# Advantages of using UVM/System Verilog IEEE standards to Verify Complex Probabilistic Constellation Shaping Design for a Coherent DSP ASIC.

Nipun Bhatt

Infinera Corporation
140 Caspian Court, Sunnyvale, CA 94089
Phone: 408-543-7485
nbhatt@infinera.com

***Abstract* – System Verilog[SV] and UVM provide various mechanisms to reuse the verification components. Configuration classes, SV-Interfaces, SV-DPI, UVM Sequences, UVM Agents, UVM TLM Interfaces and many more. This paper demonstrates the effective use of these mechanisms to create an environment which mimics the design hierarchy. It shows how verification environments can be reused from module level to sub2chip level, to sub-chip level, to full chip level testbench. It also demonstrates the effective testing mechanism to verify a complex Probabilistic Constellation Shaping [PCS] Algorithm in a high-performance Coherent DSP ASIC and highlights the significant advantages over traditional verification methodologies.**

## I. INTRODUCTION

This paper highlights the complexity of Coherent DSP design and the limitations of traditional verification methodologies. It details the various mechanisms provided by System Verilog and UVM based environment to address verification challenges.

It explains the implementation mechanism used for various areas like component reusability, parallel processing, seamless integration, scalability, debugging simplicity, coverage, and hooks for latency measurements to highlight the advantages.

## II. DESIGN COMPLEXITY

High-performance Coherent DSPs are powered by complex algorithms, one of which is a Probabilistic Constellation Shaping Encoding and Decoding algorithm [3]. A very high-level optical transceiver data path is shown in *Figure 1*. It shows how the functional layers are divided into different categories.

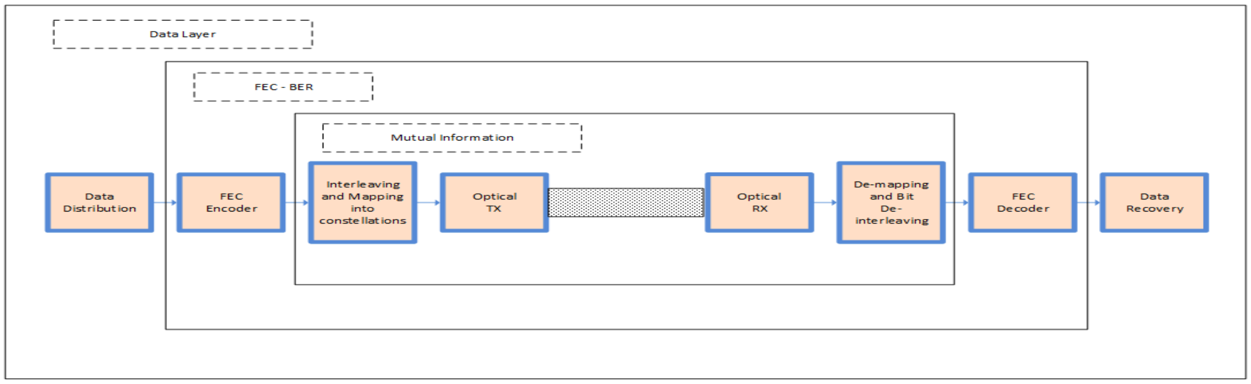


*Figure 1: Optical Coherent Transceiver Datapath*

PCS features are listed, highlighting the design complexity.

i) Subcarrier[5] level bandwidth distribution.
ii) Configuration based scheduling of client traffic.
iii) Symbol level Encoding and Decoding[3] (zero-tolerance Decoding algorithm).
iv) Parallel processing using several Encoders and Decoders. Throughput and longer codewords drive parallel processing.

v) The importance of latency to recreate the subcarrier level traffic.

vi) Interleave and deinterleave design functions and the symbol level communication with Encoder and Decoder blocks to form the data boundaries.

## III. TRADITIONAL VERIFICATION METHODOLOGY AND DRAWBACKS

DSP models are written and verified in MATLAB. Algorithm specific MATLAB models can be converted to C-models so that they can be used for verification. Traditionally, using C-models to verify algorithm-specific RTL design is very popular as the simple C-model integration in the System Verilog testbench using DPI import and then deploying it for targeted block-level testing, which makes it an ideal solution.

System Verilog DPI allows direct inter-language function calls between System Verilog and any foreign programming language with a C function call[1]. Functions are implemented in C and given import declaration in the System Verilog module (*Figure 2*). Simplicity comes at the cost of scalability and reusability. Shortcomings are as follows.

i) Simple System Verilog module with DPI import can verify the module level design but is not very flexible to be integrated at a higher level of testbench hierarchy compared to UVM agent/environment integration.

ii) They are not very effective when parallel processing is a requirement. Especially with multiple instances of algorithm-specific blocks with each block working on a different configuration(Encoder and Decoder slices instances *Figure 7* and *Figure 8*). The test is overloaded with managing configurations of multiple SV modules and the instance hierarchy to program the C-models compared to the efficient config_db approach.

iii) System Verilog module with DPI import is highly inefficient when communicating with upper or lower-level environment components compared to TLM ports which provide great flexibility modeling layered protocol.

One can make the argument about having an end to end C model instead of using SV and UVM based methodology and fix all the issues highlighted above but, that too has its own limitations.

iv) Configuration intensive design with a constraint random verification is not possible using only end-to-end C-models.

v) An end-to-end C-model approach is inefficient for self-checking tests. They cannot provide flexible debugging hooks.

vi) The effort and work we put in at module level are not scalable and reusable without more toil at a higher level which has timeline or resource implications.

