**Physical coding library**

Three different packages, related through area of application, are part of this project : **endec\_8b10b**, **endec\_64b66b** and **scrambler\_descrambler**. As the names hint, the first two contain the implementations of 8b10b and 64b66b codes. These codes are used in physical layers of various data transmission protocols to improve certain properties of the data stream. The last package models an additive and a multiplicative scrambler/de-scrambler pair. These are also important components of communication systems.

**1.Endec\_8b10b**

**1.1.General**

8b10b coding scheme is used in communications, enabling DC-balance and bounded disparity of data stream while providing enough transitions for clock recovery. The code transforms an 8bit input value into a 10bit value that's sent bit by bit on the serial line. Symbols of the code are chosen such that at every 10bit boundary the difference between transmitted 1's and transmitted 0's is not greater that +1 or lesser than -1.

**1.2.Endec\_8b10b package**

The package contains the implementation of an encoder and a decoder along with coverage definitions, modeled using the UVM library. The path of the package file relative to the project directory is:

*physical\_coding\_library/encoder\_decoder/sv/endec\_8b10b/endec\_8b10b\_pkg.sv*

**1.3.Encoder and decoder usage**

Once the package file is imported the encoder and decoder classes can be instantiated under any *uvm\_component* class of the importing environment.

To encode 8bit input data the encoder class defines the *encode()* function that has the following signature:

***virtual function bit*** *[9:0] encode (endec\_8b10b\_enc\_in\_dec\_out\_s a\_encode\_in);*

It accepts as input a struct type named  *endec\_8b10b\_enc\_in\_dec\_out\_s* and outputs the encoded 10bit symbol.

The input struct type definition can be found in the *endec\_8b10b\_types.svh* file that's part of the package and is listed below:

*//serves as input of the encoder and output of the decoder*

***typedef struct*** *{*

*// 8 bit value input for encoding*

*// or output of decoding*

***bit*** *[7:0] enc\_dec\_8b\_val;*

*// field indicating if it's data or control symbol*

***bit*** *is\_k\_symbol;*

*//used to register decoding errors*

*// 0 - no error*

*// 1 - disparity error*

*// 2 - symbol error*

***bit*** *[1:0] decode\_err;*

*} endec\_8b10b\_enc\_in\_dec\_out\_s;*

It defines an 8bit field for the value to be encoded, a one bit field indicating if it's a data or a control symbol and a field used for registering decode errors.

To decode 10bit symbols the decoder class defines the *decode()* function with the following signature:

***virtual function*** *endec\_8b10b\_enc\_in\_dec\_out\_s decode (****input bit*** *[9:0] a\_coded\_symbol);*

It accepts as input a 10bit coded symbol and outputs the struct holding the decoded 8bit data with the control/data symbol indication and the *decode\_err* field that registers decoding errors.

**1.4.Endec\_8b10b verification package**

The verification package instantiates the encoder and decoder classes feeding the encoder with random input and using it's output as input for the encoder which outputs the

decoded symbols. Inputs of the encoder and outputs of the decoder are passed to the scoreboard which checks these coincide.

The verification package file's path, relative to the project directory, is:

*/physical\_coding\_library/encoder\_decoder/examples/endec\_8b10b/ve/endec\_8b10b\_ve\_pkg.sv*

**1.5.Endec\_8b10b tests**

The tests provided with the above packages are located under directory:

*/physical\_coding\_library/encoder\_decoder/examples/endec\_8b10b/tests*

This directory contains the base test class that instantiates the verification environment and two child test classes, one running random input data and one running only control input data.

**2.Endec\_64b66b**

**2.1.General**

A code frequently used in high-speed communication systems is the 64b66b code. Its main advantages are the reduced overhead and the fact that it can generate enough transitions for clock recovery and alignment of data at the receiver. As the name of the coding suggests, it transforms 64bit input data into a 66bit coded block of which the first 2 MSBs constitute a preamble identifying the type of the block(which can be data or control). These two bits also guarantee that at least one transition exists for every transmitted 66 bits, thus aiding clock synchronization. Scrambling is applied on the 64LSBs of the coded block to ensure relatively even distribution of 1's and 0's inside the transmitted stream.

