, and Johan Sköld. *4G LTE/LTE-Advanced for Mobile Broadband*. Second edition. Amsterdam ; New York: Elsevier, 2014.

[4] "Wireless LAN Medium Access Control (MAC) and Physical layer (PHY) Specifications." IEEE Std 802.11 – 2012.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

See Also

Blocks

OFDM Channel Estimator | OFDM Equalizer

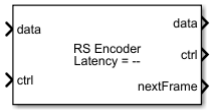
Functions

`lteDLChannelEstimate` | `nrChannelEstimate` | `wlanLLTFChannelEstimate`

Introduced in R2020a

RS Encoder

Encode message to RS codeword
Library: Wireless HDL Toolbox / Error Detection and Correction



Description

The RS Encoder block encodes message data to a Reed-Solomon (RS) codeword. The block accepts message data and a `samplecontrol` bus and outputs codeword data symbols and a `samplecontrol` bus.

Because the latency of the block varies, the block provides output port **nextFrame** that indicates when the block is ready to accept new input message data. The block provides an architecture suitable for HDL code generation and hardware deployment and supports shortened message lengths.

You can use this block to model many communication system forward error correcting (FEC) codes. The block supports digital subscriber line (DSL), WiMAX (802.16 m and e), digital video broadcast handheld (DVB-H) terminals, digital video broadcast satellite (DVB-S) services, and digital video broadcast satellite services to handheld (DVB-SH) devices below 3 MHz.

Ports

Input

`data` — Input message data
scalar

Input message data, specified as a scalar representing one symbol.

The input word length must be an unsigned integer equal to `ceil`($\log_2$(**Codeword length (N)**)). For an input data word length of 3, the codeword length parameter, **Codeword length (N)**, must be 7.

`double` and `single` data types are allowed for simulation, but not for HDL code generation.

Data Types: `double` | `single` | `uint8` | `uint16` | `fixed point`

`ctrl` — Control signals accompanying sample stream
`samplecontrol` bus

Control signals accompanying the sample stream, specified as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the input frame
- `end` — Indicates the end of the input frame
- `valid` — Indicates that the data on the input **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

Output

`data` — Encoded codeword data
scalar

Encoded codeword data, returned as a scalar. This output data width is same as the input data width.

Data Types: `double` | `single` | `uint8` | `uint16` | `fixed point`

`ctrl` — Control signals accompanying sample stream
`samplecontrol` bus

Control signals accompanying the sample stream, returned as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the output frame
- `end` — Indicates the end of the output frame
- `valid` — Indicates that the data on the output **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

`nextFrame` — Block ready indicator
scalar

Block ready indicator, returned as a scalar.

The block sets this signal to `1` (`true`) when the block is ready to accept the start of the next frame. If the block receives an input **ctrl.start** signal while **nextFrame** is `0` (`false`), the block discards the frame in progress and begins processing the new data.

Data Types: `Boolean`

Parameters

`Codeword length (N)` — Length of codeword
7 (default) | integer in the range from 7 to 65,535

Specify the codeword length.

The codeword length must be an integer equal to 2^M – 1, where M is an integer in the range from 3 to 16. For more information on representing data for RS codes, see “Integer Format (Reed-Solomon Only)”.

`Message length (K)` — Length of message
3 (default) | integer in the range from 3 to (**Codeword length (N)** – 2)

Specify the message length.

For more information on representing data for RS codes, see “Integer Format (Reed-Solomon Only)”.

`Source of primitive polynomial` — Primitive polynomial source
`Auto` (default) | `Property`

Specify the source of the primitive polynomial.

- Select `Auto` to specify the primitive polynomial based on the **Codeword length (N)** parameter value. The degree of the primitive polynomial is calculated as M = `ceil`($\log_2$(**Codeword length (N)**)).
- Select `Property` to specify the primitive polynomial using the **Primitive polynomial** parameter.

`Primitive polynomial` — Primitive polynomial
`[1 0 1 1]` (default) | binary row vector

Specify a binary row vector representing the primitive polynomial in descending order of powers.

For more information on how to specify a primitive polynomial, see “Primitive Polynomials and Element Representations”.

Dependencies

To enable this parameter, set the **Source of primitive polynomial** parameter to `Property`.

`Source of B, the starting power for roots of the primitive polynomial` — Source of starting power for roots of primitive polynomial
`Auto` (default) | `Property`

Specify the source of the starting power for roots of the primitive polynomial.

- Select `Auto` to use the default **B value** parameter value, `1`.
- Select `Property` to enable the **B value** parameter.

`B value` — Starting power for roots of primitive polynomial
`1` (default) | positive integer

Specify the starting power for roots of the primitive polynomial.

Dependencies

To enable this parameter, set the **Source of B, the starting power for roots of the primitive polynomial** parameter to `Property`.

`Enable puncturing` — Puncture pattern source
`off` (default) | `on`

Select this parameter to enable the **Puncture pattern vector** parameter.

`Puncture pattern vector` — Puncture vector
`[1; 1; 0; 0]` (default) | binary column vector

Specify a binary column vector of length **Codeword length (N)** – **Message length (N)**. A value of `1` indicates that the block data symbol is not punctured, and remained unchanged from the data stream. A value of `0` indicates that the data symbol is punctured, or removed, from the data stream.

Dependencies

To enable this parameter, select the **Enable puncturing** parameter.

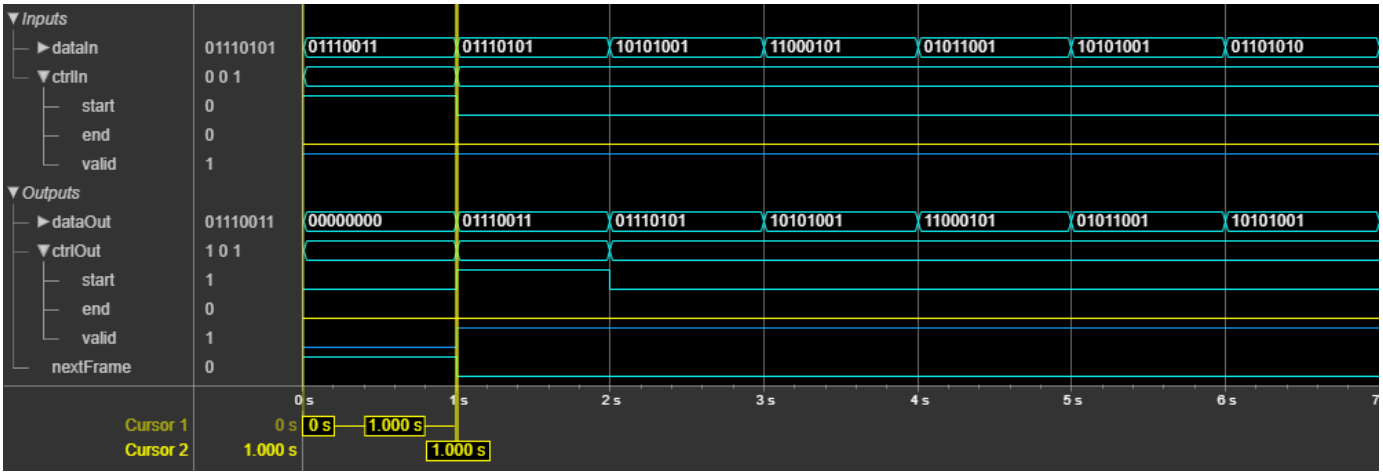
Algorithms

The RS Encoder block encodes a message data of length K into an RS codeword of length N. The block requires a minimum gap of $N – K$ clock cycles to add $N – K$ parity length to the message data of length K. During these $N – K$ parity length clock cycles, the block does not accept new data. So, the minimum duration between messages must be $N – K$ clock cycles.

- Every `start` signal that is high indicates the start of a new message. When multiple `start` high signals exist, the block accepts only the latest `start` signal.
- `start` and `end` high signals are valid only when the `valid` signal of the block is high.
- The block accepts `end` signals with the corresponding `start` signal. In case of multiple `end` high signals, the block accepts only the first `end` high signal and ignores the remaining `end` high signals.

Latency

This figure shows a sample output of the RS Encoder block with latency according to the DVB-S standard configuration, **Codeword length (N)** and **Message length (K)** parameter values specified as `255` and `239`, respectively, and with puncturing disabled. In this case, the latency of the block is 1 clock cycle.



Performance

The performance of the synthesized HDL code varies with your target and synthesis options. The input data type used for generating HDL code is `fixdt(0,8,0)`.

This table shows the resource and performance data synthesis results when using the block with **Codeword length (N)** and **Message length (K)** parameter values specified as `255` and `239`, respectively. The generated HDL is targeted to the Xilinx Zynq- 7000 ZC706 evaluation board. The design achieves a clock frequency of 440 MHz.

Resource	Number Used
LUTs	237
Registers	154
DSPs	0

Resource	Number Used
Block RAMs	7.5

References

[1] Wicker, Stephen B. *Error Control Systems for Digital Communication and Storage*. Englewood Cliffs, NJ: Prentice Hall, 1995.

