Channel Modelling in Complex Serial IPs

Jayesh Ranjan Majhi
Saravana Balakrishnan
Navnit Kumar Kashyap
Cadence Design Systems, Bangalore

***Abstract*- In High-speed Serial Interface IPs (e.g. PCIe, USB etc.) there are numerous reasons for signal distortion or deformation due to practical environmental behaviour or material behaviour. The verification of Serial IP Controllers become more complex when simulating with PHY-Model (for faster results) as testbench needs to mimic the Analog PHY output behaviour. The receiver end needs to be robust enough to handle all type of signal distortion which falls under the IP Specification umbrella. The data received by a high-speed Serial IP have a large amount of signal distortion due to multiple factors like lane-to-lane skew, PPM variation in clocks, Jitter, Bit Error Rate (BER) in data, Bit shift caused due to CDR circuit. So, we need to exercise the design with data consisting of all the signal distortion parameters.**

In this paper, we have demonstrated methodology for stimulus creation with real life scenarios to effectively verify a high-speed Serial Controller IP.

Keywords - PPM, Jitter, Bit Error, Bit Shift, Skew, CDR, DUT, PHY, PIPE, PCIe, USB

I. Introduction

To create a robust IP, we need to create stimulus with real-life data that has Lane-to-Lane Skew, Clock PPM variations, Lane Jitter, Bit-Error-Rate (BER) and Bit shift. Across industry there are no push button mechanisms provided to enable real-life scenario modelling.

A "Channel Model" that can inject/add Skew, Jitter, PPM, BER and Bit-Shift to the received data would be an ideal solution. We have accomplished this Channel Model through a sequence of channel models which provides mechanism to create realistic modelling in the received data.

The Channel Model is in an always-on mode, across complete regression test suite, allowing us to run tests effectively such that the DUT (represented as "Controller" below) is always stressed.

II. Channel Modelling in Complex Serial IPs

A. Proposed Methodology

To create a channel module which can be just plugged in a testbench to generate all the possible real-life signal distortion in the received data and clock in all simulations. This model will enable us to stress DUT with parameters such as Skew/Bit-error/PPM/Jitter/Bit-Shift.

B. Channel Models

The received data from link partner is manipulated by passing through the following modules in sequence

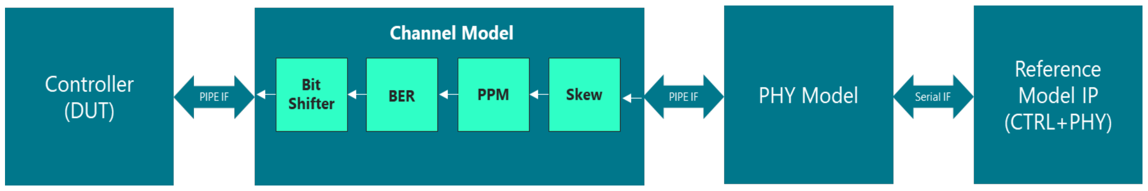


Figure 1. Data Flow through channel models

- *Reference Model IP (Controller + PHY)*
    - The Data transmitted by Link Partner Device (BFM) is good data without any signal distortion.
    - The raw good data from Link Partner Device (BFM) is passed through the above channel models in sequence
- *Skew Insertion Model*
    - Adds Static and Dynamic Skew.
    - The data received on each lane is delayed depending on the programmed skew value.
- *PPM Model*
    - The output data from skew model is passed to the PPM and Jitter model for PPM and Jitter addition.
    - The PPM and Jitter is added to the received data by generating the recovered clock with variation in clock period.
- *BER (Bit Error Rate) Injector*
    - This model has a speed sensitive bit error injection mechanism which flips a bit at regular interval in controlled random manner.
    - The model also predicts the expected error and downgrade them on the fly.
- *Bit Shifter*
    - This model essentially mimics the CDR circuit output data which has the bit shifted data.
    - The data coming out of Bit-Shifter is fed into the DUT.
- *Controller DUT*
    - The final data received by DUT has Skew, PPM, Jitter, Bit Error Rate (BER) and Bit shifting.
    - This matches with the real-life data received by any high-speed Serial IP.