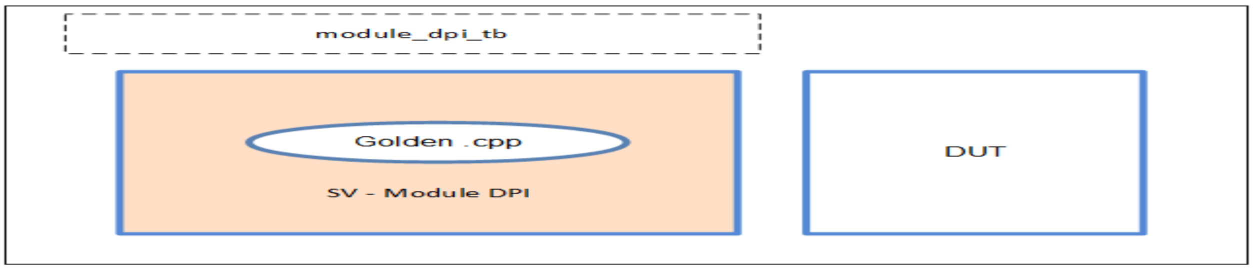


*Figure 2: System Verilog Module DPI Testbench for Algorithm Specific Design*

To verify the algorithms with several configurations, parallel data processing, and complex scheduling design, and to achieve the verification goals in timely manner we opted to use UVM/System Verilog IEEE standards in view of the reusability, portability, and seamless integration advantages without compromising on the reusability of the C-models.

## IV. Detailed Description Of Verification Infrastructure And Advantages

This paper highlights the development effort around four building blocks: Scheduler Agent, Slice level Encoder Agent, Slice level Decoder Agent and Interleave/De-interleave Agent [3], pillars of the PCS environment which mimics the design implementation and hierarchy. As a first step we created Encoder and Decoder slice level UVM Agents, the basic building blocks verifies the math function of Probabilistic Constellation Shaping block [3], both on the receive [Decoder] and transmit [Encoder] paths.

To maintain the data integrity of the core math function – Encoding and Decoding, slice level UVM Agents are integrated with system-level C-models using DPI import on the driving and monitoring path (*Figure 3* and *Figure 4*). There are several instances of Encoder and Decoder slice level Agents as part of PCS sub2chip (*Figure 7* and *Figure 8*), receive sub-chip and full-chip environment. Common ps_slice_cnfg configuration class is used across the slice level Agents (*Program 1*) and Scheduler reference model. The number of PCS instances increases manifold as we go up in the testbench hierarchy.

PCS Encoder and Decoder slice level testbench were developed to verify one instance of the encoder and decoder slice with 100% coverage goals. PCS Encoder and Decoder slices are designed to operate at bit rates, as high as 800Gbps for a few supported Giga baud rates. Test combinations increase owning to the configuration support at some fraction of lower data-rate granularity to support the subcarrier level data distributions. Configuration class ps_slice_cnfg accepts the high-level user-programmable configurations i) bit rate and ii) baud rate iii) datatype, (*Program 1*) it derives all the other low-level configuration needed to configure C model and RTL in PS Encoder and Decoder Slice(*Program 2*). It is very critical to have each slice work independently on its configuration to support the subcarrier level bandwidth distribution requirement.