[2] Clark, George C., and J. Bibb Cain. *Error-Correction Coding for Digital Communications*. Applications of Communications Theory. New York: Plenum Press, 1981.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

See Also

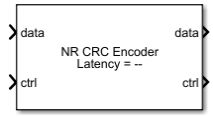
Blocks

Integer-Input RS Encoder | Integer-Output RS Decoder | RS Decoder

Introduced in R2020b

NR CRC Encoder

Generate CRC code bits and append them to input data
Library: Wireless HDL Toolbox / Error Detection and Correction



Description

The NR CRC Encoder block calculates and generates a short, fixed-length binary sequence, known as the cyclic redundancy check (CRC) checksum, appends it to each frame of streaming data samples, and outputs CRC-encoded data. The block accepts and returns a data sample stream with accompanying control signals. The control signals indicate the validity of the samples and the boundaries of the frame.

The block supports scalar and vector inputs and outputs data as either a scalar or vector based on the input data. To achieve higher throughput, the block accepts a binary vector or unsigned integer scalar input and implements a parallel architecture. The input data width must be less than or equal to the length of the CRC polynomial and the length of the CRC polynomial, must be divisible by the input data width. The block supports all CRC polynomials specified according to the 5G new radio (NR) standard 3GPP TS 38.212 [1]. When you select the CRC24C polynomial, the block supports dynamic CRC mask.

The block provides an interface and hardware-optimized architecture suitable for HDL code generation and hardware deployment.

Ports

Input

`data` — Input data

binary scalar | binary vector | unsigned integer scalar

Input data, specified as a binary scalar, binary vector, or unsigned integer scalar.

You can specify the input data with one of these options:

- Scalar – Specify an integer representing several bits. For this case, the block supports unsigned integer (`uint8`, `uint16`, or `ufix`*N*) and `Boolean` data types.
- Vector – Specify a vector of binary values of size *N*. For this case, the block supports `Boolean` and `ufix1` data.

N is the input data width, and it must be less than or equal to the length of the CRC polynomial and a factor of the specified CRC polynomial length.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Example: For the CRC type `CRC24A`, the valid data widths are 24, 12, 8, 6, 4, 3, 2, and 1. An integer input is interpreted as a binary word. For example, when you specify a `uint8` input of `19`, it is equivalent to a vector input `[0 0 0 1 0 0 1 1]`.

Data Types: `double` | `single` | `ufix1` | `uint8` | `uint16` | `Boolean` | `ufixN`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, specified as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the input frame
- `end` — Indicates the end of the input frame
- `valid` — Indicates that the data on the input **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

`CRCMask` — CRC checksum mask

nonnegative integer

CRC checksum mask, specified as a nonnegative integer representing a binary word from 0 to $2^{CRCLength} - 1$, where *CRCLength* is the length of the CRC polynomial.

This mask is typically a radio network temporary identifier (RNTI). The RNTI is used to XOR the CRC checksum.

Dependencies

To enable this port, set the **CRC type** parameter to `CRC24C` and select the **Enable CRC mask input port** parameter.

Data Types: `ufix24`

Output

`data` — CRC-encoded data

scalar | vector

CRC-encoded data with appended CRC checksum, returned as a scalar or vector. The output data type and size are the same as the input data.

Data Types: `double` | `single` | `ufix1` | `uint8` | `uint16` | `Boolean` | `ufixN`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, returned as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the output frame
- `end` — Indicates the end of the output frame

- `valid` — Indicates that the data on the output **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

Parameters

`CRC type` — Type of CRC

`CRC16` (default) | `CRC6` | `CRC11` | `CRC24A` | `CRC24B` | `CRC24C`

Select the type of CRC. Each CRC type indicates a polynomial, as shown in this table.

CRC Type	Polynomial
`CRC6`	[1 1 0 0 0 0 1]
`CRC11`	[1 1 1 0 0 0 1 0 0 0 0 1]
`CRC16`	[1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1]
`CRC24A`	[1 1 0 0 0 0 1 1 0 0 1 0 0 1 1 0 0 1 1 1 1 1 0 1 1]
`CRC24B`	[1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1]
`CRC24C`	[1 1 0 1 1 0 0 1 0 1 0 1 1 0 0 0 1 0 0 0 1 0 1 1 1]

These CRC polynomials are specified according to the 5G NR standard 3GPP TS 38.212 [1].

`Enable CRC mask input port` — Enable CRC checksum mask input port

`off` (default) | `on`

Select this parameter to enable the **CRCMask** input port.

Dependencies

To enable this parameter, set the **CRC type** parameter to `CRC24C`.

Algorithms

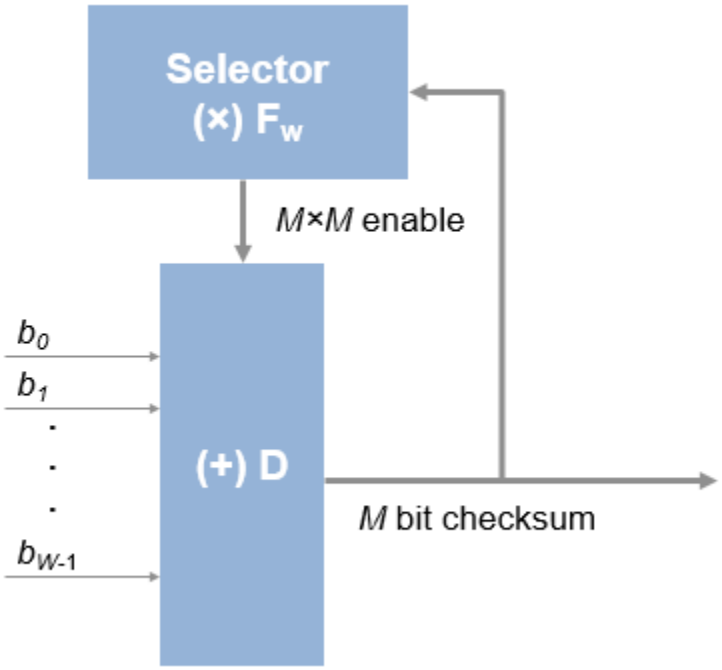
When you use a binary vector or unsigned integer scalar input, the block implements a parallel CRC algorithm [2].

To provide high throughput for modern communications systems, the block implements the CRC algorithm with a parallel architecture. This architecture recursively calculates M bits of a CRC checksum for each W input bits. At the end of the frame, the final checksum result is appended to the message. For a polynomial length of M, the recursive checksum calculation for W bits in parallel is

$$X' = F_W(\times)X(+)D$$

F_W is an M-by-M matrix that selects elements of the current state for the polynomial calculation with the new input bits. D is an M-element vector that provides the new input bits, ordered in relation to

the generator polynomial and padded with zeros. The block implements the (×) with logical AND and (+) with logical XOR.



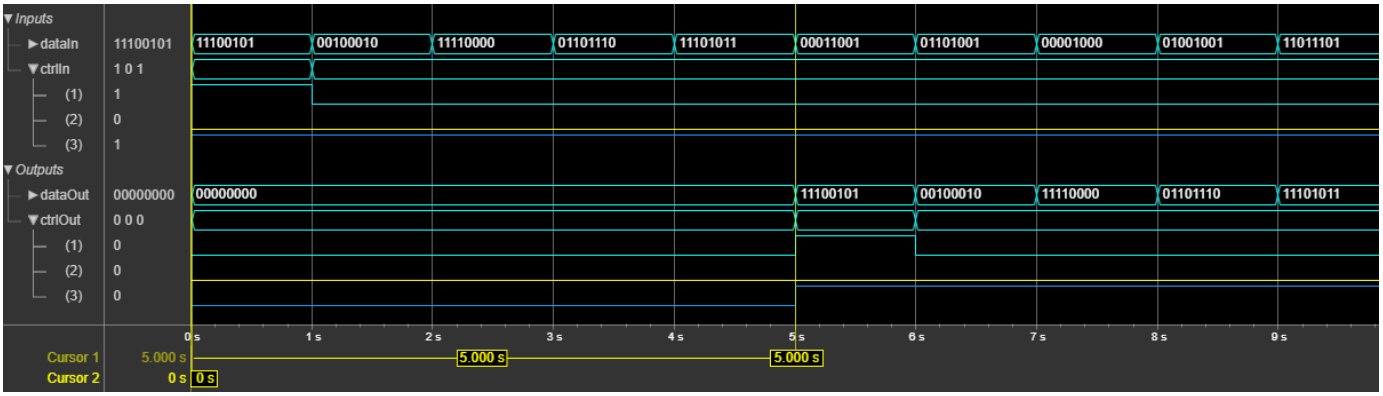
Latency

The latency of the block varies with the CRC polynomial length, the type of input (scalar or vector), and the data width of the input. The latency of the block is calculated from the start of the input frame to the start of the output frame by using the formula (*CRCLength*/*N*) + 3 clock cycles, where *N* is the input data width.

The frame gap between two frames (that is, from **ctrl.end** of the first frame to **ctrl.start** of the next frame) must be greater than the length of the CRC polynomial plus the latency of the first frame. In case of continuous inputs, the block discards the first frame and starts processing the next frame.

