Parameterized and Re-usable Jitter Model for Serial and Parallel Interfaces

Amlan Chakrabarti, Member of Technical Staff, AMD India Pvt. Ltd., Bangalore, India
(amlan.chakrabarti@amd.com)

Malathi Chikkanna, Member of Technical Staff, AMD India Pvt. Ltd., Bangalore, India
(malathi.chikkanna@amd.com)

***Abstract*—Jitter is the short term variation of a signal with respect to its ideal position in time. When verifying any serial or parallel interface, it is critical to model this jitter during data transmission. A jitter model modifies the width of a data bit when the master agent transmits data to a slave, or when the designunder test (DUT) receives data. In packet based protocols that employ a serial link to communicate to the link partner, many times the clock is not sent along with the data. In the physical (PHY) layer, when the data is received, the clock needs to be recovered from the data. Modeling the bit width variation on the link is crucial to exhaustively verifying the clock-data recovery (CDR) logic. If the DUT has an elastic buffer, such a variation can expose problems such as inadequate depth of the elastic buffer. Traditionally, we have developed a separate jitter model for each new interface. In this paper, we describe an approach to develop a parameterized and reusable jitter model using universal verification methodology (UVM) that can be used for any serial and parallel interface.The jitter model developed using this approach on a serial interface found 2 critical bugs that could have potentially caused a silicon respin.**

Keywords—Jitter Model;Clock-Data Recovery(CDR);Parameterized;Reusable;UVM.

I. INTRODUCTION

Jitter refers to the variation of a signal in time with respect to its ideal position. Jitter occurs in real systems from sources such as phase-locked loops (PLLs), random thermal noise from a crystal, signal transmitters, traces, and cables. Jitter on data can result in incorrect data being captured at the receiver. A jitter model modifies the width of a data bit during data transmission. On serial interfaces, many times the clock is not sent along with the data. This is done to reduce the pin count and hence the package cost. In such protocols, when the data is received, the clock needs to be recovered from the data using a clock-data-recovery (CDR) logic. To exhaustively verify the CDR logic, it is essential to model the bit width variation on the link. Through this modeling, problems such as inadequate depth of an elastic buffer in the DUT can be exposed.

For a parallel interface on the board, different bits of a data bus can experience varying amounts of jitter. The distributions in the jitter model mentioned above enable us to mimic this behavior. Thus, a jitter model enables us to determine the maximum amount of jitter which the designundertest (DUT) can tolerate.

II. TYPES OF JITTER MODELING

The variation introduced by the jitter model can be deterministic (sinusoidal, triangular), random(Gaussian), or a combination of both.
Sinusoidal jitter can be modeled through the following equation:
sj_offset + (sj_ampl*$sin(2*3.1416*curr_sj_freq*$realtime))
where

- sj_offset = an initial offset of the jitter sinusoid
- sj_ampl = amplitude of the jitter sinusoid
- curr_sj_freq = frequency of the jitter sinusoid

Random jitter can be modeled as
rj_offset + $dist_normal(rj_seed, 0, rj_stdev)*10/1000
where

- rj_offset = an initial offset for the random jitter
- rj_seed = initial seed for the normal distribution
- rj_stdev = standard deviation for the density function

III. PAST APPROACH FOR MODELING JITTER

Figure 1 depicts the way jitter was modeled for different interfaces in the past.

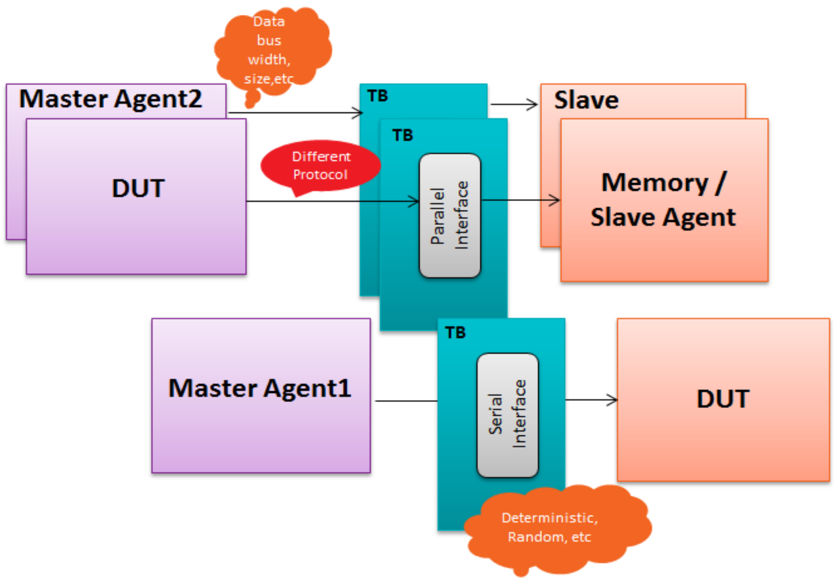


Figure 1: Past approach for modeling jitter.