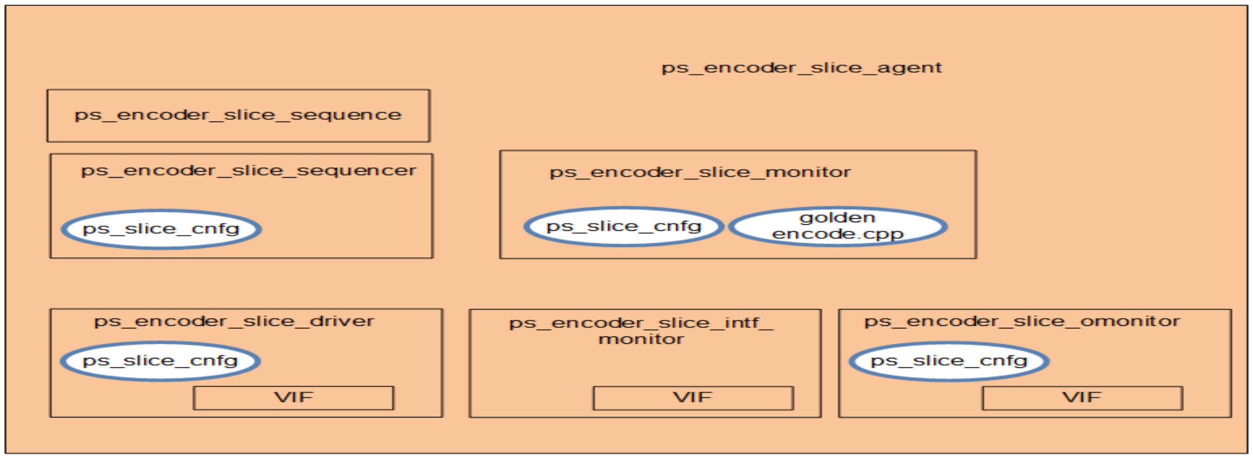


*Figure 3: PCS Encoder Slice Agent*

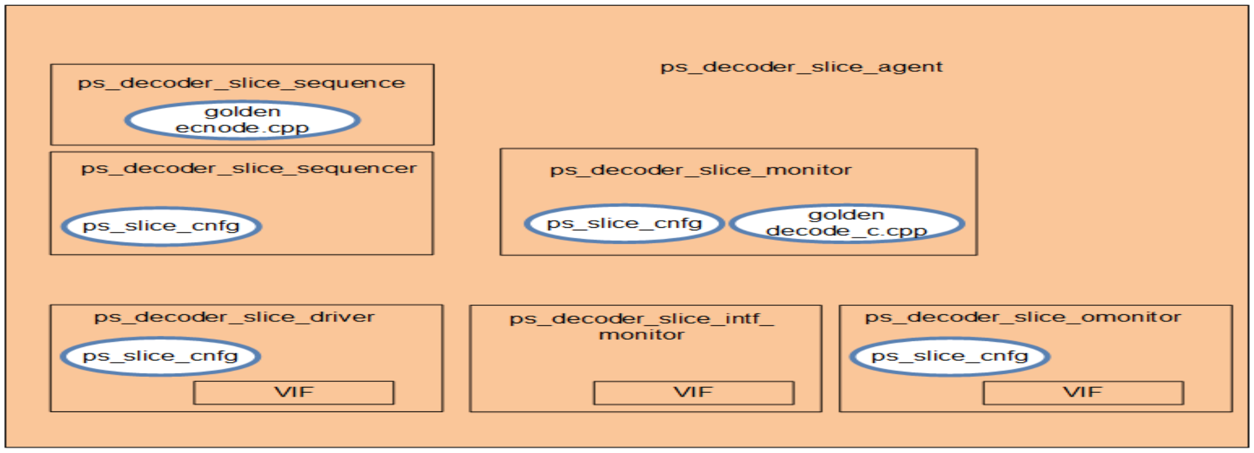


*Figure 4: PCS Decoder Slice Agent*

*Program 1 SAMPLE CONFIGURATION CODE – ps_slice_cnfg.sv*

```
 // Many more parameters are derived using Giga Baud and Bit Rate (data_rate_per_wave) configuration
// They are used to program the Golden C model and Design Configuration

 constraint const_giga_baud {
    giga_baud dist {"GigaBaud1, GigaBaud2, GigaBaud3"};
    // actual code is removed, only variation is displayed
 }
 constraint const_data_rate_per_wave {
    data_rate_per_wave dist {"P to Q"};  // where P is lowest data rate and Q is highest data rate
    // actual code is removed and only the range is displayed
 }
 constraint const_payload_type {
    payload_type dist {INC:/1, FIXED_55:/1, FIXED_AA:/1, RANDOM:/1, FIXED_FF:/1, FILE:/1};
 }
```

*Program 2 SAMPLE EXTENDED TEST CODE*

```
// A-Giga baud and B-Gbps is used as an example in the program
// Few variations of A and value of B Up to 800gbps with some fraction of B intervals

`ifndef DECODER
  `include "test_ps_encoder_slice.sv"
`else
  `include "test_ps_decoder_slice.sv"
`endif

`ifndef DECODER
class test_ps_slice_Agbaud_Bgbits extends test_ps_encoder_slice;
`else
class test_ps_slice_Agbaud_Bgbits extends test_ps_decoder_slice;
`endif
              `uvm_component_utils(test_ps_slice_Agbaud_Bgbits)
              ps_slice_config  ps_config;

              function new(string name = "test_ps_slice_Agbaud_Bgbits", uvm_component parent = null);
                …
              endfunction // new

              virtual function void build_phase(uvm_phase phase);// the build phase
                super.build_phase(phase);

              // Create the object
              ps_config = ps_slice_config::type_id::create("ps_config", this);
              uvm_config_db#(ps_slice_config)::set(this, "*", "ps_slice_config", ps_config);
                // Randomizing the Configuration
                ps_config.randomize() with {
                                    ps_config.giga_baud == A; // Example Values
                                    ps_config.data_rate_per_wave==B;
                                    ps_config.payload_type == RANDOM;
                                   };
              endfunction // build_phase
```

Scheduler Agent is another key component developed for PCS verification (*Figure 5*). The scheduling algorithm is implemented in System Verilog reference model and integrated as a part of UVM Agent to create a predictor model. Same UVM Agent is used in both transmit and receive paths. Scheduler effectively writes or reads to the buffer as needed by the design configuration. Scheduler Agent communicates with several slice Agents.

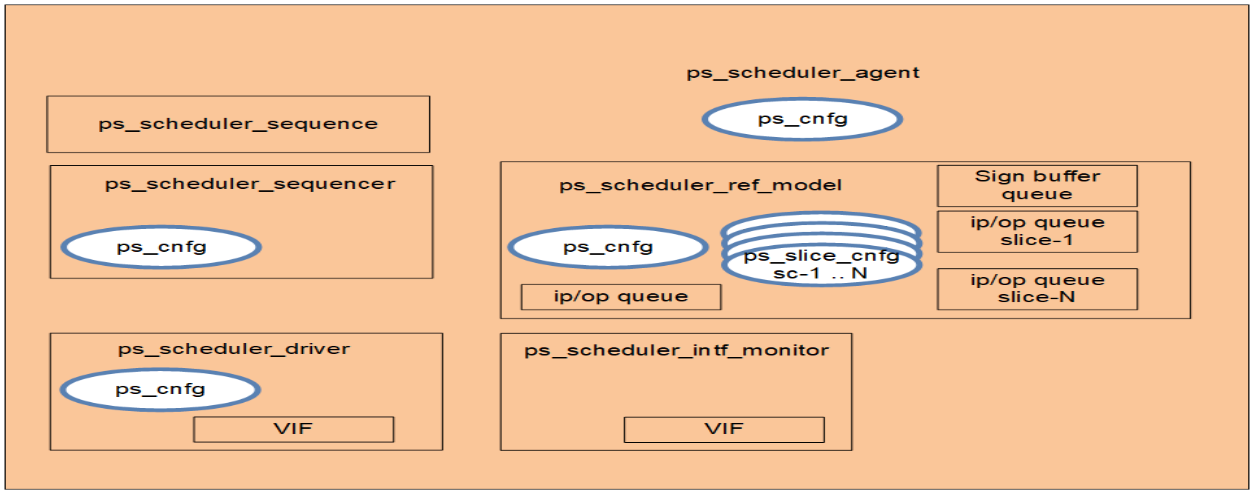


*Figure 5: PCS Scheduler Agent*

A key feature of the Scheduler Agent is to distribute and recreate the data using subcarrier level bandwidth requirements. On the transmit path (*Figure 7*), it distributes subcarrier level data to many PCS Encoder slices periodically so that they can work in parallel on the longer codewords. On the receive path (*Figure 8*), it gathers the data from many PCS Decoder slices periodically based on the subcarrier level bandwidth requirements and creates the original client data. Incoming data to the reference model is stored in "queue" data structures, which continuously builds and shrinks in size on both receive and transmit directions.

Common configuration classes i) ps_slice_cnfg.sv ii) ps_cnfg.sv is reused on transmit and receive path for both module and sub2chip and higher level in the testbench hierarchy. Data distribution and recovery are purely based on the bandwidth defined for each subcarrier, either equal or constrained random distribution. The key configuration defined in the pc_cnfg.sv is about how the data is distributed on the subcarrier.

De-interleave(or interleave) UVM Agent (*Figure 6*) uses the design configuration, to achieve the layered testbench structure which mirrors the design hierarchy. The main function is implemented as a reference model working on the design configuration. Interleave/De-interleave Agent communicates to several slice level Agents.

On the transmit path, De-interleave Agent performs interleaving on the Encoded data coming out of several PCS Encoder slices and creates the boundary. Interleaved data is stored in such a way that I (In-phase) and Q (Quadrature) bits remain together when the symbols are recovered. Each constellation point is represented with the number of bits depending on the type of modulation. The bits which belong to different constellation points are put together by PCS interleave block.