III. IMPLEMENTATION DETAILS

A. *Skew Insertion Model*

This model adds both static and dynamic skew in each lane. The received data from link partner or reference model IP is delayed depending on the programmed skew value.

a. *Static Skew*

Static skew is inserted using skew insert model. This model is plugged in between BFM and DUT in the RX Path of each lane. If skew insertion is enabled for a lane, the data received from BFM is passed through skew insert block which adds the programmed skew value. The skewed data (output

from skew insert block) is connected to the PPM model, which sees different skew on different lanes. The static skew is re-calculated every time the link goes idle. This is done in dynamic skew insert component which monitors the link LTSSM and randomizes the static skew.
The below table explains the connections done between Skew Insert and PPM model at the testbench top.

Rx (data_in) [42 bits]	**Lane 0**	**Rx (data_out) [42 bits]**
From BFM	**skew_ins[0]**	**To DUT**
skew_ins_pipe_rxvalid[0] = pcie_pipe_passive_if.RxValid[0]		ppm_if.pipe_rxvalid[0] = skew_ins_pipe_rxvalid[0]
skew_ins_pipe_rxdatavalid[0] = pcie_pipe_passive_if.RxDataValid[0]		ppm_if.pipe_rxdatavalid[0] = skew_ins_pipe_rxdatavalid[0]
skew_ins_pipe_rxstartblock[0] = pcie_pipe_passive_if.RxStartBlock[0]		ppm_if.pipe_rxstartblock[0] = skew_ins_pipe_rxstartblock[0]
skew_ins_pipe_rxsyncheader[1:0] = pcie_pipe_passive_if.RxSyncHeader[1:0]		ppm_if.pipe_rxsyncheader[1:0] = skew_ins_pipe_rxsyncheader[1:0]
skew_ins_pipe_rxdata[31:0] = pcie_pipe_passive_if.RxData[31:0]		ppm_if.pipe_rxdata[31:0] = skew_ins_pipe_rxdata[31:0]
skew_ins_pipe_rxdatak[3:0] = pcie_pipe_passive_if.RxDataK[3:0]		ppm_if.pipe_rxdatak[3:0] = skew_ins_pipe_rxdatak[3:0]
skew_ins_pipe_rxelecidle[0] = pcie_pipe_passive_if.RxElecIdle[0]		ppm_if.pipe_rxelecidle[0] = skew_ins_pipe_rxelecidle[0]
Rx (data_in) [42 bits]	**Lane 1**	**Rx (data_out) [42 bits]**
From BFM	**skew_ins[1]**	**To DUT**
skew_ins_pipe_rxvalid[1] = pcie_pipe_passive_if.RxValid[1]		ppm_if.pipe_rxvalid[1] = skew_ins_pipe_rxvalid[1]
skew_ins_pipe_rxdatavalid[1] = pcie_pipe_passive_if.RxDataValid[1]		ppm_if.pipe_rxdatavalid[0] = skew_ins_pipe_rxdatavalid[1]
skew_ins_pipe_rxstartblock[1] = pcie_pipe_passive_if.RxStartBlock[1]		ppm_if.pipe_rxstartblock[1] = skew_ins_pipe_rxstartblock[1]
skew_ins_pipe_rxsyncheader[3:2] = pcie_pipe_passive_if.RxSyncHeader[3:2]		ppm_if.pipe_rxsyncheader[3:2] = skew_ins_pipe_rxsyncheader[3:2]
skew_ins_pipe_rxdata[63:32] = pcie_pipe_passive_if.RxData[63:32]		ppm_if.pipe_rxdata[63:32] = skew_ins_pipe_rxdata[63:32]
skew_ins_pipe_rxdatak[7:4] = pcie_pipe_passive_if.RxDataK[7:4]		ppm_if.pipe_rxdatak[7:4]= skew_ins_pipe_rxdatak[7:4]
skew_ins_pipe_rxelecidle[1] = pcie_pipe_passive_if.RxElecIdle[1]		ppm_if.pipe_rxelecidle[1] = skew_ins_pipe_rxelecidle[1]