In this methodology, a separate jitter model is developed for each interface. There can be significant differences between the interfaces, such as serial/parallel, difference in data bus port names, and bus widths. Also the protocols to which the interfaces comply can be entirely different. Hence, reusing a jitter model across different interfaces is not easy. Furthermore, since jitter is modeled in the interface, the model is always enabled. Therefore, it requires static allocation of memory.

IV. PROPOSED METHODOLOGY FOR DEVELOPING A JITTER MODEL

Figure 2 depicts our proposed approach for developing a jitter model.

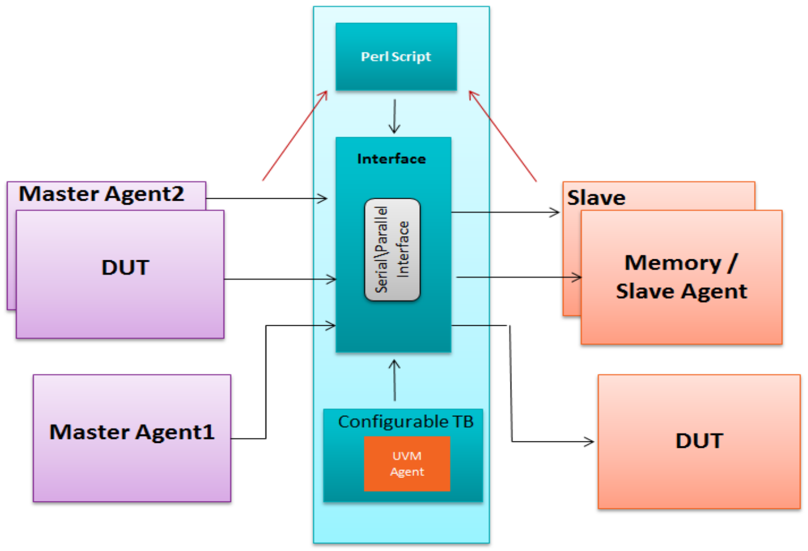


Figure 2: Proposed approach to model jitter.

In this proposed approach, the jitter model is developed using Universal Verification Methodology(UVM). This approach is parameterized, reusable, andscalable.

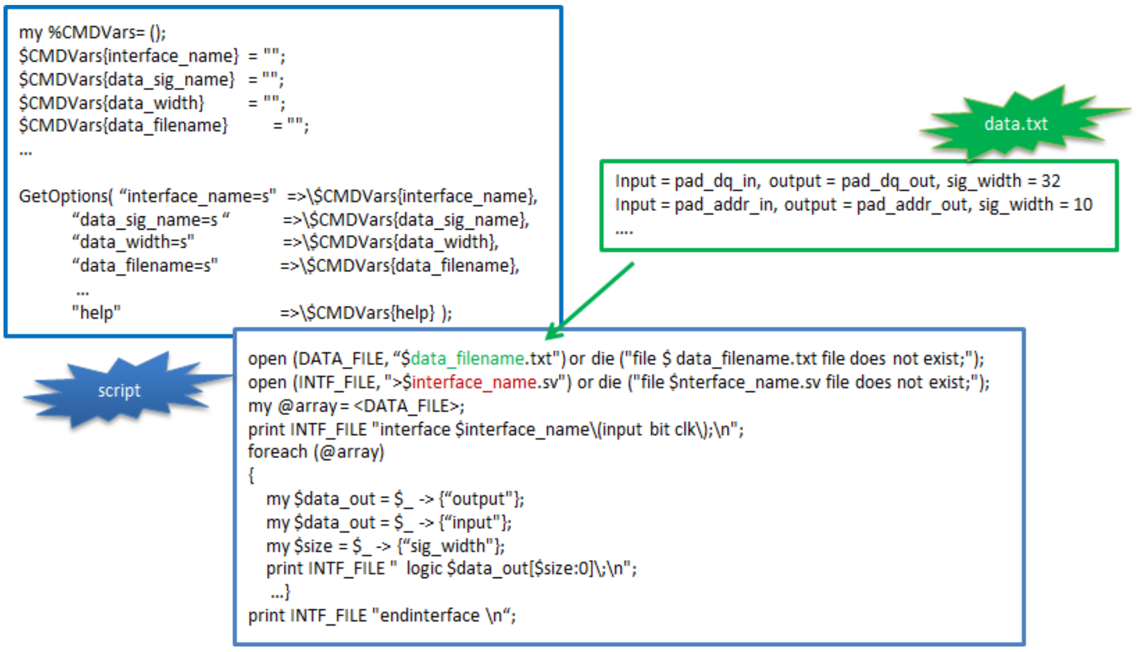
V. COMPONENTS OF THE SOLUTION

A typical jitter model has the following parameters:

a) Parameters of the interface on which the jitter has to be injected. These include the data bus port names and the data bus width

b) Testbench parameters, such as type of jitter and enable/disable of the jitter model.