On the Receive path, the main function of the De-interleave Agent is to validate the markers and start de-interleaving the bits to distribute through Decoder slice level Agents. With FEC in the data-path, we have considered that the data going through the Decoder slices is error-free and the generated I (in-phase) and Q (Quadrature) bits are recovered together for the decoding algorithm to work. Like in the scheduler Agent, the data is stored in the "queue" structures which periodically builds and shrinks in size.

The order of the bits is very important to recreate the client data. Heavy use of configurations, virtual interfaces, and transaction-level modeling helped achieve effective communication between the UVM Agents and seamless data-path monitoring.

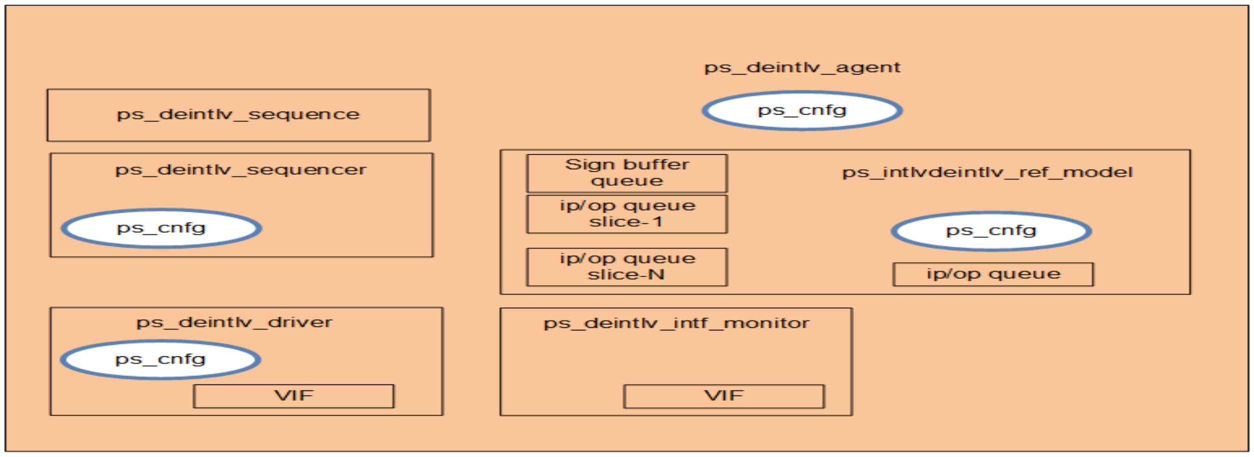


*Figure 6: PCS De-interleave Agent*

PCS sub2chip level environment integrated one instance of Scheduler Agent, several Encoder and Decoder slice Agents, and one instance of De-interleave Agent (*Figure 7* and *Figure 8*). A closer look at both test-benches gives clarity on the emphasis of the component reuse. No component is designed for any specific environment or path. UVM TLM interface provides key support in communication between the components in both transmit and receive direction for layered UVM environments.

Environment code for the transmit path is captured in (*Program 3*) and for receive path is captured in (*Program 4*). Both the sample code highlights the integrated UVM components and how they are connected. In (*Program 3* code, we see three UVM agents while In (*Program 4*) code, we see five UVM agents. There are two instances of scheduler agents on the receive path which is more complex. In a receive path, random data is scheduled, encoded and interleaved before it is driven on the interface.

Our unique approach of integrating complete transmit predictor UVM components as part of the receive path Deinterleave sequence would not be possible without UVM/System Verilog based environments. Due to the reference model implementation in both Scheduler and Deinterleave (or interleave) agents. It is very convenient to mimic the complete transmit hierarchy using UVM TLM ports.

In the connect phase of (*Program 4*) Deinterleave (or interleave) sequencer is connected to the scheduler reference model to distribute the payload data to multiple PCS Encoder slice monitors. Encoded data is transferred from various slices to the Deinterleave (or interleave) reference model where it performs the interleaving function and makes the data available in the Interleave Queue which is part of the Deinterleave sequencer. Data from the sequencer queue is eventually used by Deinterleave (or interleave) driver to pull the data from the sequencer and drive on the interface.

On the monitoring path, Deinterleave (or interleave) agent is used to collect the data from the interface and deinterleave the data using the reference model. Deinterleaved data is distributed to many decoder slice monitors where it goes through the C-models to produce the decoded output, which is sent back scheduler agent to recreate the client data. (*Program 4*) connect phase highlights the monitoring path connections.

The environment structure provides handy debugging hooks for any failure observed at the boundary of each Agent. Using virtual interfaces, monitors are hooked up to compare slice level codewords in both the directions. A single bit mismatch can be caught and isolated very easily with these features. Testbench provides a bit-level comparison for both OH and client payload using two separate scoreboards. All the scoreboard in use and how the connection established is captured in (*Program 3*) and (*Program 4*).

On the transmit path environment, slice level scoreboards are used to debug any Encoding issues and end-to-end scoreboard to check the data integrity. Similarly, on the receive path, slice level scoreboards are used to debug any decoding issues and end-to-end scoreboard to check the data between predicted output and actual output. On the receive path, one additional scoreboard is implemented to check the randomly generated client data with actual decoded client data.