Table 1. Top level connections between Skew and PPM Model

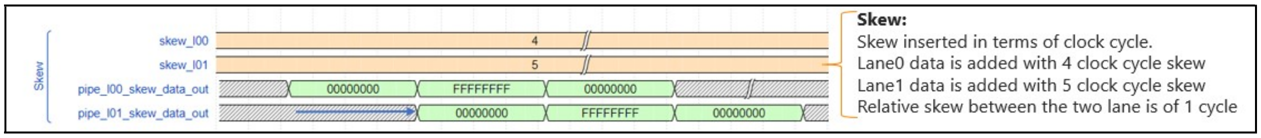
The below snippet shows the instantiation of skew insert module at the tb_top. All the signals that must be skewed are passed to the data_in input of skew_insert module

Skew Insert Verilog Model: skew_insert.v
Skew Insert Block Instantiation: //Create this instantiation in separate file sve/cdn_pcie_hpa_skew_insert.sv and include it in cdn_pcie_hpa_tb_top.sv

```
        skew_insert #(.width(42)) skew_insert_Link0_Lane0
          (
           .pipe_clk(top_if.linkIf[0].pcieIf_0.PIPE_PCLK),
           .pipe_rst_n(top_if.linkIf[0].pcieIf_0.PIPE_PHY_RESET_N),
//NOTE: Skew_insert block is aligned with pipe_clk inputs now. so it will give 1 less number of skew inserted.
//To overcome this, add +1 to skew_sel
           .skew_sel (top_if.linkIf[0].pcieIf_0.skew_sel_lane[0]+1),                // Connect this to i_cfg.max_skew_for_skew_insert_block[0]
           .data_in(top_if.linkIf[0].pcieIf_0.data_in_0),
           .data_out(top_if.linkIf[0].pcieIf_0.data_out_0)
          );
```

b. *Dynamic Skew (Skip Addition/Deletion)*

Skip symbols received by DUT are added/deleted randomly using StartPacket callback in the BFM. Whenever active_bfm transmits skip, the skip symbols are randomly added/removed based on the current speed and the current running skew limit. This is done on top of static skew that is already present. The DUT is expected to detect the skips and deskew properly and no failure should be observed in the simulation.



Skew:
Skew inserted in terms of clock cycle.
Lane0 data is added with 4 clock cycle skew
Lane1 data is added with 5 clock cycle skew
Relative skew between the two lane is of 1 cycle

Figure 2. Skew – Example Waveform

B. *PPM/Jitter Model*

In real life no clock is 100% accurate. Every clock has some amount of deviation from nominal value, these deviations are calculated in terms of PPM (Parts per million). 1 PPM means 1 clock deviation in 1 million pulse of clock. In most of the high-speed serial IP, clocks are not shared between transmitter and receiver. Therefore, the mismatch of transmitter and receiver frequency is apparent and big concern. Each IP Specification is designed to tolerate this divergence of clock frequency between transmitter and receiver. If this is not addressed robustly during IP design, it is possible that the receiver end would miss some clock cycles of data in between.
Therefore, to create a robust IP there is definite need to stress the DUT by generating these scenarios randomly in each simulation. The PPM sub-block which we have developed is able to generate clock with programed PPM value and clock frequency.