In our proposed approach for the interface parameters, we have developed a Perl script which takes the interface signals and their properties as parameters from a file(or command line), and automatically generates the interface file based on these parameters. The file consists of the inputs, outputs, and thebus widths. The script reads these values from the file and generates the interface file as shown in Figure 3 below.



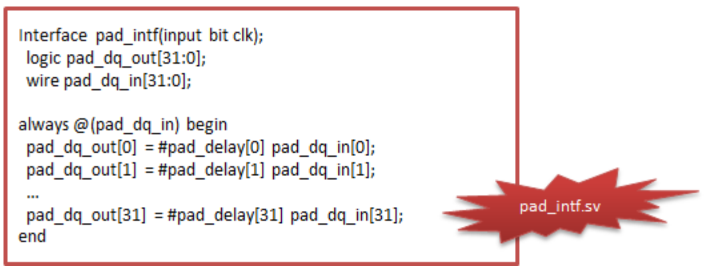
```
my %CMDVars= ();
$CMDVars{interface_name}  = "";
$CMDVars{data_sig_name}  = "";
$CMDVars{data_width}       = "";
$CMDVars{data_filename}       = "";
...

GetOptions( “interface_name=s"  =>\$CMDVars{interface_name},
        “data_sig_name=s “         =>\$CMDVars{data_sig_name},
        “data_width=s"              =>\$CMDVars{data_width},
        “data_filename=s"          =>\$CMDVars{data_filename},
        ...
        "help"                         =>\$CMDVars{help} );
```

```
Input = pad_dq_in,  output = pad_dq_out, sig_width = 32
Input = pad_addr_in, output = pad_addr_out, sig_width = 10
....
```

```
open (DATA_FILE, “$data_filename.txt") or die ("file $ data_filename.txt file does not exist;");
open (INTF_FILE, ">$interface_name.sv") or die ("file $nterface_name.sv file does not exist;");
my @array = <DATA_FILE>;
print INTF_FILE "interface $interface_name\(input bit clk\);\n";
foreach (@array)
{
   my $data_out = $_ -> {“output"};
   my $data_out = $_ -> {“input"};
   my $size = $_ -> {“sig_width"};
   print INTF_FILE "  logic $data_out[$size:0]\;\n";
    ...}
print INTF_FILE "endinterface \n“;
```

Figure 3: Script for generating the interface file.

Also, the script automatically adds the logic to insert the delay between the inputs and the outputs. Figure 4 shows a snippet of an example interface file. The interface file is then hooked between the testbench and the design under test.



```
Interface  pad_intf(input bit clk);
 logic pad_dq_out[31:0];
 wire pad_dq_in[31:0];

always @(pad_dq_in) begin
 pad_dq_out[0]  = #pad_delay[0] pad_dq_in[0];
 pad_dq_out[1]  = #pad_delay[1] pad_dq_in[1];
 ...
 pad_dq_out[31]  = #pad_delay[31] pad_dq_in[31];
end
```

Figure 4: Example interface file.

The jitter model is developed as an UVM agent, which performs the bit width variation based on the testbench parameters. This is described in sections VII and VIII.

VI. UVM TESTBENCH ENVIRONMENT

Figure 5 below shows the UVM testbench developed with the proposed approach. The DUT can have a serial or a parallel interface. The environment is comprised of agents which are used to transmit data to/receive data from the DUT via serial/parallel interface. The jitter agent shown in Figure 5 injects jitter on the interface, based on the configuration parameters.