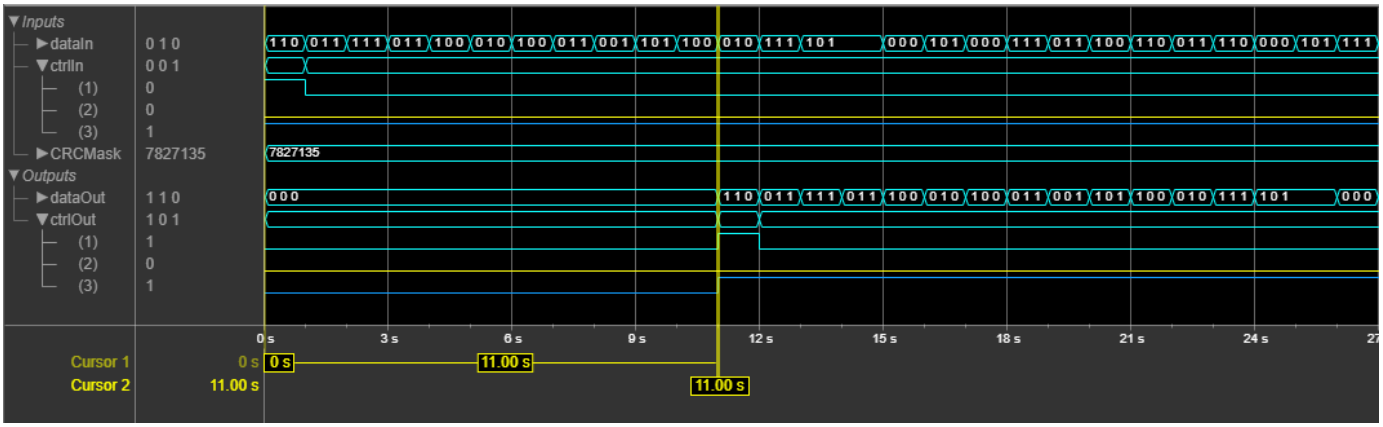
Scalar Input

This figure shows a sample output and latency of the NR CRC Encoder block when you specify a scalar input of data type `ufix8` with a data width of 8, and set the **CRC type** parameter to `CRC16`. The latency of the block is 5 clock cycles, as shown in this figure.



Vector Input

This figure shows a sample output and latency of the NR CRC Encoder block when you specify a vector input of data type `ufix1` with a data width 3, set the **CRC type** parameter to `CRC24C`, and select the **Enable CRC mask input port** parameter. The latency of the block is 11 clock cycles, as shown in this figure.



Performance

The performance of the synthesized HDL code varies with your target and synthesis options. It also varies based on the CRC polynomial length and the input data width.

This table shows the resource and performance data synthesis results of the block when the **CRC type** parameter is set to `CRC24A` for a scalar input of data type `ufix1`, scalar input of data type `ufix24`, and 24-by-1 vector input of data type `ufix1`. The generated HDL is targeted to the Xilinx Zynq- 7000 ZC706 evaluation board.

Input Data		Slice LUTs	Slice Registers	Block RAMs	Maximum Frequency in MHz
Scalar	`ufix1`	147	168	0	614.6
	`ufix24`	207	192	0	581.0
Vector		182	160	0	571.1

References

[1] 3GPP TS 38.212. “NR; Multiplexing and Channel Coding.” *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*.

[2] Campobello, G., G. Patane, and M. Russo. “Parallel CRC Realization.” *IEEE Transactions on Computers* 52, no. 10 (October 2003): 1312–19. https://doi.org/10.1109/TC.2003.1234528.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

See Also

Blocks

NR CRC Decoder

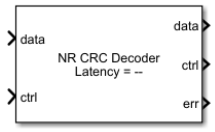
Functions

`nrCRCDecode` | `nrCRCEncode`

Introduced in R2021a

NR CRC Decoder

Detect errors in input data using CRC
Library: Wireless HDL Toolbox / Error Detection and Correction



Description

The NR CRC Decoder block calculates the cyclic redundancy check (CRC) checksum and compares it with the appended CRC checksum for each frame of streaming data samples. If the two CRC checksums do not match, the block reports an error. The block accepts and returns a data sample stream with accompanying control signals. The control signals indicate the validity of the samples and the boundaries of the frame.

The block supports scalar and vector inputs and outputs data as either a scalar or vector based on the input data. To achieve higher throughput, the block accepts a binary vector or unsigned integer scalar input and implements a parallel architecture. The input data width must be less than or equal to the length of the CRC polynomial, and the length of the CRC polynomial must be divisible by the input data width. The block supports all CRC polynomials specified according to the 5G new radio (NR) standard 3GPP TS 38.212 [1]. When you select the CRC24C polynomial, the block supports dynamic CRC mask.

The block provides an interface and hardware-optimized architecture suitable for HDL code generation and hardware deployment.

Ports

Input

`data` — CRC-encoded data

binary scalar | binary vector | unsigned integer scalar

CRC-encoded data, specified as a binary scalar, binary vector, or unsigned integer scalar.

You can specify the input data with one of these options:

- Scalar – Specify an integer representing several bits. For this case, the block supports unsigned integer (`uint8`, `uint16`, or `ufix`*N*) and `Boolean` data types.
- Vector – Specify a vector of binary values of size *N*. For this case, the block supports `Boolean` and `ufix1` data types.

N is the input data width, and it must be less than or equal to the length of the CRC polynomial and a factor of the specified CRC polynomial length.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Example: For the CRC type `CRC24A`, the valid data widths are 24, 12, 8, 6, 4, 3, 2, and 1. An integer input is interpreted as a binary word. For example, when you specify a `uint8` input of `19`, it is equivalent to a vector input `[0 0 0 1 0 0 1 1]`.

Data Types: `double` | `single` | `ufix1` | `uint8` | `uint16` | `Boolean` | `ufixN`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, specified as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the input frame
- `end` — Indicates the end of the input frame
- `valid` — Indicates that the data on the input **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

`CRCMask` — CRC checksum mask

nonnegative integer

CRC checksum mask, specified as a nonnegative integer representing a binary word from 0 to $2^{CRCLength} - 1$, where *CRCLength* is the length of the CRC polynomial.

This mask is typically a radio network temporary identifier (RNTI). The RNTI is used to XOR the CRC checksum.

Dependencies

To enable this port, set the **CRC type** parameter to `CRC24C port` and select the **Enable CRC mask input port** parameter.

Data Types: `ufix24`

Output

`data` — CRC-decoded data

scalar | vector

CRC-decoded data, returned as a scalar or vector. The output data type and size are the same as the input data.

Data Types: `double` | `single` | `ufix1` | `uint8` | `uint16` | `Boolean` | `ufixN`

`ctrl` — Control signals accompanying sample stream

`samplecontrol` bus

Control signals accompanying the sample stream, returned as a `samplecontrol` bus. The bus includes the `start`, `end`, and `valid` control signals, which indicate the boundaries of the frame and the validity of the samples.

- `start` — Indicates the start of the output frame
- `end` — Indicates the end of the output frame

- `valid` — Indicates that the data on the output **data** port is valid

For more detail, see “Sample Control Bus”.

Data Types: `bus`

`err` — Indication of corruption of received data
binary or integer scalar

Indication of corruption of the received data, returned as a binary or integer scalar.

When this value is `1`, the message contains at least one error. When this value is `0`, the message contains zero errors. This value is valid when **ctrl.end** is set 1 `true`.

If you select the **Full checksum mismatch** parameter, this port returns the integer XOR result of the checksum mismatch. This value is the result of the logical CRC difference between the CRC checksum comprised in the input and the CRC checksum recalculated across the data part of the input. If you specify a CRC mask, the block XORs the checksum with the specified CRC mask.

Data Types: `Boolean` | `ufix24`

Parameters

`CRC type` — Type of CRC
`CRC16` (default) | `CRC6` | `CRC11` | `CRC24A` | `CRC24B` | `CRC24C`

Select the CRC type. Each CRC type indicates a polynomial, as shown in this table.

CRC Type	Polynomial
CRC6	[1 1 0 0 0 0 1]
CRC11	[1 1 1 0 0 0 1 0 0 0 0 1]
CRC16	[1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1]
CRC24A	[1 1 0 0 0 0 1 1 0 0 1 0 0 1 1 0 0 1 1 1 1 1 0 1 1]
CRC24B	[1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1]
CRC24C	[1 1 0 1 1 0 0 1 0 1 0 1 1 0 0 0 1 0 0 0 1 0 1 1 1]

These CRC polynomials are specified according to 5G new radio (NR) standard 3GPP TS 38.212 [1].

`Full checksum mismatch` — Enable full checksum
`off` (default) | `on`

Select this parameter to enable full checksum.

When you select this parameter, the **err** port returns an integer that represents the locations of bit mismatches in the CRC checksum bits. When you clear this parameter, the **err** port returns a Boolean value indicating whether any CRC checksum bits are mismatched.

`Enable CRC mask input port` — Enable CRC checksum mask input port
`off` (default) | `on`

Select this parameter to enable the **CRCMask** input port.