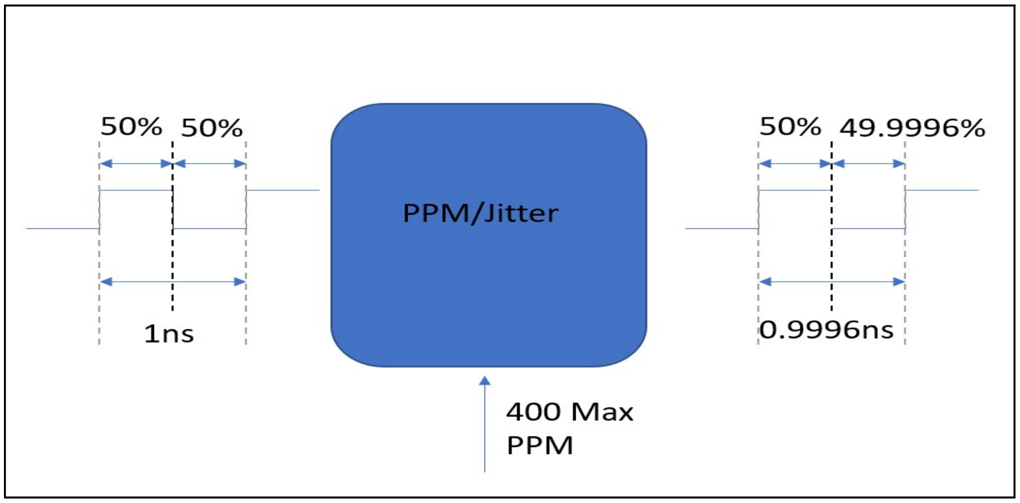


Figure 3. PPM/Jitter Model

This model generates a clock on the programmed frequency with PPM and Jitter. It is designed to generate a dynamic PPM level clock frequency. The dynamic PPM level logic mimics the effect of PPM as well as Jitter. The dynamic PPM logic keeps on changing the clock period within the allowed range.

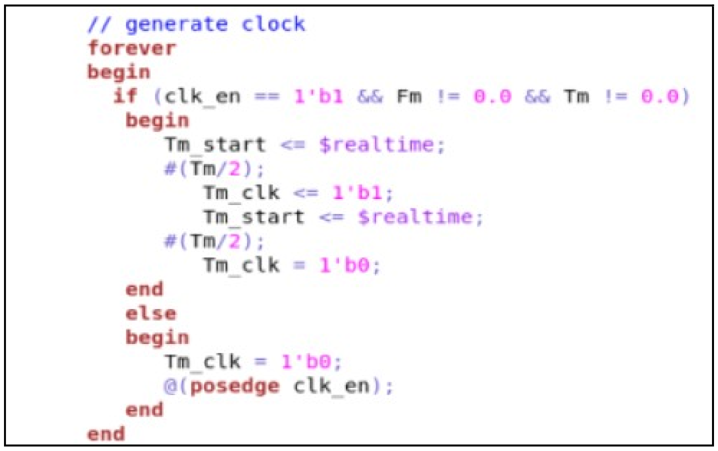
Formula used for calculating the change in frequency for programed PPM level is mentioned below:

df = (f *600)/ 106
dTp = (1/df)

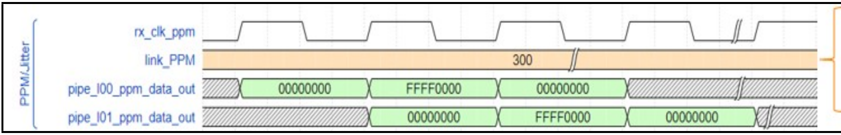
Define for formula:

df	= change in frequency due to PPM
f	= desired frequency
dTp	= change in time period due to PPM

The change time period (dTp) is then added or subtracted from the desired time period (1/df) which is then used to generate the new clock with PPM/Jitter.



```
// generate clock
forever
begin
  if (clk_en == 1'b1 && Fm != 0.0 && Tm != 0.0)
   begin
      Tm_start <= $realtime;
      #(Tm/2);
         Tm_clk <= 1'b1;
         Tm_start <= $realtime;
      #(Tm/2);
         Tm_clk = 1'b0;
   end
   else
   begin
      Tm_clk = 1'b0;
      @(posedge clk_en);
   end
end
```



PPM/Jitter:
Each lane have PPM of 300 PPM at the given instance. Each data cycle period is synchronized with the rx_clock which consists of PPM/Jitter.

Figure 4. PPM/Jitter – Sample Waveform showing clock with PPM + Jitter

C. *BER (Bit Error Rate)*

In high speed IP there is a considerable chance that some bit of data stream will get flipped due to material and environmental properties.
This model contains logic to inject bit error at regular intervals. Regulated Bit Error is injected randomly in Ordered Set during every speed change scenario. This is accomplished using Bit Error Rate injection model.

The Bit Error model implements a counter that keeps a count of number of bits received in the data stream. Once the programed threshold level is reached, one bit is flipped randomly in the data stream. The bit error model is made configurable to mask the bit error injection in some phases of simulation.