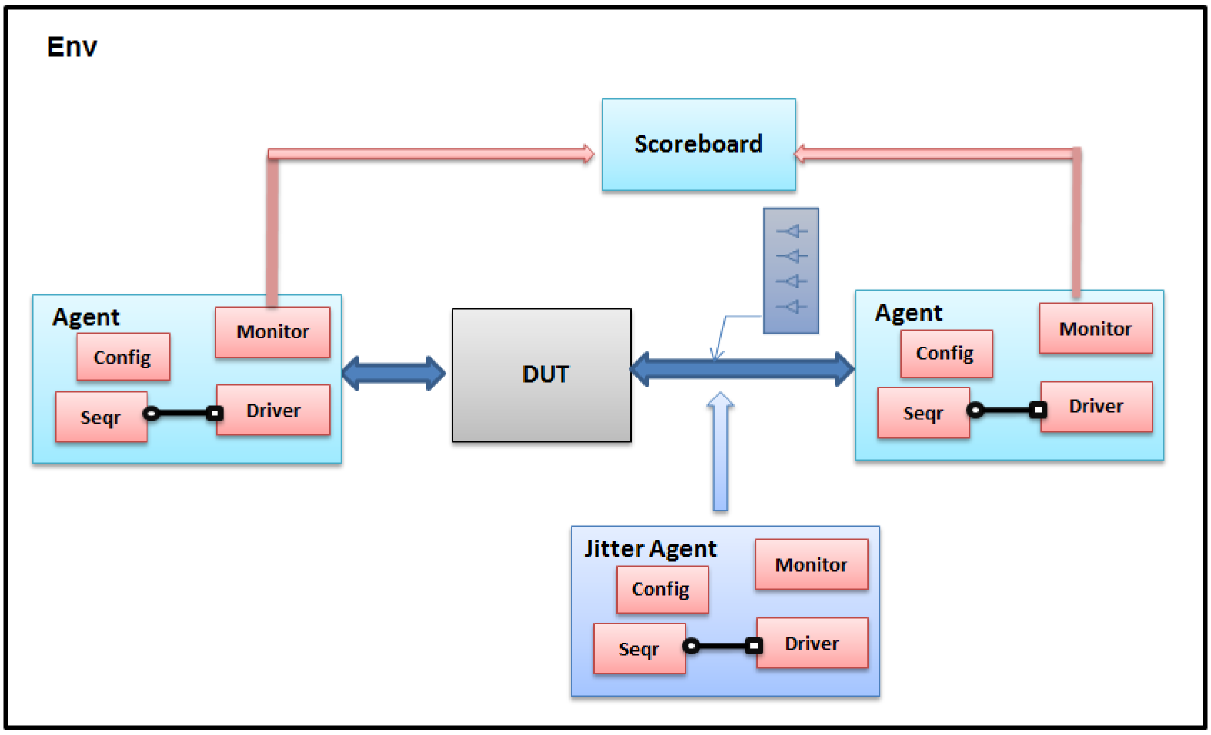
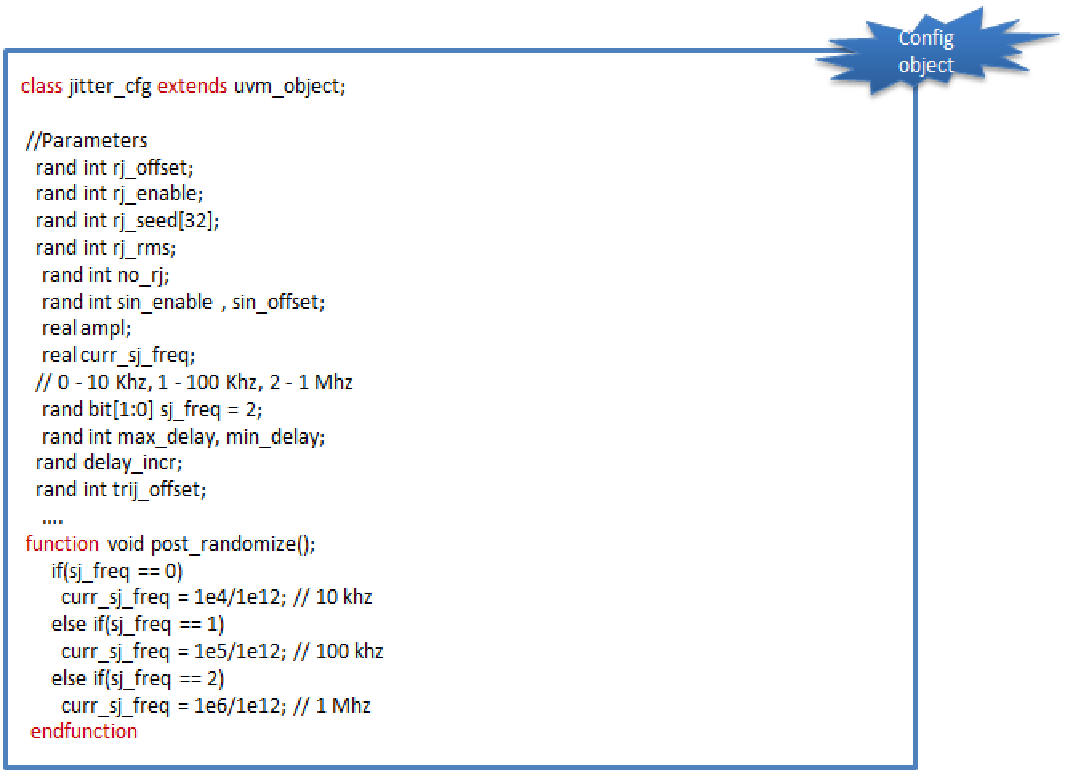


Figure 5: UVM testbench developed with the proposed approach.