Dependencies

To enable this parameter, set the **CRC type** parameter to `CRC24C`.

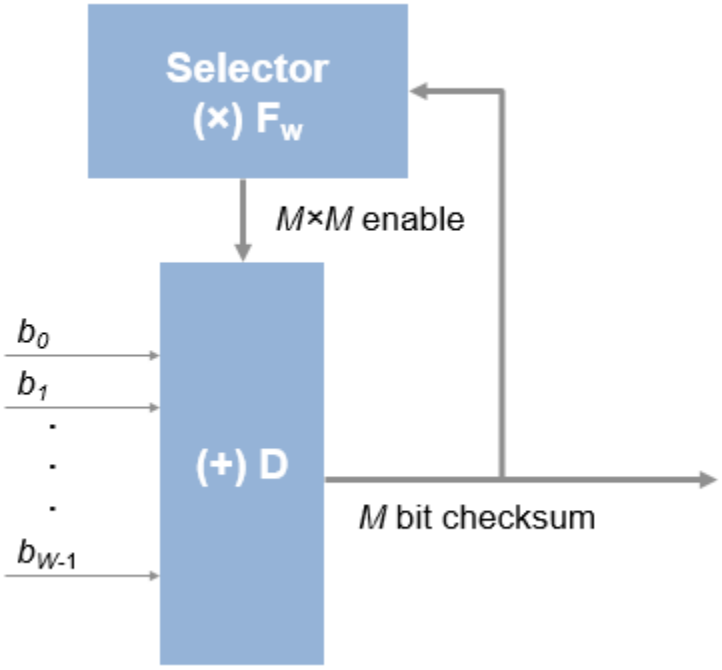
Algorithms

When you use a binary vector or unsigned integer scalar input, the block implements a parallel CRC algorithm [2].

To provide high throughput for modern communications systems, the block implements the CRC algorithm with a parallel architecture. This architecture recursively calculates M bits of a CRC checksum for each W input bits. At the end of the frame, the final checksum result is appended to the message. For a polynomial length of M, the recursive checksum calculation for W bits in parallel is

$$X' = F_W(\times)X(+)D$$

F_W is an M-by-M matrix that selects elements of the current state for the polynomial calculation with the new input bits. D is an M-element vector that provides the new input bits, ordered in relation to the generator polynomial and padded with zeros. The block implements the (×) with logical AND and (+) with logical XOR.



Latency

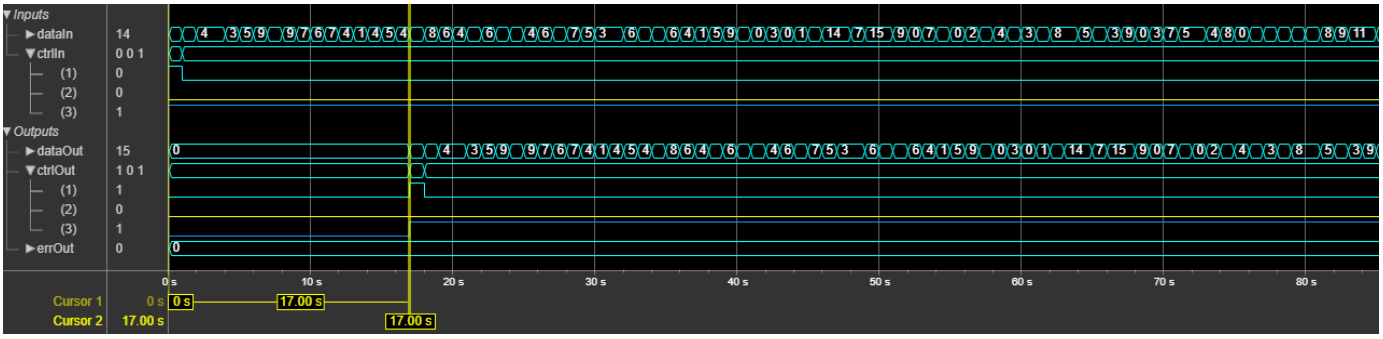
The latency of the block varies with the CRC polynomial length, the type of input (scalar or vector), and the data width of the input. The latency of the block is calculated from the start of the input

frame to the start of output frame by using the formula ((*CRCLength*/*N*) x 3) + 5 clock cycles, where *N* is the input data width.

The frame gap between two frames (that is from **ctrl.end** of the first frame to **ctrl.start** of the next frame) must be greater than the length of the CRC polynomial plus the latency of the first frame. In case of continuous inputs, the block discards the first frame and starts processing the next frame.

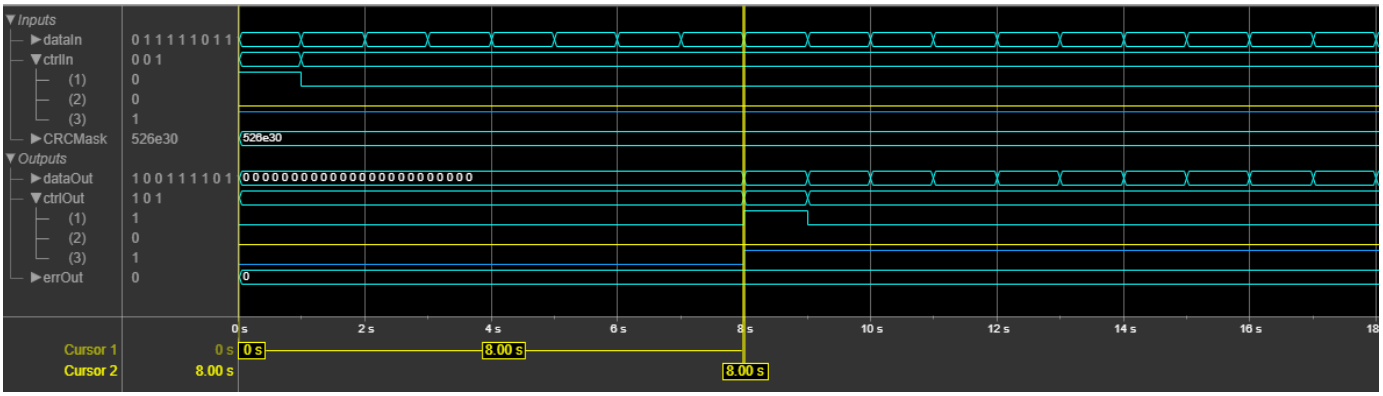
Scalar Input

This figure shows a sample output and latency of the NR CRC Decoder block when you specify a scalar input of data type `ufix4` with a data width of 4, and set the **CRC type** parameter to `CRC16`. The latency of the block is 17 clock cycles, as shown in this figure.



Vector Input

This figure shows a sample output and latency of the NR CRC Decoder block when you specify a vector input of data type `ufix1` with a data width of 24, set the **CRC type** parameter to `CRC24C`, and select the **Enable CRC mask input port** parameter. The latency of the block is 8 clock cycles, as shown in this figure.



Performance

The performance of the synthesized HDL code varies with your target and synthesis options. It also varies based on the CRC polynomial length and the input data width.

This table shows the resource and performance data synthesis results of the block when the **CRC type** parameter is set to `CRC24A` for a scalar input of data type `ufix1`, , scalar input of data type `ufix24`, and for a 24-by-1 vector input of data type `ufix1`. The generated HDL is targeted to the Xilinx Zynq- 7000 ZC706 evaluation board.

Input Data		Slice LUTs	Slice Registers	Block RAMs	Maximum Frequency in MHz
Scalar	`ufix1`	246	345	0	581.4
	`ufix24`	403	483	0	553.1
Vector		361	458	0	526.6

References

[1] 3GPP TS 38.212. “NR; Multiplexing and Channel Coding.” *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*.

[2] Campobello, G., G. Patane, and M. Russo. “Parallel CRC Realization.” *IEEE Transactions on Computers* 52, no. 10 (October 2003): 1312–19. https://doi.org/10.1109/TC.2003.1234528.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

See Also

Blocks

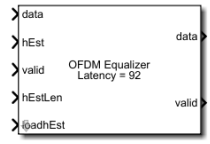
NR CRC Encoder

Functions
nrCRCDecode | nrCRCEncode

Introduced in R2021a

OFDM Equalizer

Equalize OFDM data using channel estimates
Library: Wireless HDL Toolbox / Modulation



Description

The OFDM Equalizer block equalizes the OFDM data using channel estimates. The block supports zero-forcing (ZF) and minimum mean square error (MMSE) algorithms for channel equalization in the frequency domain. The block accepts data symbols, estimated channel (**hEst**), and the estimated channel length per symbol (**hEstLen**) data ports and **valid** and **loadhEst** control ports. The block outputs an equalized data port and a **valid** control port.

You can use this block to equalize channel effects in different communications standards, such as long term evolution (LTE) [1], 5G new radio (NR) standard TS 38.212 [2], and wireless local area network (WLAN) [3].

The block provides an interface and architecture suitable for HDL code generation and hardware deployment.

Ports

Input

`data` — OFDM data
scalar

OFDM data, specified as a complex-valued scalar.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `signed fixed point`

`hEst` — Channel estimated data
scalar

Channel estimated data, specified as a complex-valued scalar.