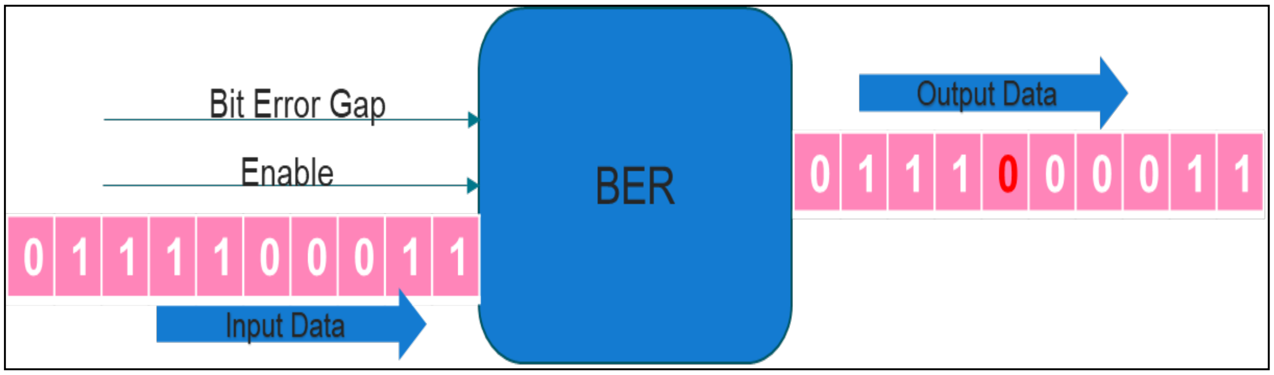
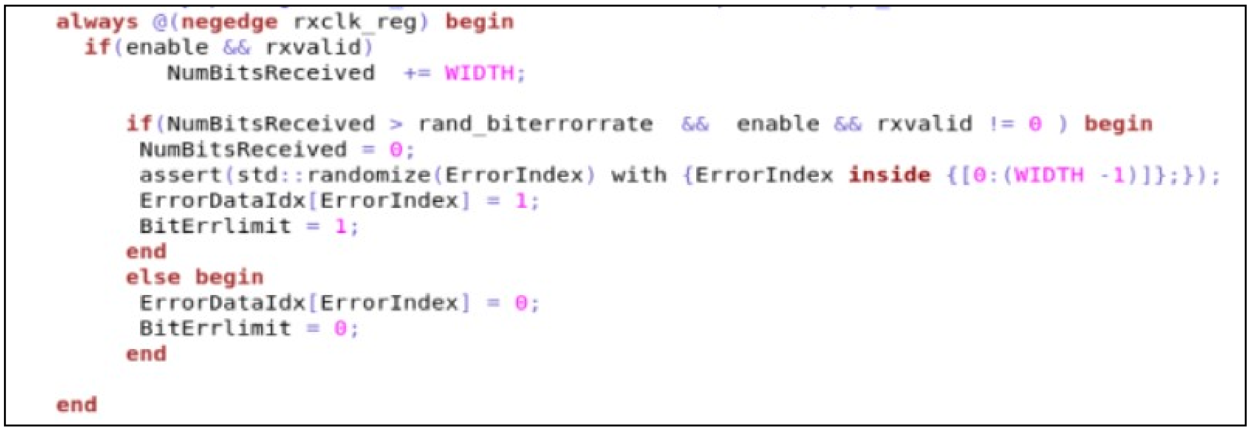


Figure 5. BER Model

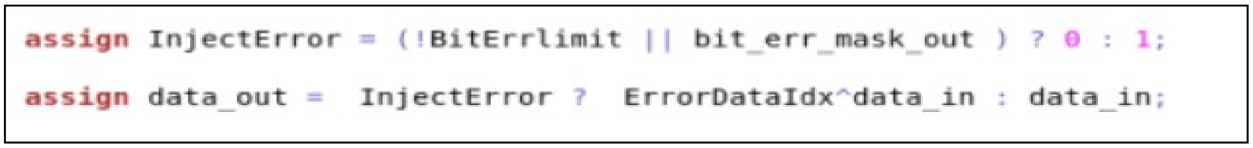
The basic logic for bit error injection is shown in the below piece of code



```
always @(negedge rxclk_reg) begin
  if(enable && rxvalid)
        NumBitsReceived   += WIDTH;

     if(NumBitsReceived > rand_biterrorrate  &&  enable && rxvalid != 0 ) begin
      NumBitsReceived = 0;
      assert(std::randomize(ErrorIndex) with {ErrorIndex inside {[0:(WIDTH -1)]};});
      ErrorDataIdx[ErrorIndex] = 1;
      BitErrlimit = 1;
     end
     else begin
      ErrorDataIdx[ErrorIndex] = 0;
      BitErrlimit = 0;
     end

end
```



```
assign InjectError = (!BitErrlimit || bit_err_mask_out ) ? 0 : 1;

assign data_out =  InjectError ?  ErrorDataIdx^data_in : data_in;
```