VII. COMPONENTS OF JITTERAGENT– CONFIGURATION OBJECT

The jitter agent has a configuration object which containstestbench parameters. These parameters can be configured by the user based on design requirements. The type of jitter can be deterministic (sinusoidal or triangular), random (Gaussian), or a combination of both. For a sinusoidal jitter, the amplitude and frequency of the jitter sinusoid need to be considered. For jitter with a triangular profile, the minimum and maximum levels and the frequency of repetition need to be considered.Figure 6 below shows a sample configuration object.

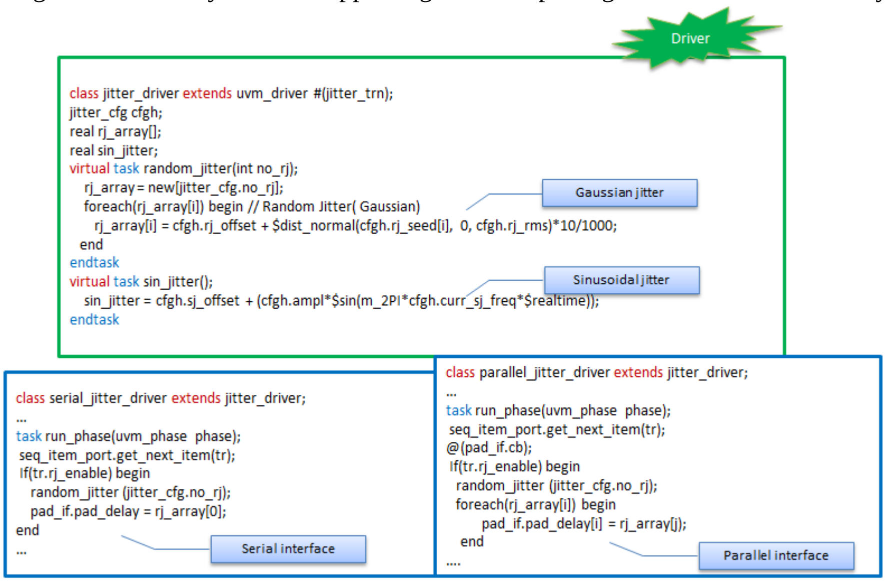


```
class jitter_cfg extends uvm_object;

 //Parameters
  rand int rj_offset;
  rand int rj_enable;
  rand int rj_seed[32];
  rand int rj_rms;
   rand int no_rj;
   rand int sin_enable , sin_offset;
   real ampl;
   real curr_sj_freq;
  // 0 - 10 Khz, 1 - 100 Khz, 2 - 1 Mhz
   rand bit[1:0] sj_freq = 2;
   rand int max_delay, min_delay;
  rand delay_incr;
  rand int trij_offset;
   ....
 function void post_randomize();
     if(sj_freq == 0)
       curr_sj_freq = 1e4/1e12; // 10 khz
     else if(sj_freq == 1)
       curr_sj_freq = 1e5/1e12; // 100 khz
     else if(sj_freq == 2)
       curr_sj_freq = 1e6/1e12; // 1 Mhz
  endfunction
```

Figure 6: Sample configurationobject.

VIII. COMPONENTS OF JITTER AGENT– DRIVER

The base class of the jitter driver computes the jitter based on the configuration object parameters. This base class driver is extended to create drivers specific to a serial or a parallel interface. For a serial or parallel interface, the delays present in the interface are assigned the jitter values computed in the driver. These delays cause the incoming data to be delayed before appearing at the output.Figure 7 illustrates this delay computation.



```
class jitter_driver extends uvm_driver #(jitter_trn);
jitter_cfg cfgh;
real rj_array[];
real sin_jitter;
virtual task random_jitter(int no_rj);
   rj_array = new[jitter_cfg.no_rj];
   foreach(rj_array[i]) begin // Random Jitter( Gaussian)
     rj_array[i] = cfgh.rj_offset + $dist_normal(cfgh.rj_seed[i],  0, cfgh.rj_rms)*10/1000;
  end
endtask
virtual task sin_jitter();
   sin_jitter = cfgh.sj_offset  + (cfgh.ampl*$sin(m_2PI*cfgh.curr_sj_freq*$realtime));
endtask
```