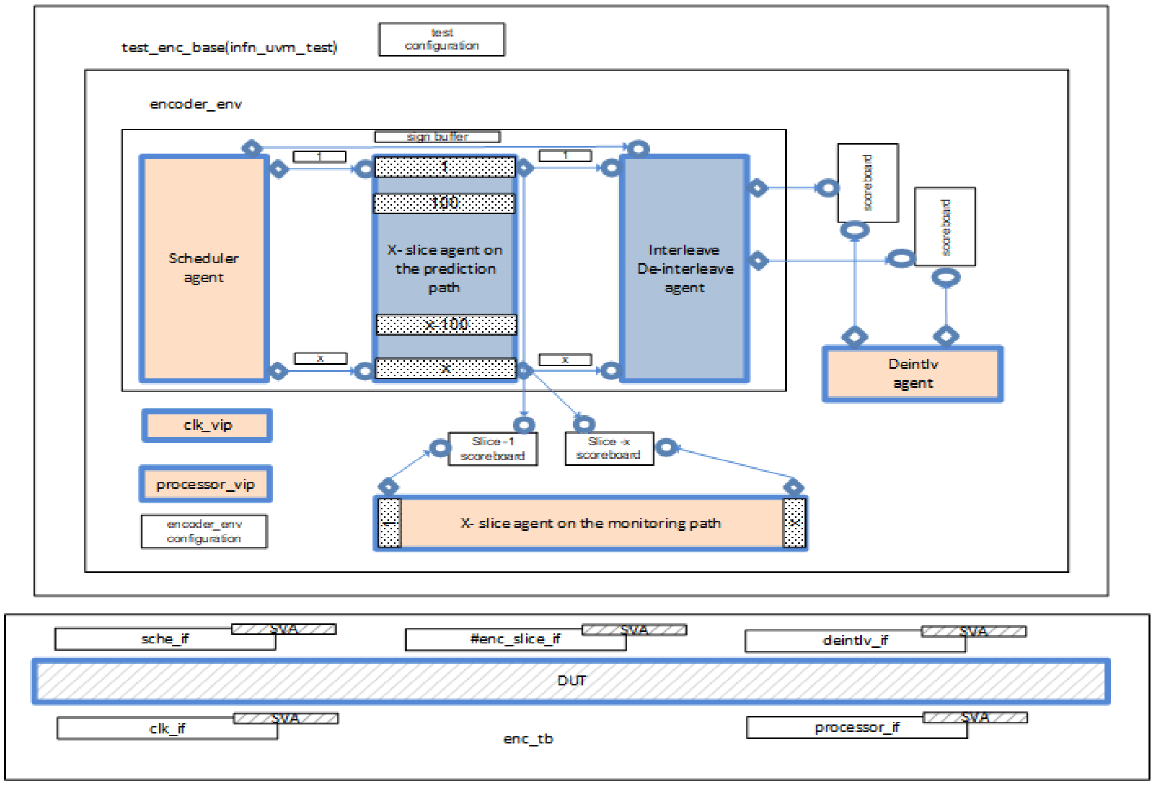


Figure 7: Transmit PCS sub2chip

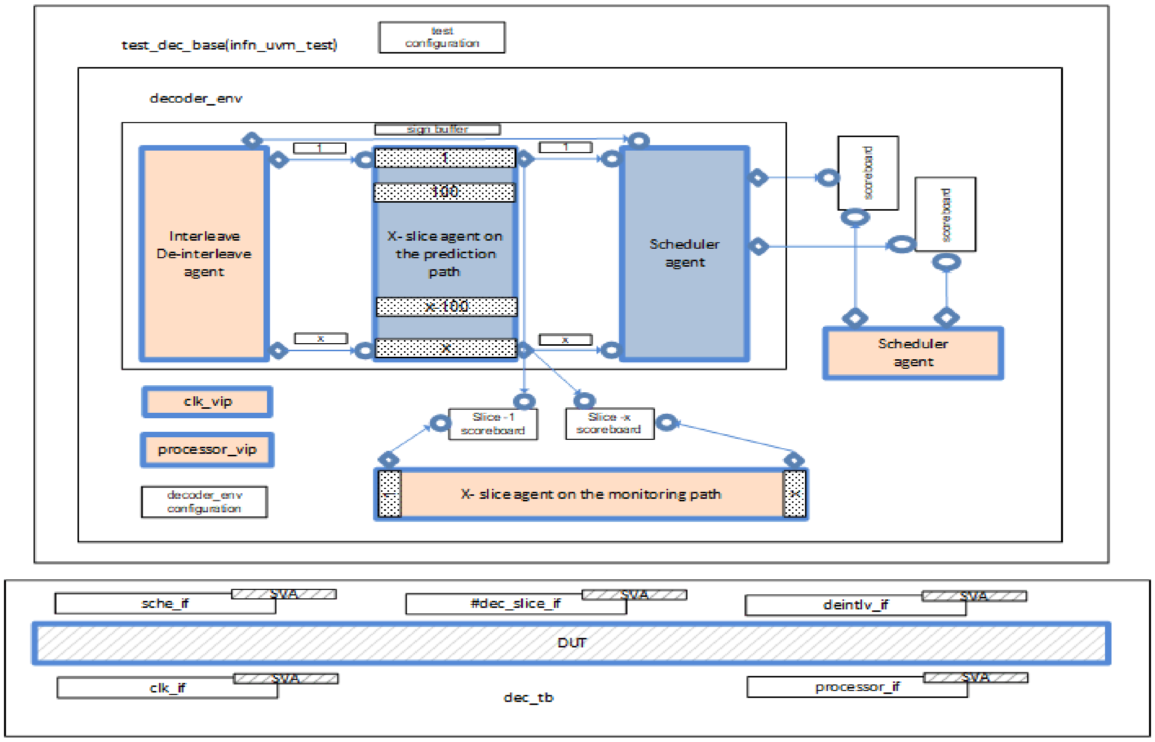


Figure 8: Receive PCS sub2chip

*Program 3 PCS Encoder sub2chip Environment Code*

```
// This is not a complete environment code – connect_phase should be a point of interest

   //Configurator Instantiation
   ps_slice_config  ps_sl_config[N];  ps_config  ps_s2c_config;  ps_enc_top_cfg ps_top_s2c_config;

   //Agent Instantiation
   ps_scheduler_Agent   ps_sche_Agent;  ps_encoder_slice_Agent    ps_enc_slice_Agent[X];
   ps_deinterleave_Agent  ps_deintlv_Agent;

   // Adding code for slice output monitoring
   ps_encoder_slice_omonitor     ps_enc_sl_0_omon;  ...   ps_encoder_slice_omonitor     ps_enc_sl_X_omon;
   virtual ps_encoder_slice_intf ps_enc_sl_0_vi; ...  virtual ps_encoder_slice_intf ps_enc_sl_X_vi;

  scoreboard#(xxx_sequence_item)    s2c_sb[X];
  scoreboard#(xxx_slice_seq_item)   enc_s2c_sb;
  scoreboard#(xxx_sequence_item)    enc_s2c_sign_sb;

endclass:ps_encoder_sub2chip_env

function void ps_encoder_sub2chip_env::connect_phase(uvm_phase phase);
     uvm_env::connect_phase(phase);
    `uvm_info("Environment", "connect", UVM_LOW);

    ps_sche_Agent.ps_sche_ref.sign_aport.connect(ps_deintlv_Agent.ps_intlv_ref.reqbuff_export);

    if(ps_s2c_config.ps_bypass == 0) begin
      for(int i=0; i<X; i++) begin
         ps_sche_Agent.ps_sche_ref.wr_aport[i].connect(ps_enc_slice_Agent[i].ps_enc_slice_mon.req_port);
         if(i%Y==0) ps_enc_slice_Agent[i].ps_enc_slice_mon.aport.connect(s2c_sb[i].rx_exp_export);
         ps_enc_slice_Agent[i].ps_enc_slice_mon.aport.connect(ps_deintlv_Agent.ps_intlv_ref.req_export);
      end

      ps_enc_sl_0_omon.aport.connect(s2c_sb[0].rx_act_export);
      ps_enc_sl_X_omon.aport.connect(s2c_sb[X-a].rx_act_export);

    end // if(ps_bypass
    else begin
      for(int i=0; i<Y; i++) begin
         ps_sche_Agent.ps_sche_ref.wr_aport[i].connect(ps_deintlv_Agent.ps_intlv_ref.reqbypass_export);
      end
    end

    ps_deintlv_Agent.ps_deintlv_mon_exp.aport.connect(enc_s2c_sb.rx_exp_export);
    ps_deintlv_Agent.ps_deintlv_mon_act.aport.connect(enc_s2c_sb.rx_act_export);
    ps_deintlv_Agent.ps_deintlv_mon_exp.sign_aport.connect(enc_s2c_sign_sb.rx_exp_export);
    ps_deintlv_Agent.ps_deintlv_mon_act.sign_aport.connect(enc_s2c_sign_sb.rx_act_export);

 endfunction :connect_phase
```

Program 4 PCS Decoder sub2chip Environment Code

```
// This is not a complete environment code – connect_phase should be a point of interest
   //Configurator Instantiation
   ps_slice_config  ps_sl_config[N];   ps_config  ps_s2c_config;    ps_dec_top_cfg ps_top_s2c_config;

   //Agent Instantiation
   ps_scheduler_Agent              ps_sche_Agent;