The below waveform snapshot shows the bit error injected data on lane 0 and lane 1

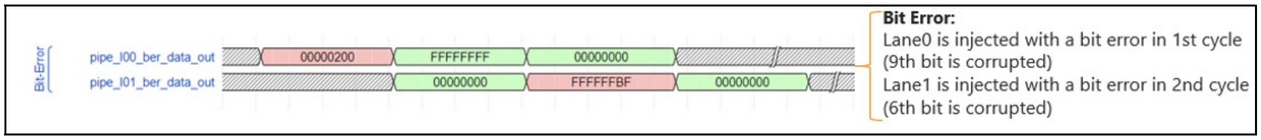

Bit-Error
pipe_l00_ber_data_out
pipe_l01_ber_data_out
00000200
FFFFFFFF
00000000
00000000
FFFFFFBF
00000000
Bit Error:
Lane0 is injected with a bit error in 1st cycle (9th bit is corrupted)
Lane1 is injected with a bit error in 2nd cycle (6th bit is corrupted)

Figure 6. BER Model

D. *Bit Shifter*

In Serial IP, the PHY uses a CDR (clock and data recovery) circuit to recover the data and clock from incoming streams of binary inputs. During this process there is possibility of bit shifting of data along the data bus width. Therefore, it's not obvious that the valid data starts from bit 0 of data bus.
The Bit shift model shifts the incoming data based on the programmed number of shifts. The incoming clock cycle of data is left shifted with the programed bit-shift value and bits overflowed are stored in a temporary variable. In the next cycle, the overflown data is appended to the incoming data on LHS after applying the shift to the current data.

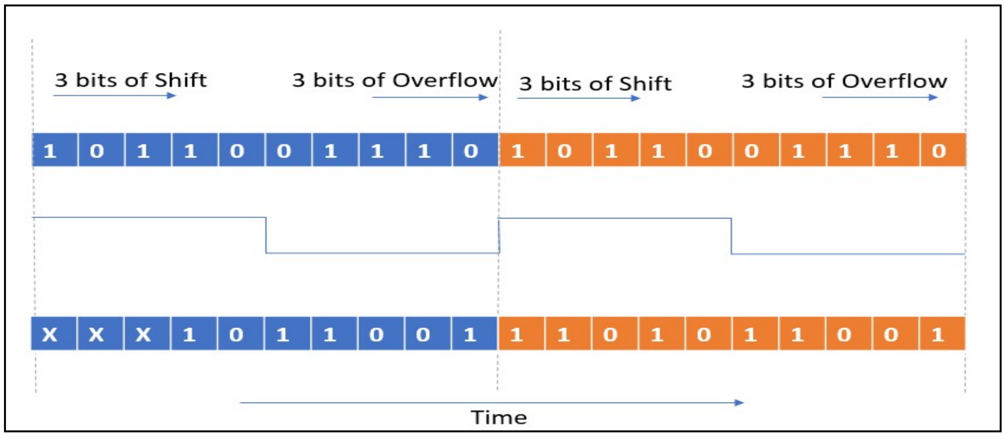
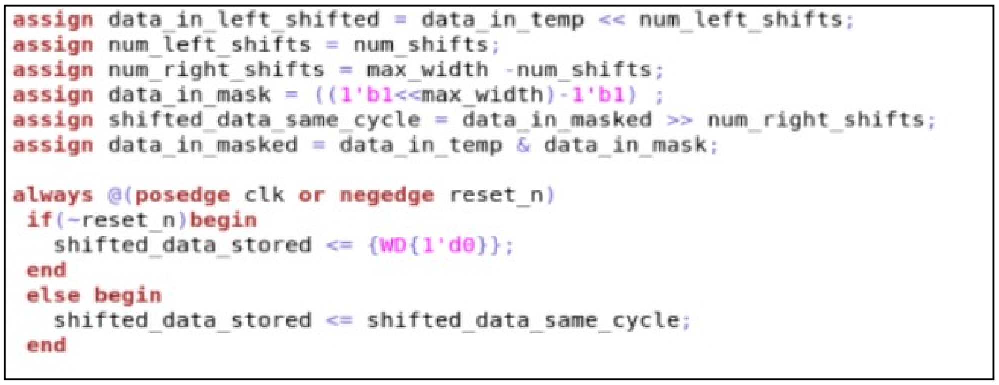