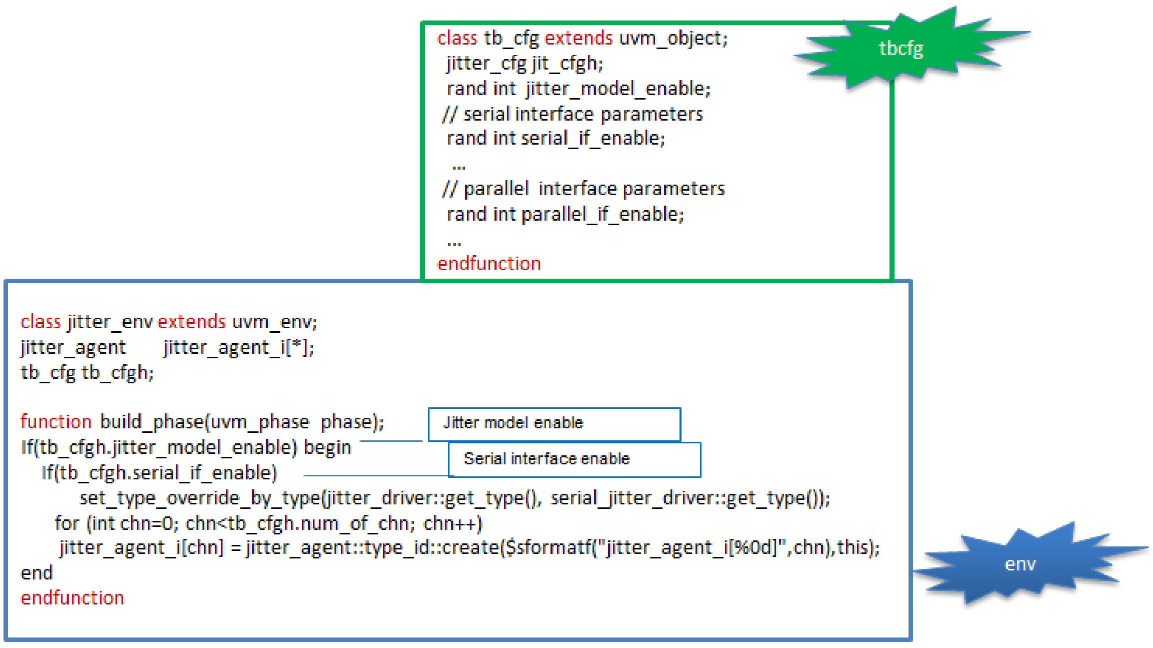
```
class serial_jitter_driver extends jitter_driver;
...
task run_phase(uvm_phase  phase);
 seq_item_port.get_next_item(tr);
 If(tr.rj_enable) begin
   random_jitter (jitter_cfg.no_rj);
   pad_if.pad_delay = rj_array[0];
end
...
```

```
class parallel_jitter_driver extends jitter_driver;
...
task run_phase(uvm_phase  phase);
 seq_item_port.get_next_item(tr);
@(pad_if.cb);
 If(tr.rj_enable) begin
  random_jitter (jitter_cfg.no_rj);
  foreach(rj_array[i]) begin
        pad_if.pad_delay[i] = rj_array[j];
   end
....
```

Figure 7: Example of a jitter driver.

IX. CREATION OF THE JITTER AGENT

The jitter agent consists of the configuration object, driver, sequencer, and monitor. This agent can be configured as active or passive. When the agent is configured as passive, only the monitor is instantiated. When the agent is configured as active, all of the driver, sequencer,and monitor are instantiated. However,the jitter agent is dynamic in nature and can be disabled when not needed. This can be programmed by the user through the configuration object. Hence, it doesn’t require any static allocation of memory. Figure 8 below shows how the creation of jitter agent can be dynamically controlled by the configuration object.



```
class tb_cfg extends uvm_object;
 jitter_cfg jit_cfgh;
 rand int  jitter_model_enable;
// serial interface parameters
 rand int serial_if_enable;
  ...
// parallel  interface parameters
 rand int parallel_if_enable;
 ...
endfunction
```

```
class jitter_env extends uvm_env;
jitter_agent      jitter_agent_i[*];
tb_cfg tb_cfgh;

function  build_phase(uvm_phase  phase);
If(tb_cfgh.jitter_model_enable) begin
   If(tb_cfgh.serial_if_enable)
        set_type_override_by_type(jitter_driver::get_type(),  serial_jitter_driver::get_type());
     for (int chn=0; chn<tb_cfgh.num_of_chn; chn++)
      jitter_agent_i[chn] = jitter_agent::type_id::create($sformatf("jitter_agent_i[%0d]",chn),this);
end
endfunction
```

Figure 8:Dynamic creation of the jitter agent.

X. REUSING THE JITTER MODEL FOR A NEW INTERFACE

In order to reuse this jitter model for any new interface, we need to follow these steps:a) use the Perl script to generate the interface file b)instantiate the jitter agent in theverification environment and c) configure the configuration object parameters in the test. This is illustrated in Figure 9.