   ps_scheduler_Agent              ps_sche_rd_Agent;
   ps_encoder_slice_Agent          ps_enc_slice_Agent[X];
   ps_deinterleave_Agent           ps_deintlv_Agent;
   ps_decoder_slice_Agent          ps_dec_slice_Agent[X];
   ps_scheduler_intf_monitor       ps_sche_intf_omon;

   // Adding code for slice output monitoring
   ps_decoder_slice_omonitor     ps_dec_sl_0_omon; ... ps_decoder_slice_omonitor     ps_dec_sl_X_omon;
   virtual ps_decoder_slice_intf ps_dec_sl_0_vi; ... virtual ps_decoder_slice_intf ps_dec_sl_X_vi;

  scoreboard#(xxx_slice_seq_item)   dec_s2c_sb;  scoreboard#(xxx_slice_seq_item)   e2e_s2c_sb;
  scoreboard#(xxx_sequence_item)   s2c_sb[X];    scoreboard#(xxx_sequence_item)    e2e_sl_sb[X];
endclass:ps_decoder_sub2chip_env

function void ps_decoder_sub2chip_env::connect_phase(uvm_phase phase);
   uvm_env::connect_phase(phase);

   ps_deintlv_Agent.ps_deintlv_seqr.aport_i.connect(ps_sche_Agent.ps_sche_ref.req_port);
   ps_sche_Agent.ps_sche_ref.sign_aport.connect(ps_deintlv_Agent.ps_intlv_ref.reqbuff_export);
   ps_deintlv_Agent.ps_deintlv_mon_exp.sign_aport.connect(ps_sche_rd_Agent.ps_sche_ref.reqsign_export);

if(ps_s2c_config.ps_bypass == 0) begin
     for(int i=0; i<X; i++) begin
        ps_sche_Agent.ps_sche_ref.wr_aport[i].connect(ps_enc_slice_Agent[i].ps_enc_slice_mon.req_port);
        ps_enc_slice_Agent[i].ps_enc_slice_mon.aport.connect(ps_deintlv_Agent.ps_intlv_ref.req_export);
       ps_deintlv_Agent.ps_deintlv_ref.output_to_sl_aport[i].connect(ps_dec_slice_Agent[i].ps_dec_slice_mon.req_port);
        ps_dec_slice_Agent[i].ps_dec_slice_mon.aport.connect(ps_sche_rd_Agent.ps_sche_ref.reqdata_export);
        if(i%Y==0) ps_dec_slice_Agent[i].ps_dec_slice_mon.aport.connect(s2c_sb[i].rx_exp_export);
        if(i%Y==0) ps_dec_slice_Agent[i].ps_dec_slice_mon.aport.connect(e2e_sl_sb[i].rx_act_export);
        if(i%Y==0) ps_enc_slice_Agent[i].ps_enc_slice_mon.ip_aport.connect(e2e_sl_sb[i].rx_exp_export);
     end
     ps_dec_sl_0_omon.aport.connect(s2c_sb[0].rx_act_export);
     ps_dec_sl_X_omon.aport.connect(s2c_sb[X-a].rx_act_export);
   end // if(ps_s2c_config.ps_bypass == 0
else begin
     for(int i=0; i<M; i++) begin
        ps_sche_Agent.ps_sche_ref.wr_aport[i].connect(ps_deintlv_Agent.ps_intlv_ref.reqbypass_export);
     end
     ps_deintlv_Agent.ps_deintlv_ref.output_to_scherd_aport.connect(ps_sche_rd_Agent.ps_sche_ref.reqdata_export);
end

   ps_deintlv_Agent.ps_deintlv_seqr.aport_i.connect(e2e_s2c_sb.rx_exp_export);
   ps_sche_rd_Agent.ps_sche_ref.rd_aport.connect(e2e_s2c_sb.rx_act_export);
   ps_sche_rd_Agent.ps_sche_ref.rd_aport.connect(dec_s2c_sb.rx_exp_export);
   ps_sche_intf_omon.aport.connect(dec_s2c_sb.rx_act_export);
endfunction :connect_phase
```