**2.2.Endec\_64b66b package**

The package contains code modeling a 64b66b encoder and decoder along with coverage definitions. The implementation follows the specifications from clause 49 of the IEEE 802.3 standard. The package file relative to the project directory is:

*/physical\_coding\_library/encoder\_decoder/sv/endec\_64b66b/endec\_64b66b\_pkg.sv*

**2.3.Encoder and decoder usage**

An environment importing the endec\_64b66b package can instantiate the encoder and/or decoder classes and use the provided functions to perform 64b66b encoding and/or decoding.

The encoder class defines the *encode()* function to be used for the encode process. It follows the definition of the encoding process defined in clause 49 of IEEE 802.3 standard.

The signature of the function is:

***virtual function bit*** *[65:0] encode (****bit****[71:0] a\_xgmii\_in);*

The encoding process operates on two XGMII type transfers. These are defined in clause 46 of IEEE 802.3 standard which introduces a media independent interface, abbreviated as XGMII. One such transfer consists of a 32bit data input and a 4bit input carrying information about the type of the 4 bytes contained in the first input, each bit corresponding to one byte. The aforementioned encoding function receives as input two such XGMII transfers concatenated into the 72bits input.

The result of the encoding process is a 66bits coded block explained in clause 49 of IEEE 802.3 standard.

The decoder class defines the *decode()* function to be used for decoding 66bits encoded code-blocks. It follows the definition of the decoding process defined in clause 49 of IEEE 802.3 standard. The signature of the function is:

***virtual function bit [71:0] decode (bit[65:0] a\_coded\_block\_in);***

It accepts as input a 66bit code-block and returns as output 72bits holding the information for 2 XGMII type transfers(identical to the input of the encode function).

**2.4.Endec\_64b66b verification package**

The verification environment for the endec\_64b66b package is placed in directory:

*/physical\_coding\_library/encoder\_decoder/examples/endec\_64b66b/ve*

It contains a driver that instantiates the encoder and decoder classes and feeds the encoder with stimulus requested from the sequencer. The output of the encoder is passed as input to the decoder and for each decoder output a scoreboarding versus the original encoder input is performed.

**2.5.Endec\_64b66b tests**

The environment comes with example tests under directory:

*/physical\_coding\_library/encoder\_decoder/examples/endec\_64b66b/tests*

Two tests are available: one with legal input keeping the encoder/decoder state-machines in legal operation and one that forces the state-machines in the error states by running illegal input sequences.

**3.Scrambler\_descrambler**

**3.1.General**

Scramblers and de-scramblers are used in communication systems to improve certain characteristics of the transmitted data.

There are two types of scramblers : additive, also called synchronous, and multiplicative or self-synchronizing scramblers.

Scramblers/de-scramblers of both types are implemented as linear feedback shift registers(LFSR) and they are determined by the polynomial of their LFSR. Contrasting their multiplicative counterparts, that do not require an initial LFSR state to be set, additive scramblers need to be loaded with a synchronization sequence prior to starting operation on an input stream.

**3.2.Scrambler\_descrambler package**

The package contains implementations for additive and multiplicative scramblers/de-scramblers. The modeling classes have parameters that allow users to specify the order of the defining polynomial and it's taps.

One class is defined for the additive case that models both scrambler and de-scrambler as both have the same implementation. For the multiplicative flavor the implementation varies

between the scrambler and de-scrambler so two classes are defined.

With the help of the parameters any type of scrambler/descrambler can be instantiated as this comes down to setting a specific LFSR by setting the parameters to the desired values.