Figure 7. Bit Shift Example

The below piece of code and the following waveform snapshot demonstrates the bit shifting



```
assign data_in_left_shifted = data_in_temp << num_left_shifts;
assign num_left_shifts = num_shifts;
assign num_right_shifts = max_width -num_shifts;
assign data_in_mask = ((1'b1<<max_width)-1'b1) ;
assign shifted_data_same_cycle = data_in_masked >> num_right_shifts;
assign data_in_masked = data_in_temp & data_in_mask;

always @(posedge clk or negedge reset_n)
 if(~reset_n)begin
   shifted_data_stored <= {WD{1'd0}};
 end
 else begin
   shifted_data_stored <= shifted_data_same_cycle;
 end
```

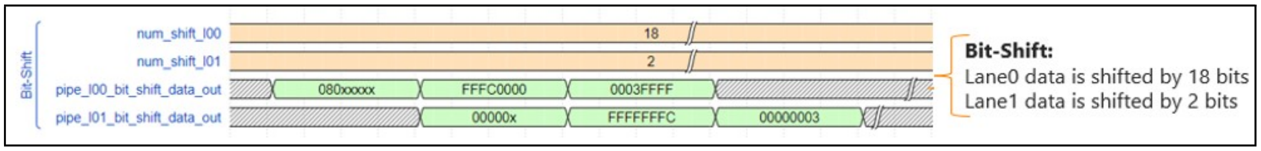

Bit-Shift
num_shift_l00
num_shift_l01
pipe_l00_bit_shift_data_out
pipe_l01_bit_shift_data_out
18
2
080xxxxx
FFFC0000
0003FFFF
00000x
FFFFFFFC
00000003
Bit-Shift:
Lane0 data is shifted by 18 bits
Lane1 data is shifted by 2 bits

Figure 8. Bit Shift Example Waveform

IV. Results

- The proposed method with the sequence of channel models provides 4x reduction in effort when compared to directed testing.
- Adapting this approach has certainly helped in exposing many critical pre-silicon RTL bugs and reduced the unexpected faults & failures.

Task	Effort	Comments
Architecture Development/Reviews	~45 days	Effort for re-usable testbench model development and reviews
Testing	~30 days	Regression failure debug and analysis
Bugs Found	Many	RTL bugs pertaining to handling of Skew, Jitter, Bit Shift etc.
Code Coverage Metrics	~20 days	Minimal effort in hitting corner case scenarios

V. CONCLUSION

Designing a robust high-speed serial IP is a critical need due to the increased complexities and functionality in AI, IOT, Automotive, Cloud and Storage applications. So, it makes sense to verify the high-speed serial IPs with real life signal distortion scenarios that are common at high frequencies. The content of this paper comes from the most common issues faced by the serial IPs operating at high frequencies.
The proposed method helped in mimicking the real-life scenarios that any complex Serial IP would face. Adapting this approach has certainly helped in bringing down the unexpected faults and failures. The proposed concept can be extended and adapted for verification of any complex Serial IP.
Thus, a channel model that mimics and adds real life signal distortion factors provides an ideal solution to verify complex high-speed Serial IPs.

VI. References

1. PCI Express® Base Specification Revision 6.0 : https://members.pcisig.com/wg/PCI-SIG/document/previewpdf/16609
2. PHY Interface For the PCI Express,SATA, USB 3.1,DisplayPort, and Converged IO Architectures Version 6.0 **:** https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/phy-interface-pci-express-sata-usb30-architectures-3-1.pdf