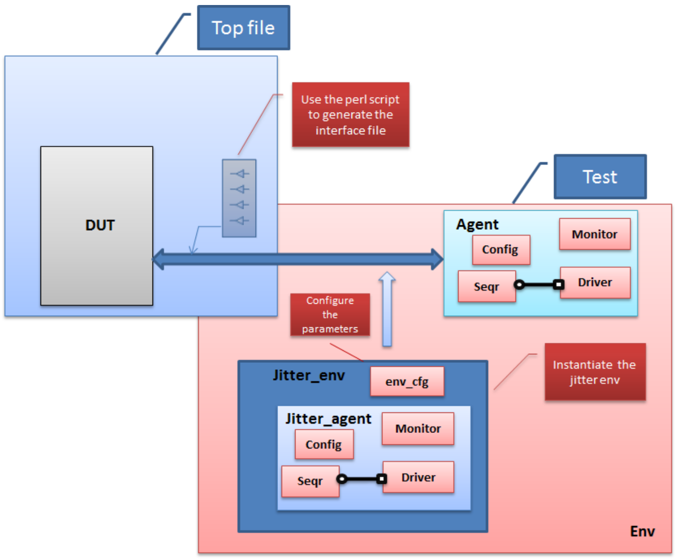


Figure 9:Reusing the jitter model for a new interface.

XI. CASE STUDIES AND RESULTS

In this section we discuss the results obtained with our proposed approach, both on serial and parallel interfaces.

A. WAVEFORMS ON SERIAL AND PARALLEL INTERFACES

In Figure 10, HSDP_P0 and HSDM_P0 refer to a differential pair of data lines on a serial interface. In Figure 10 the data bits received by the DUT don't have jitter. Hence, the width of the data bit is constant and is 2080 ps.

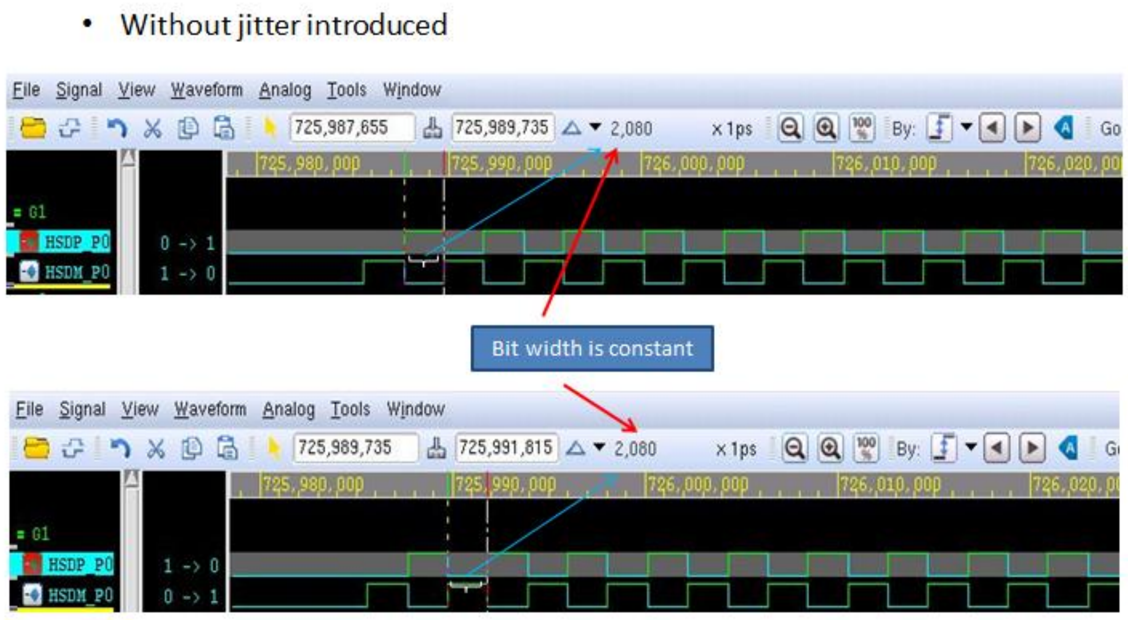


Figure 10:Data stream on a serial interface without jitter introduced.

In Figure 11, jitter has been introduced on the data stream received by the DUT on HSDP_P0 and HSDM_P0. The width of a bit varies because of this jitter. For the 2 bits marked in Figure 10, they have been measured as 2044 ps and 2097 ps.