Advantages of UVM/System Verilog based PCS environments compared to traditional verification methodology.

i) Major Advantage is the *reusability* of the UVM/System Verilog based verification infrastructure. UVM Agents and Environments are reused from module level to sub2chip level, to sub-chip level, to full-chip level verification. Additionally, reuse of all the UVM components on the transmit side and receive side of the testbench using the environment configurations (*Table 1*).

ii) PCS Algorithm standpoint the key is the codeword, especially how long the codeword is [4]. common configuration (*Program 1*) gives us *flexibility* on both Encoder and Decoder slice level Agents to have a variety of configurations on various slices in use and parallel data processing. Few *higher-level configuration generate all the lower level configurations parameters* needed for design and C models.

iii) Using *UVM set/get config_db()* mechanism all the components were programed correctly with a high level of flexibility which is one of the major advantages over traditional verification methodology described in section III. For full-chip level verification, all the configurations are used by overwriting *inline constraints*.

iv) To make use of *Transaction Level Modeling*, the interface monitor is decoupled from the protocol monitor which helped us integrate protocol monitors in the layered environments where data is processed through the C-models to generate the expected output. In the module level environment, monitors prepare the expected transaction for scoreboard compare while in sub2chip or chip-level environments it creates the transaction for the higher layer use.

v) *Reuse of all the extended test sequences and tests*. We have shown ps_encoder_slice_sequence and ps_decoder_slice_sequence as part of each agent, to highlight that only one unique sequence is needed for Encoder and Decoder slice level data path verification. Same applies to sub2chip level test sequences.

vi) Test structured is very much simplified to have all the *script generated directed tests* based on the baud rate and bit rate requirements (*Program 2*) and shared between Encoder and Decoder testing. *Sub2chip tests are also script generated* and include few more configuration values like random or equal bandwidth distribution, transmit or receive path, etc.

vii) *Integration of complete transmit environment hierarchy* as part of receive test sequence as explained in detail towards the end of section IV and captured in (*Program 4*).

viii) We could leverage *automated data checking* at the boundary of each agent. For example, the scheduler reference model has *self-checking* logic to validate the generated enable for encoder and decoder slice level agents. Interface monitor part of the scheduler agent can be used to capture the sequence of enables and can be compared against the expected sequence.

ix) *Another major advantage during the debugging,* transaction-level Encoder and Decoder output data is always available to get compared by enabling scoreboards tied at the boundaries of Encoder and Decoder slice level agents. In a nutshell, all the debugging hooks available at the component level are used as-is. This accelerates the whole data-path bring up effort.

x) *Functional coverage on configuration and scenarios* were also very simplified with UVM/System Verilog based environment. It gave us high confidence in our verification goals beforehand.

xi) *Environment* developed for PCS sub2chip in both transmit and receive directions are completely *reused on both receive and transmit sub-chip level testbench*. Which is eventually included as part of the full-chip level testbench.

xii) *Latency measurement is simplified* by porting monitors at the interface to measure the time taken for any transaction from a given source to destination path at all the level of testbench hierarchy.

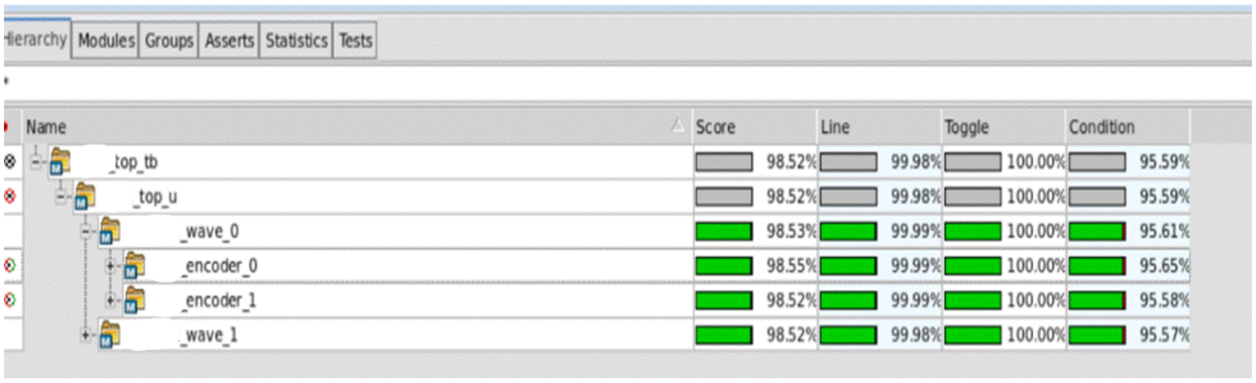
xiii) The fully populated full-chip level environment may contain multiple instances of PCS sub2chip environments. *Seamless integration of multiple instances* is only possible due to our selection of UVM/System Verilog based methodology.

## V. Coverage and Performance

Coverage goals for Encoder and Decoder sub2chip under the respective transmit and receive FEC sub-chip are achieved using a combination of random and directed tests.

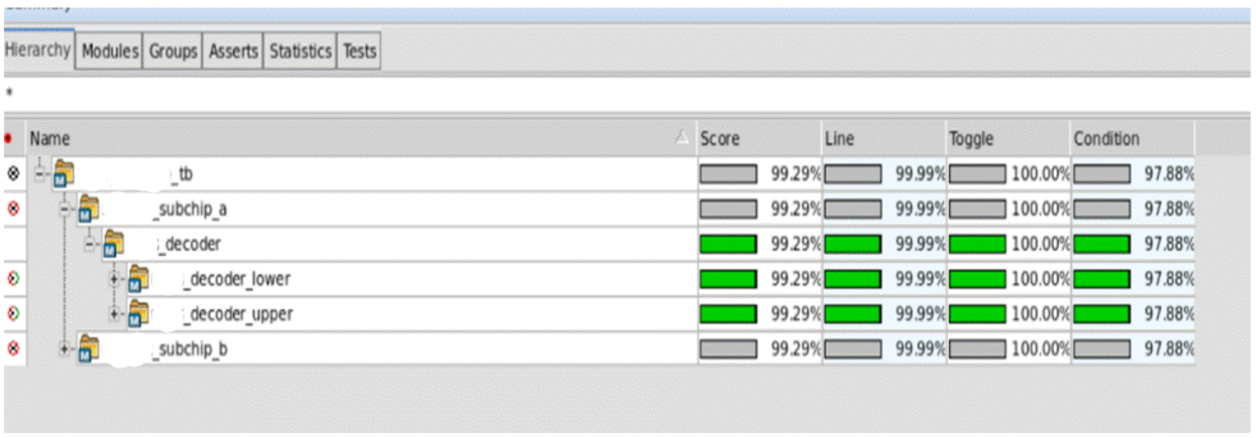
In the snippet ( *Figure 9* and *Figure 10* ), we have captured one instance of a transmit and a receive sub-chip hierarchy, highlighting coverage of two instances of Encoder and Decoder sub2chip.