**3.3.** **Scrambler\_descrambler usage**

Usage examples for both implementations are available under directory:

*/physical\_coding\_library/scrambler\_descrambler/examples*

**3.3.1 Additive scrambler/descrambler**

Inside the test */additive/tests/scrambler\_descrambler\_additive\_test.sv* there's an example of how an additive scrambler/de-scrambler pair can be instantiated and used. The declaration of the classes with specific specializations is done inside the test class:

*// convenience parameter holding the order of the polynomial*

***parameter*** *ORDER = 64;*

*// the scrambler and descrambler are actually the same*

*scrambler\_descrambler\_additive#(ORDER, 'h8001000000000000) m\_add\_scrambler;*

*// descrambler instance*

*scrambler\_descrambler\_additive#(ORDER, 'h8001000000000000) m\_add\_descrambler;*

Inside the *build\_phase()* function of the test class are the calls to *create()* that return the scrambler/descrambler instances. Since these are of additive type they also need an initial state for the LFSR to be set and this is done though the *load\_lfsr()* function calls:

***virtual function void*** *build\_phase(uvm\_phase phase);*

***super****.build\_phase(phase);*

*m\_add\_scrambler = scrambler\_descrambler\_additive #(ORDER, 'h8001000000000000)::type\_id::create(*

*" m\_add\_scrambler",*

***this***

*);*

*m\_add\_descrambler = scrambler\_descrambler\_additive #(ORDER, 'h8001000000000000)::type\_id::create(*

*" m\_add\_descrambler",*

***this***

*);*

*// load same initial values in both scrambler and descrambler*

*m\_add\_scrambler.load\_lfsr('h2a);*

*m\_add\_descrambler.load\_lfsr('h2a);*

***endfunction***

**3.3.2 Multiplicative scrambler/descrambler**

An example on the usage of the multiplicative scrambler/descrambler is provided inside test file:

*/multiplicative/tests/scrambler\_descrambler\_multiplicative\_test.sv*

Here a scrambler and a descrambler class are declared as members of the test class, with a specific specialization :

*// convenience parameter holding the order of the polynomial*

***parameter*** *ORDER = 64;*

*// the scrambler and descrambler are actually the same*

*scrambler\_descrambler\_additive#(ORDER, 'h8001000000000000) m\_add\_scrambler;*

*// descrambler instance*

*scrambler\_descrambler\_additive#(ORDER, 'h8001000000000000) m\_add\_descrambler;*

Inside the *build\_phase()* function of the test the two are instantiated by a call to *create()*:

***virtual function void*** *build\_phase(uvm\_phase phase);*

***super****.build\_phase(phase);*

*// the polynomial is x^63 + x^48*

*m\_mult\_scrambler = scrambler\_multiplicative #(ORDER, 'h8001000000000000)::type\_id::create(*

*"m\_mult\_scrambler",*

***this***

*);*

*m\_mult\_descrambler = descrambler\_multiplicative #(ORDER, 'h800100000000000)::type\_id::create(*

*"m\_mult\_descrambler",*

***this***

*);*

***endfunction***

**3.4. Scrambler\_descrambler tests**

The two tests , one for each type of scrambler/descrambler pair, instantiate the modeling classes as presented above and use bit-stream input to feed the scrambler, who's outputs are passed on as inputs to the descrambler. A scoreboarding is performed between the scrambler input and the descrambler output to check correct function.

**4. Running the examples**

A demo script is provided to run the examples that come with the packages. It is placed in */physical\_coding\_library* directory and can be used to run any of the tests in the packages. This is how an invocation, running a test from the endec\_8b10b suite, looks :

*demo.sh -ex\_to\_run endec\_8b10b*

*-test endec\_8b10b\_tests\_all\_k\_test*

*-tool vcs -seed random*

*-verbosity UVM\_HIGH*

*-quit\_cnt 1*

Below is the description of the script options:

- *i* optionstarts the simulation in interactive mode;

*- ex\_to\_run* option allows selection between existing packages with the following options to choose from : *endec\_8b10b, endec\_64b66b, scrmb\_descrmb\_add, scrmb\_descrmb\_mult* or *scrmb\_descrmb\_endec*;

- *test* option lets a user select between the tests available for the selected package;

- *tool* allows the user to select one of the three supported tools;

- *seed* can be used to set a specific seed for the run;

- *verbosity* can be used to set the desired level of detail for messaging;

- *quit\_cnt* option allows the user to set the number of UVM\_ERRORs that causes

the simulation to end (default is 0 meaning the simulation will continue past all

UVM\_ERRORs); setting it to 1 will stop the simulation on the first UVM\_ERROR;