The input data type must be `fixdt(1,k,m)`, where k is less than or equal to 30, and m is less than k.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `single` | `double` | `int8` | `int16` | `fixed point`

`valid` — Control to indicate valid input data
scalar

Control to indicate valid input data, specified as a Boolean scalar.

This port is a control signal that indicates when the input **data** and **hEst** port values are valid. When this value is `1`, the block captures the values from the **data** and **hEst** input ports. When this value is `0`, the block ignores the values on the **data** and **hEst** input ports.

Data Types: `Boolean`

`hEstLen` — Length of estimated channel per symbol

scalar

Length of the estimated channel per symbol, specified as a scalar in the range from 2 to 65,536.

To support the minimum number of subcarriers per symbol, this data type must be `fixdt(0,k,0)`, where k is greater than or equal to 2.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

`loadhEst` — Channel estimates control

scalar

Channel estimates control, specified as a Boolean scalar.

When this value is `1`, the block loads the channel estimates until the length of the channel estimate specified by the **hEstLen** input port. **hEstLen** is sampled at **loadhEst**.

When this value is `0`, and the input is `1`, the block performs equalization with the previously sampled **hEstLen** input and the stored **hEst** input values. If the previously sampled **hEstLen** value is not available, the block performs equalization with the instantaneous inputs **data**, **hEst**, and **nVar**. For more information, see “Algorithms” on page 1-233.

Data Types: `Boolean`

`nVar` — Noise variance

scalar

Noise variance, specified as a scalar.

When the input **valid** is `1`, the block samples the **nVar** port. This value must be of data type `fixdt(0,k,m)`, where k is less than or equal to 16, and m is less than or equal to k.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Dependencies

To enable this port, set the **Equalization method** parameter to `MMSE`.

Data Types: `single` | `double` | `uint8` | `uint16` | `fixed point`

`reset` — Reset internal states

scalar

Reset internal states of the block to start with new data, specified as a scalar.

Dependencies

To enable this port, select the **Enable reset input port** parameter.

Data Types: `Boolean`

Output

`data` — Complex output data
scalar

Complex output data, returned as a scalar. The output data type is the same as the input data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `signed fixed point`

`valid` — Indicates valid output data
scalar

Indicates valid output data, returned as a Boolean scalar.

This port is a control signal that indicates when the **data** output port is valid. When the data samples are available on the **data** output port, the block sets this output **valid** value to `1`.

Data Types: `Boolean`

Parameters

`Equalization method` — Equalization method
`ZF` (default) | `MMSE`

Select the equalization method. For more information about the equalization methods, see “Algorithms” on page 1-233.

`Maximum length of channel estimate per symbol` — Maximum length of channel estimate per symbol
52 (default) | integer in the range from 2 to 65, 536

Specify the maximum length of the channel estimate per symbol.

To support the minimum number of subcarriers per symbol, which is 2, the data type of the **hEstLen** input must be `fixdt(0,k,0)`, where k is greater than or equal to 2.

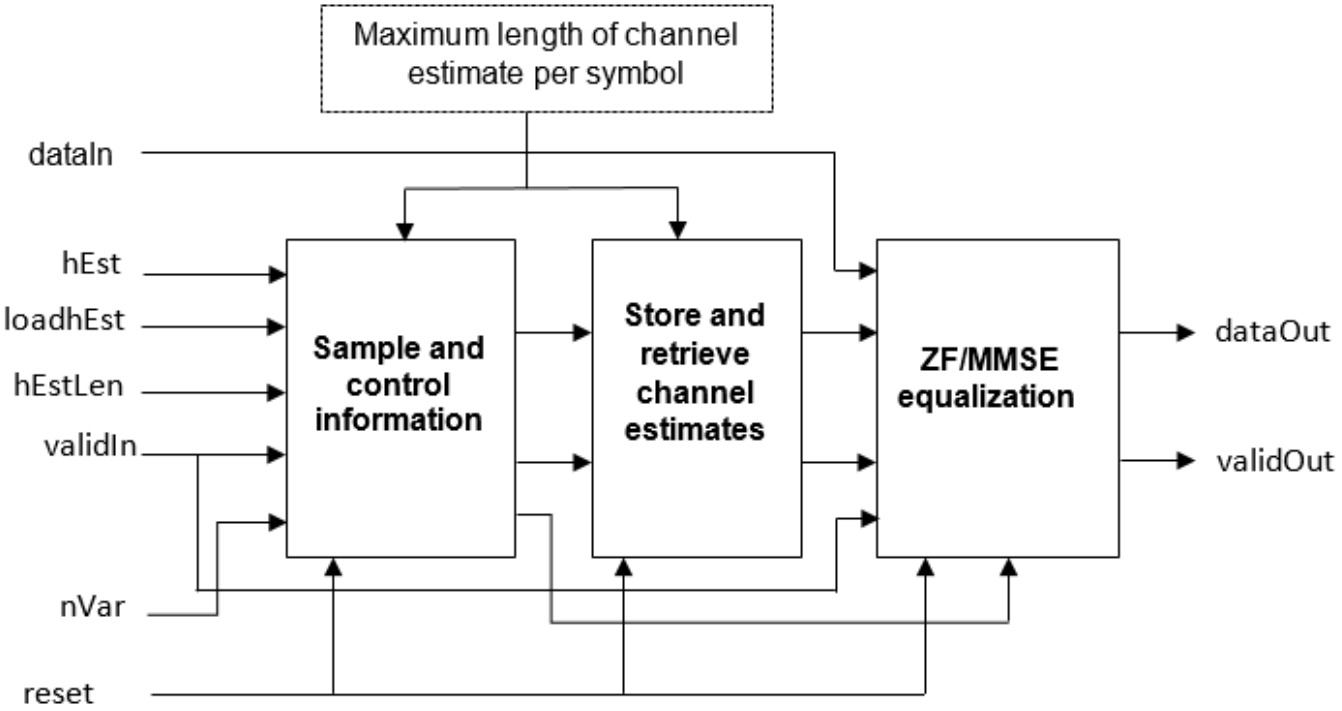
`Enable reset input port` — Reset signal
`off` (default) | `on`

Select this parameter to enable the **reset** input port.

Algorithms

The OFDM Equalizer block supports ZF and MMSE algorithms for channel equalization in the frequency domain. The block stores the estimated channel information to equalize the OFDM symbols and generates the equalized output using these algorithms.

The OFDM Equalizer block operation sequence is implemented using these subsystem blocks: Sample and control information, Store and retrieve channel estimates, and ZF/MMSE equalization. This figure shows these blocks.



The Sample and control information block samples and validates the **hEstLen** input based on the **loadhEst** input signal, validates the **hEst** and **nVar** inputs based on the **validIn** input signal, and outputs the sampled **hEstOut** output, **nVarOut** output, and the control information signals that are used in storing and retrieving channel information. The Store and retrieve channel estimates block stores and retrieves the channel using RAM and switches. The ZF/MMSE Equalization block performs ZF or MMSE equalization using these equations. The **nVar** input port is available when you set the **Equalization method** parameter to `MMSE`.

- **ZF Algorithm**:

  $$dataOut_p = (hEst_p^* \times dataIn_p) / |hEst_p|^2$$

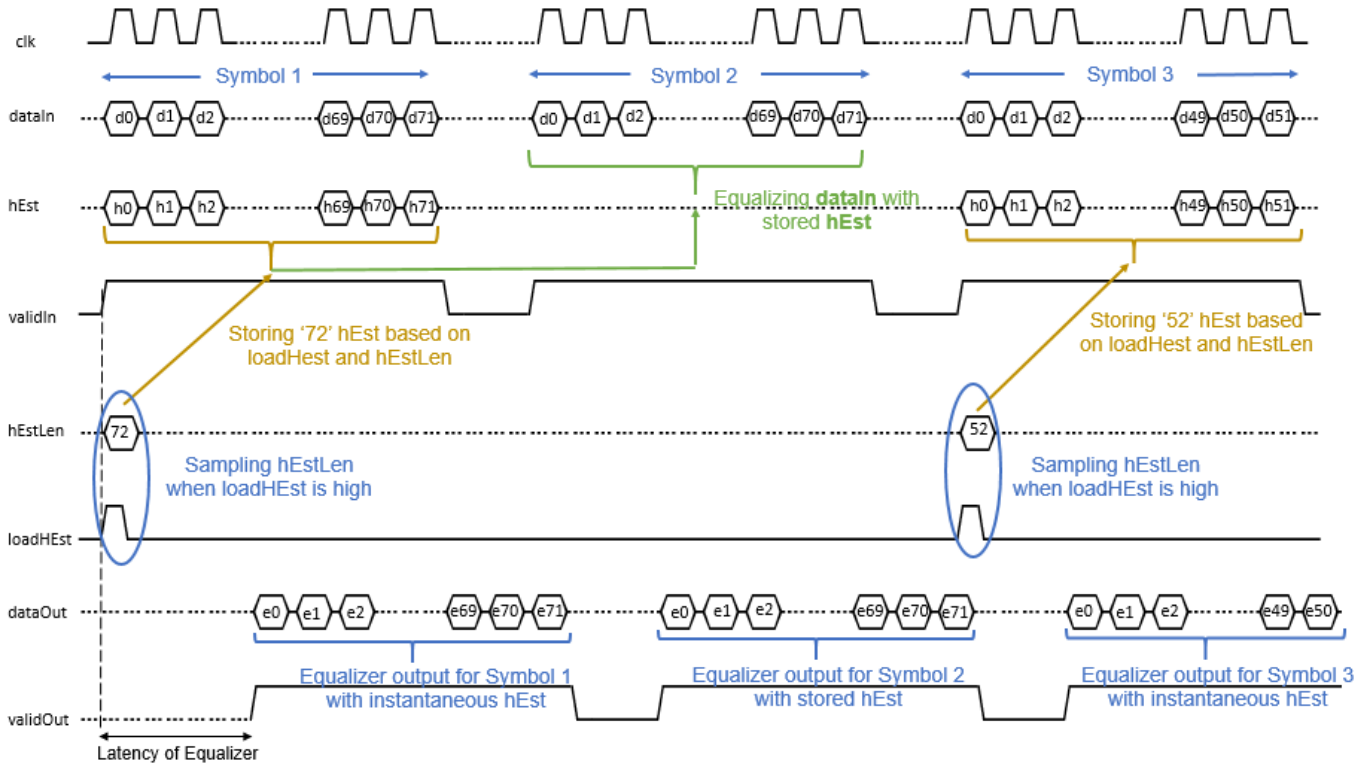
- **MMSE Algorithm**:

  $$dataOut_p = (hEst_p^* \times dataIn_p) / (|hEst_p|^2 + nVar_p)$$

In these equations,

- *dataIn* is the demodulated output provided as an input to the block
- *hEst* is the estimated channel
- $hEst^*$ is the Hermitian of the estimated channel
- *dataOut* is the equalized output
- *nVar* is the noise variance
- *p* is equal to 0, 1, ... *NSPS*, where *NSPS* is the number of subcarriers per symbol.

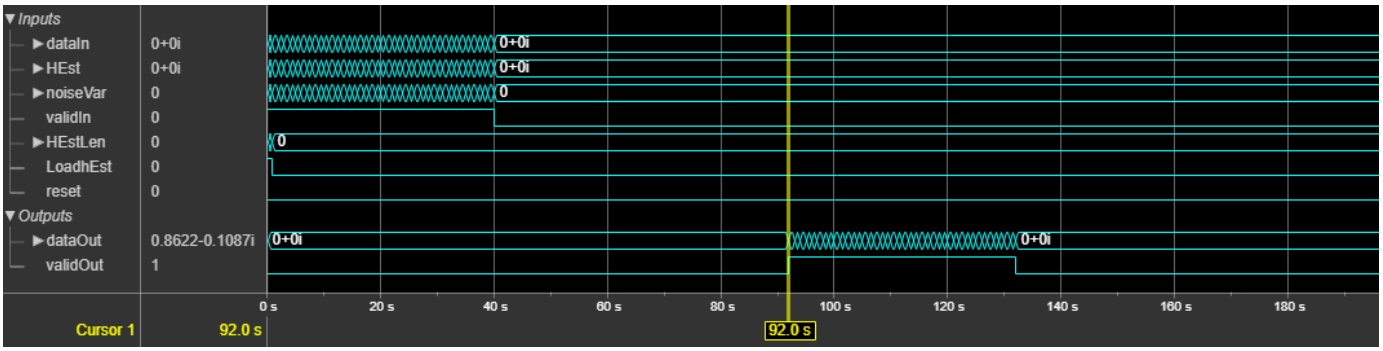
This figure shows a sample block operation when you set the **Equalization method** parameter to `ZF`.



In this figure, you can see three symbols (Symbol 1, Symbol 2, and Symbol 3) are input to the **dataIn** port. When the **validIn** input is 1 (high) and the **loadHest** input is 1 (high), the block samples the **hEstlLen** input value, which is 72 in this example. Based on the **hEstlLen** value, for Symbol 1, the block provides the equalized output for the instantaneous **hEst** input values. When the **loadHest** value changes to `0` (low), the block stores the **hEst** values and provides the equalized output for Symbol 2 based on the stored **hEst** values. The **hEstLen** value remains the same until the **loadHest** changes to `0` (low). Similarly, for Symbol 3, the block provides the equalized output for the instantaneous **hEst** values based on the **hEstlLen** value, which is 52 in this example.

Latency

This figure shows a sample output of the OFDM Equalizer block when you set the **Equalization method** parameter to `MMSE` and the **Maximum length of channel estimate per symbol** parameter to 52. The latency of the block is 92 clock cycles.



Performance

The performance of the synthesized HDL code varies with your target and synthesis options.

This table shows the resource and performance data synthesis results of the block when you set the **Equalization method** parameter to `MMSE`, the **Maximum length of channel estimate per symbol** parameter to `52`, and the **hEstLen** port to `20`. The input data is of data type `fixdt(1,28,16)`. The generated HDL is targeted to the Xilinx Zynq- 7000 ZC706 evaluation board. The design achieves a clock frequency of 244.6 MHz.

Resource	Number Used
Slice LUTs	7380
Slice Registers	8063
DSPs	24
Block RAMs	0

References

[1] 3GPP TS 36.211 version 14.2.0 Release 14. "Physical channels and modulation." *LTE - Evolved Universal Terrestrial Radio Access (E-UTRA)*.

[2] 3GPP TS 38.212. “NR; Multiplexing and Channel Coding.” *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*.

[3] "Wireless LAN Medium Access Control (MAC) and Physical layer (PHY) Specifications." IEEE Std 802.11 – 2012.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).

InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

See Also

Blocks
OFDM Channel Estimator

Functions
`lteEqualizeMMSE` | `lteEqualizeZF` | `nrEqualizeMMSE`

Introduced in R2021a

Functions

whdlFramesToSamples

Convert frame-based data to sample stream

Syntax

```
[samples,ctrl,len] = whdlFramesToSamples(frames)
[samples,ctrl,len] = whdlFramesToSamples(frames,postsampleidles,
postframeidles)
[samples,ctrl,len] = whdlFramesToSamples(frames,postsampleidles,
postframeidles,samplesize)
[samples,ctrl,len] = whdlFramesToSamples(frames,postsampleidles,
postframeidles,samplesize,interleaved)
```

Description

`[samples,ctrl,len] = whdlFramesToSamples(frames)` serializes frame-based data into a stream of samples and accompanying control signals. The control signals indicate the validity of the samples and the boundaries of the frames. The function also returns a vector, `len`, of the frame size corresponding to each sample.

`[samples,ctrl,len] = whdlFramesToSamples(frames,postsampleidles, postframeidles)` inserts idle cycles in the sample stream, `samples`. Specify the number of idle cycles to insert between input samples, `postsampleidles`, and the number of idle cycles between frames, `postframeidles`.

`[samples,ctrl,len] = whdlFramesToSamples(frames,postsampleidles, postframeidles,samplesize)` creates a sample stream where each sample is represented by `samplesize` values. The function inserts `samplesize` zeros for each idle cycle requested. The `ctrl` and `len` vectors are the same size as when `samplesize` is 1.

`[samples,ctrl,len] = whdlFramesToSamples(frames,postsampleidles, postframeidles,samplesize,interleaved)` orders the sample stream, assuming the input samples are interleaved, when `interleaved` is 1 (`true`). The `interleaved` argument is valid only when `samplesize` is greater than 1.

Examples

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`.

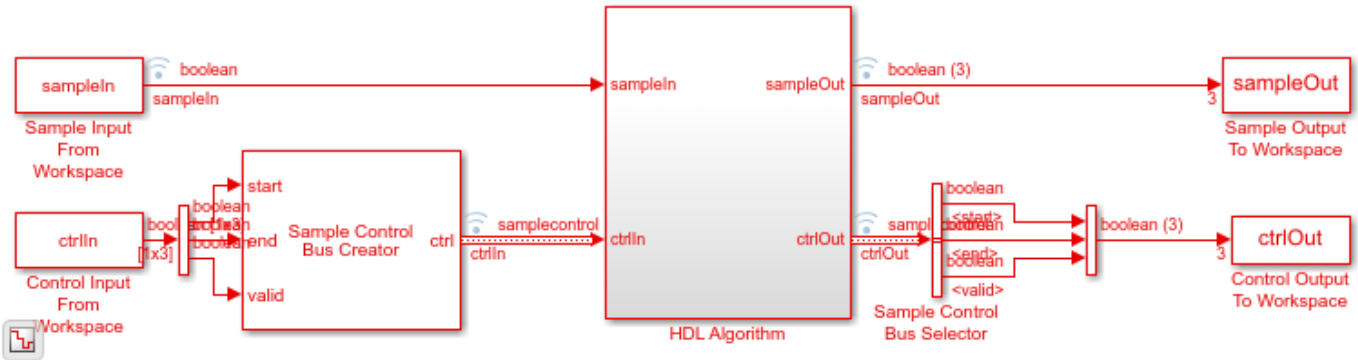
```
rng(0);
turboframesize = 40;
numframes = 2;

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);
```