Performance tests were covered as part of directed tests, where the focus was around the throughput and data integrity during the simulation for multiple, multi-frames. Another key test category was error recovery.



| Name | Score | Line | Toggle | Condition |
|---|---|---|---|---|
| _top_tb | ▭ 98.52% | ▭ 99.98% | ▭ 100.00% | ▭ 95.59% |
| _top_u | ▭ 98.52% | ▭ 99.98% | ▭ 100.00% | ▭ 95.59% |
| _wave_0 | 🟩 98.53% | 🟩 99.99% | 🟩 100.00% | 🟩 95.61% |
| _encoder_0 | 🟩 98.55% | 🟩 99.99% | 🟩 100.00% | 🟩 95.65% |
| _encoder_1 | 🟩 98.52% | 🟩 99.99% | 🟩 100.00% | 🟩 95.58% |
| _wave_1 | 🟩 98.52% | 🟩 99.98% | 🟩 100.00% | 🟩 95.57% |

*Figure 9: PCS Encoder sub2chip coverage*



| Name | Score | Line | Toggle | Condition |
|---|---|---|---|---|
| _tb | ▭ 99.29% | ▭ 99.99% | ▭ 100.00% | ▭ 97.88% |
| _subchip_a | ▭ 99.29% | ▭ 99.99% | ▭ 100.00% | ▭ 97.88% |
| _decoder | 🟩 99.29% | 🟩 99.99% | 🟩 100.00% | 🟩 97.88% |
| _decoder_lower | 🟩 99.29% | 🟩 99.99% | 🟩 100.00% | 🟩 97.88% |
| _decoder_upper | 🟩 99.29% | 🟩 99.99% | 🟩 100.00% | 🟩 97.88% |
| _subchip_b | ▭ 99.29% | ▭ 99.99% | ▭ 100.00% | ▭ 97.88% |

*Figure 10: PCS Decoder sub2chip coverage*

## VI. Summary And Results

Traditionally, DSP algorithms are verified using MATLAB models and C-models. Our decision to verify the complex design blocks using the UVM/System Verilog IEEE standards capabilities helped us reuse all the UVM Environments, Agents, Configurations, Virtual Interfaces and Sequences [1] from block-level to full-chip verification. We created a common configuration class following constraint random methodologies [2], which was shared between Encoder and Decoder slice level environments. Similarly, a common configuration was used at the sub2chip level for receive and transmit path. We created only one unique base sequence for each PCS sub2chip on transmit and receive direction and all the extended tests were reused between PCS Encoder and Decoder sub2chip verification. We reused a sub2chip level environment as is at sub-chip and full-chip level verification testbench (*Table 1*).

*Table 1: Reuse of UVM Objects, Components and Sequences*

| **UVM Components** | **Module Level** | **Sub2chip level** | **Sub-chip level** | **Full-chip level** | **Receive / Transmit** |
|---|---|---|---|---|---|
| Configurations | ✓ | ✓ | ✓ | ✓ | ✓ |
| Monitors | ✓ | ✓ | ✓ | ✓ | ✓ |
| Sequences | ✓ | ✓ | ✓ | ✓ | |
| Adaptive Sequences | | ✓ | ✓ | ✓ | |
| Scoreboards | ✓ | ✓ | ✓ | ✓ | ✓ |
| Agents on receive and transmit | ✓ | ✓ | ✓ | ✓ | ✓ |
| Environments | | ✓ | ✓ | ✓ | |

Complex symbol level data distribution check was achieved at sub2chip level testbench environment, by using several slice level scoreboards in parallel along with the end to end scoreboard checks. Having C-models integrated as part of the slice Agent eliminated any symbol level encoding and decoding issues on the monitoring path. Virtual Interfaces helped us achieve the effective integration of monitors at each level of the testbench hierarchy. The standalone self-checking scheduler reference model immensely helped find any scheduling issues for any design configuration. The importance of scheduling is verified with the accuracy of the number of enables generated in order. Effective marker generation and monitoring functions were implemented towards the interleave/deinterleave interface to validate the number of bits sent out or received in the fixed timeslots.

Implementing a reusable UVM/System Verilog based testbench environment helped achieve the verification goals on time. Contribution towards the sub2chip level verification efforts can be evaluated by measuring the reuse of test stimulus on both receive and transmit direction. It has immensely helped to verify configuration heavy design blocks like PCS. Leveraging the environment components, we could exercise the system level scenarios at the full-chip level without changing any sub-chip level environment code. It has helped to close the coverage with close to 100% coverage goals. Overall, reusable environment components validated bit-level integrity in both transmit and receive direction for the PCS sub2chip and at full-chip level. Performance and Error recovery were the other two critical test scenarios effectively verified.

## VII. References

[1] 1800.2-2017 - IEEE Standard for Universal Verification Methodology Language Reference Manual
[2] 1800-2017 - IEEE Standard for System Verilog--Unified Hardware Design, Specification, and Verification Language
[3] Infinera Proprietary Algorithms – M Torbatian, D Chan, HH Sun, S Thomson, KT Wu - *US Patent App. 16/152,349, 2019*
[4] www.infinera.com/blog/long-codewords-the-secret-to-successful-probabilistic-constellation-shaping
[5] www.infinera.com/white-paper/The-Ultimate-Guide-to-Nyquist-Subcarriers