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
```

Input Arguments

`frames` — Frames of input samples

column vector | cell array of column vectors

Frames of input samples, specified as a column vector or a cell array of column vectors. The frames in the cell array can be different sizes.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

`postsampleidles` — Number of idle cycles to insert between samples

0 (default) | integer

Number of idle cycles to insert between samples, specified as an integer. The function inserts `samplesize` zeros for each idle cycle, and sets all control signals to 0 (`false`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`postframeidles` — Number of idle cycles to insert between frames

0 (default) | integer

Number of idle cycles to insert between frames, specified as an integer. The function inserts `samplesize` zeros for each idle cycle, and sets all control signals to 0 (`false`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`samplesize` — Number of values representing each sample
1 (default) | positive integer

Number of values representing each sample, specified as a positive integer. The function returns one set of control signals for each `samplesize` values.

For example, in the LTE standard, the turbo code rate is 1/3, so each turbo-encoded sample is represented by one systematic, and two parity values: S_n, P_n1, and P_n2. In this case, set `samplesize` to 3.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`interleaved` — Order of output samples relative to input order
0 (default) | logical scalar

Order of output samples relative to input order, when more than one value represents each sample, specified as a logical scalar.

For example, for 1/3 turbo-encoded samples, the input frame can be ordered `[S_1 P1_1 P2_1 S_2 P1_2 P2_2]` or `[S_1 S_2 P1_1 P1_2 P2_1 P2_2]`. In the first case, the default output would be the same order as the input. To achieve that output order for the second input, set `interleaved` to 1 (`true`).

Data Types: `logical`

Output Arguments

`samples` — Stream of samples
column vector

Stream of samples, returned as a column vector. For N samples in an input frame, the output is N + `samplesize`×(N×`idlecyclesbetweensamples` + `idlecyclesbetweenframes`) values per frame.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

`ctrl` — Control signals accompanying sample stream
M-by-3 matrix

Control signals accompanying sample stream, returned as an M-by-3 matrix. The matrix includes three control signals, `start`, `end`, and `valid`, for each `samplesize` elements in `samples`. For N input samples in F frames, M is N + N×`idlecyclesbetweensamples` + F×`idlecyclesbetweenframes`. When you import this variable into Simulink, use a Sample Control Bus Creator block to convert the signals into the bus type used by the Wireless HDL Toolbox blocks.

Data Types: `logical`

`len` — Frame length
column vector of integers

Frame length, returned as a column vector of integers. This value is the number of valid samples in the corresponding frame for each `samplesize` elements in `samples`. This vector is the same length as `ctrl`.

Data Types: `double`

See Also

Blocks
Frame To Samples | Samples To Frame

Functions
`whdlSamplesToFrames`

Topics
“Streaming Sample Interface”

Introduced in R2017b

whdlSamplesToFrames

Convert sample stream to frame-based data

Syntax

```
outframes = whdlSamplesToFrames(samples,ctrl)
outframes = whdlSamplesToFrames(samples,ctrl,maxlen)
outframes = whdlSamplesToFrames(samples,ctrl,maxlen,interleaved)
```

Description

`outframes = whdlSamplesToFrames(samples,ctrl)` composes frame-based data from a sample stream and corresponding control signals. The control signals indicate the validity of the samples and the boundaries of the frames. The function calculates the maximum frame length from the input data and control signals, and removes any idle or nonvalid samples from the data.

`outframes = whdlSamplesToFrames(samples,ctrl,maxlen)` composes frame-based data, using the maximum frame length. If an input frame described by `samples` is larger than `maxlen`, the function truncates the frame.

`outframes = whdlSamplesToFrames(samples,ctrl,maxlen,interleaved)` orders the frame-based data, assuming the input samples are interleaved, when `interleaved` is 1 (`true`). The `interleaved` argument is valid only when each sample is represented by multiple values. The function computes the number of values representing each sample by comparing the length of `samples` and `ctrl`.

Examples

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;
```

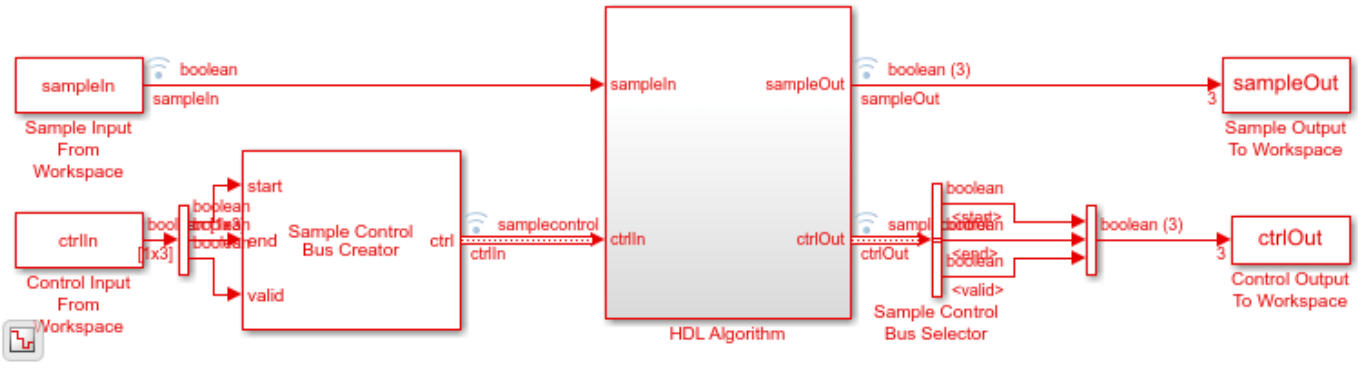
```
txBits    = cell(1,numframes);
codedData = cell(1,numframes);

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);
```



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
```

Input Arguments

`samples` — Stream of samples

column vector

Stream of output samples, specified as a column vector. The vector can include idle cycles between samples and between frames. Idle cycles are discarded. The frames represented by the stream can be different sizes. The vector length, *N*, must be an integer multiple of the length of the `ctrl` matrix, *M*. Differing lengths mean that each sample is represented by *N*/*M* values.

For example, in the LTE standard, the turbo code rate is 1/3, so each turbo-encoded sample is represented by one systematic, and two parity values: S_n, P_n1, and P_n2. In that case, the length of `samples` must be three times the length of `ctrl`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

`ctrl` — Control signals accompanying sample stream

M-by-3 matrix

Control signals accompanying the sample stream, specified as an *M*-by-3 matrix. The matrix includes three control signals, `start`, `end`, and `valid`, for each sample in `samples`. Each sample can be represented by more than one value. In that case, the length of `samples` must be an integer multiple of *M*.

For example, in the LTE standard, the turbo code rate is 1/3, so each turbo-encoded sample is represented by one systematic, and two parity values: S_n, P_n1, and P_n2. In that case, the length of `samples` must be three times the length of `ctrl`.

Data Types: `logical`

`maxlen` — Maximum frame length

integer

Maximum frame length, specified as an integer. The input frames in `samples` can be different sizes. The output column vector reflects the size of the input frame, according to `ctrl`. If a frame is larger than `maxlen`, the function truncates the frame and returns a warning message.

Data Types: `double`

`interleaved` — Order of output samples relative to input order
0 (default) | logical scalar

Order of output samples relative to input order, when more than one value represents each sample, specified as a logical scalar.

For example, 1/3 turbo-encoded samples are represented by $[S_1\ P_11\ P_12\ S_2\ P_21\ P_22]$. To reorder the samples so that systematic and parity values are grouped together, set `interleaved` to 1 (`true`). The output order is then $[S_1\ S_2\ P_11\ P_21\ P_12\ P_22]$.

Data Types: `logical`

Output Arguments

`outframes` — Frames of output samples
column vector | cell array of column vectors

Frames of output samples, returned as a column vector or a cell array of column vectors. The size of the output column vector reflects the size of the input frame, as determined by the control signals in `ctrl`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

See Also

Blocks
Frame To Samples | Samples To Frame

Functions
`whdlFramesToSamples`

Topics
“Verify Turbo Decoder with Streaming Data from MATLAB”
“Streaming Sample Interface”

Introduced in R2017b

ltehdlFramesToSamples

(Removed) Convert frame-based data to sample stream

Note `ltehdlFramesToSamples` name is changed to `whdlFramesToSamples`. Retained this file as part of compatibility consideration process. This function will be removed soon. See “Compatibility Considerations”.

Syntax

```
[samples,ctrl,len] = ltehdlFramesToSamples(frames)
[samples,ctrl,len] = ltehdlFramesToSamples(frames,postsampleidles,
postframeidles)
[samples,ctrl,len] = ltehdlFramesToSamples(frames,postsampleidles,
postframeidles,samplesize)
[samples,ctrl,len] = ltehdlFramesToSamples(frames,postsampleidles,
postframeidles,samplesize,interleaved)
```

Description

`[samples,ctrl,len] = ltehdlFramesToSamples(frames)` serializes frame-based data into a stream of samples and accompanying control signals. The control signals indicate the validity of the samples and the boundaries of the frames. The function also returns a vector, `len`, of the frame size corresponding to each sample.

`[samples,ctrl,len] = ltehdlFramesToSamples(frames,postsampleidles, postframeidles)` inserts idle cycles in the sample stream, `samples`. Specify the number of idle cycles to insert between input samples, `postsampleidles`, and the number of idle cycles between frames, `postframeidles`.

`[samples,ctrl,len] = ltehdlFramesToSamples(frames,postsampleidles, postframeidles,samplesize)` creates a sample stream where each sample is represented by `samplesize` values. The function inserts `samplesize` zeros for each idle cycle requested. The `ctrl` and `len` vectors are the same size as when `samplesize` is 1.

`[samples,ctrl,len] = ltehdlFramesToSamples(frames,postsampleidles, postframeidles,samplesize,interleaved)` orders the sample stream, assuming the input samples are interleaved, when `interleaved` is 1 (`true`). The `interleaved` argument is valid only when `samplesize` is greater than 1.

Input Arguments

`frames` — Frames of input samples

column vector | cell array of column vectors

Frames of input samples, specified as a column vector or a cell array of column vectors. The frames in the cell array can be different sizes.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

`postsampleidles` — Number of idle cycles to insert between samples
0 (default) | integer

Number of idle cycles to insert between samples, specified as an integer. The function inserts `samplesize` zeros for each idle cycle, and sets all control signals to 0 (`false`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`postframeidles` — Number of idle cycles to insert between frames
0 (default) | integer

Number of idle cycles to insert between frames, specified as an integer. The function inserts `samplesize` zeros for each idle cycle, and sets all control signals to 0 (`false`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`samplesize` — Number of values representing each sample
1 (default) | positive integer

Number of values representing each sample, specified as a positive integer. The function returns one set of control signals for each `samplesize` values.

For example, in the LTE standard, the turbo code rate is 1/3, so each turbo-encoded sample is represented by one systematic, and two parity values: S_n, P_n1, and P_n2. In this case, set `samplesize` to 3.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`interleaved` — Order of output samples relative to input order
0 (default) | logical scalar

Order of output samples relative to input order, when more than one value represents each sample, specified as a logical scalar.

For example, for 1/3 turbo-encoded samples, the input frame can be ordered `[S_1 P1_1 P2_1 S_2 P1_2 P2_2]` or `[S_1 S_2 P1_1 P1_2 P2_1 P2_2]`. In the first case, the default output would be the same order as the input. To achieve that output order for the second input, set `interleaved` to 1 (`true`).

Data Types: `single` | `double` | `logical`

Output Arguments

`samples` — Stream of samples
column vector

Stream of samples, returned as a column vector. For N samples in an input frame, the output is N + `samplesize`×(N×`idlecyclesbetweensamples` + `idlecyclesbetweenframes`) values per frame.

`ctrl` — Control signals accompanying sample stream
M-by-3 matrix

Control signals accompanying sample stream, returned as an M-by-3 matrix. The matrix includes three control signals, `start`, `end`, and `valid`, for each `samplesize` elements in `samples`. For N input samples in F frames, M is N + N×`idlecyclesbetweensamples` +

F×`idlecyclesbetweenframes`. When you import this variable into Simulink, use a Sample Control Bus Creator block to convert the signals into the bus type used by the Wireless HDL Toolbox blocks.

`len` — Frame length
column vector of integers

Frame length, returned as a column vector of integers. This value is the number of valid samples in the corresponding frame for each `samplesize` elements in `samples`. This vector is the same length as `ctrl`.

Compatibility Considerations

`ltehdlFramesToSamples` function has been removed
Errors starting in R2021a

`ltehdlFramesToSamples` function has been removed. Use the `whdlFramesToSamples` function instead. The functionality of the two functions is the same. To update your code, replace instances of `ltehdlFramesToSamples` with `whdlFramesToSamples`.

See Also

Blocks
Frame To Samples | Samples To Frame

Functions
`whdlFramesToSamples` | `whdlSamplesToFrames`

Topics
“Verify Turbo Decoder with Streaming Data from MATLAB”
“Streaming Sample Interface”

Introduced in R2017b

ltehdlSamplesToFrames

(Removed) Convert sample stream to frame-based data

Note `ltehdlSamplesToFrames` name is changed to `whdlSamplesToFrames`. Retained this file as part of compatibility consideration process. This function will be removed soon. See “Compatibility Considerations”.

Syntax

```
outframes = ltehdlSamplesToFrames(samples,ctrl)
outframes = ltehdlSamplesToFrames(samples,ctrl,maxlen)
outframes = ltehdlSamplesToFrames(samples,ctrl,maxlen,interleaved)
```

Description

`outframes = ltehdlSamplesToFrames(samples,ctrl)` composes frame-based data from a sample stream and corresponding control signals. The control signals indicate the validity of the samples and the boundaries of the frames. The function calculates the maximum frame length from the input data and control signals, and removes any idle or nonvalid samples from the data.

`outframes = ltehdlSamplesToFrames(samples,ctrl,maxlen)` composes frame-based data, using the maximum frame length. If an input frame described by `samples` is larger than `maxlen`, the function truncates the frame.

`outframes = ltehdlSamplesToFrames(samples,ctrl,maxlen,interleaved)` orders the frame-based data, assuming the input samples are interleaved, when `interleaved` is 1 (`true`). The `interleaved` argument is valid only when each sample is represented by multiple values. The function computes the number of values representing each sample by comparing the length of `samples` and `ctrl`.

Input Arguments

`samples` — Stream of samples

column vector

Stream of output samples, specified as a column vector. The vector can include idle cycles between samples and between frames. Idle cycles are discarded. The frames represented by the stream can be different sizes. The vector length, *N*, must be an integer multiple of the length of the `ctrl` matrix, *M*. Differing lengths mean that each sample is represented by *N*/*M* values.

For example, in the LTE standard, the turbo code rate is 1/3, so each turbo-encoded sample is represented by one systematic, and two parity values: S_n, P_n1, and P_n2. In that case, the length of `samples` must be three times the length of `ctrl`.

`ctrl` — Control signals accompanying sample stream

M-by-3 matrix

Control signals accompanying the sample stream, specified as an *M*-by-3 matrix. The matrix includes three control signals, `start`, `end`, and `valid`, for each sample in `samples`. Each sample can be

represented by more than one value. In that case, the length of `samples` must be an integer multiple of *M*.

For example, in the LTE standard, the turbo code rate is 1/3, so each turbo-encoded sample is represented by one systematic, and two parity values: S_n, P_n1, and P_n2. In that case, the length of `samples` must be three times the length of `ctrl`.

`maxlen` — Maximum frame length
integer

Maximum frame length, specified as an integer. The input frames in `samples` can be different sizes. The output column vector reflects the size of the input frame, according to `ctrl`. If a frame is larger than `maxlen`, the function truncates the frame and returns a warning message.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`interleaved` — Order of output samples relative to input order
0 (default) | logical scalar

Order of output samples relative to input order, when more than one value represents each sample, specified as a logical scalar.

For example, 1/3 turbo-encoded samples are represented by $[S_1\ P_11\ P_12\ S_2\ P_21\ P_22]$. To reorder the samples so that systematic and parity values are grouped together, set `interleaved` to 1 (`true`). The output order is then $[S_1\ S_2\ P_11\ P_21\ P_12\ P_22]$.

Data Types: `single` | `double` | `logical`

Output Arguments

`outframes` — Frames of output samples
column vector | cell array of column vectors

Frames of output samples, returned as a column vector or a cell array of column vectors. The size of the output column vector reflects the size of the input frame, as determined by the control signals in `ctrl`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

Compatibility Considerations

`ltehdlSamplesToFrames` function has been removed
Errors starting in R2021a

`ltehdlSamplesToFrames` function has been removed. Use the `whdlSamplesToFrames` function instead. The functionality of the two functions is the same. To update your code, replace instances of `ltehdlSamplesToFrames` with `whdlSamplesToFrames`.

See Also

Blocks
Frame To Samples | Samples To Frame

Functions
`whdlFramesToSamples` | `whdlSamplesToFrames`

Topics
“Verify Turbo Decoder with Streaming Data from MATLAB”
“Streaming Sample Interface”

Introduced in R2017b

samplecontrolbus

Create sample-streaming control bus

Syntax

```
samplecontrolbus
```

Description

`samplecontrolbus` declares a `samplecontrol` type bus instance in the workspace. This instance is required for HDL code generation. Call this function before you generate HDL code from Wireless HDL Toolbox blocks.

Examples

Declare Bus in Base Workspace

In the `InitFcn` callback function of your Simulink model, or at the MATLAB® command line, use this command to declare a `samplecontrol` type bus instance in the base workspace. If you create your model with the Wireless HDL Toolbox model template, this step is done for you.

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

If you do not declare an instance of `samplecontrolbus` in the base workspace, you might encounter this error when you generate HDL code in Simulink.

```
Cannot resolve variable 'samplecontrol'
```

See Also

Blocks
Frame To Samples | Samples To Frame

Topics
“Streaming Sample Interface”

Introduced in R2017b