- With jitter introduced

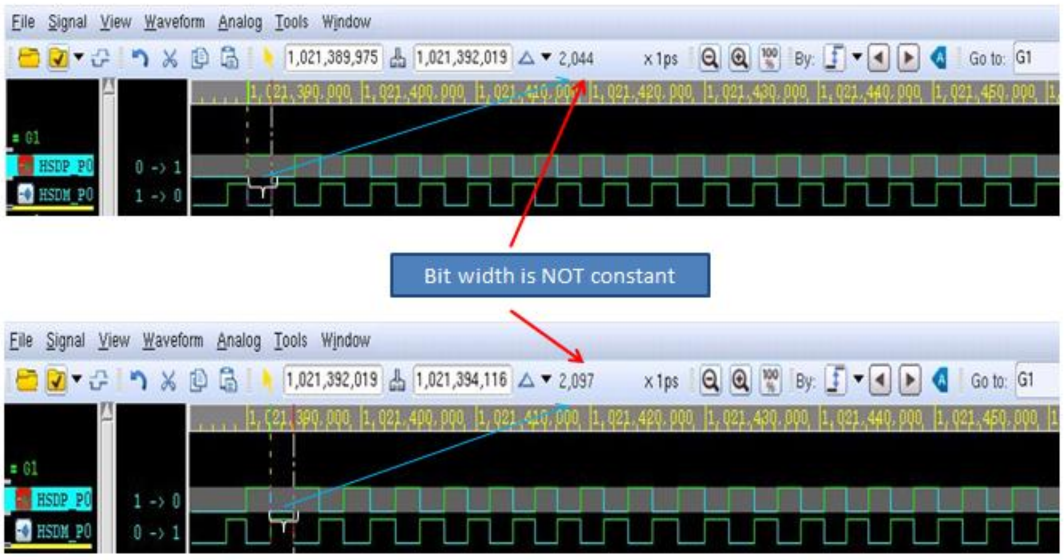


Figure 11: Data stream on a serial interface with jitter introduced.

Figure 12 depicts the introduction of jitter on a parallel interface on the write path. During a write operation, jitter has been introduced on the parallel data bus and the write data strobe.BP_DQ is a 32 bit bidirectional data bus and BP_WDQS_T is the write strobe without any jitter. BP_delay_DQ is the 32 bit data bus output, which has varying amounts of jitter/skew introduced on every bit of BP_DQ. BP_WDQS_delay_T is the write strobe obtained after introducing jitter on BP_WDQS_T. At the receiving end(agent/slave), the data bus BP_delay_DQ is sampled with respect to the write strobe BP_WDQS_delay_T. The varying amounts of jitter introduced is based on the jitter profile configured in the configuration object, and the monitor ensures that the jitter is within the given jitter margin.

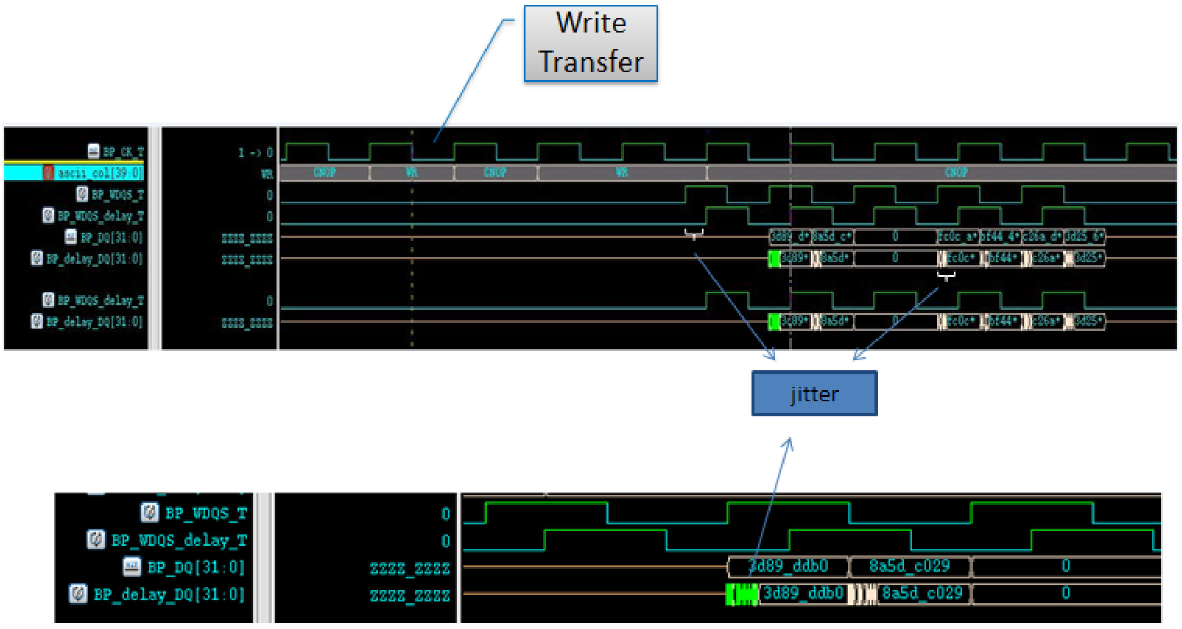


Figure 12: Jitter on a parallel interface on the write path.

Figure 13 depicts the introduction of jitter on the read path on the same interface. During a read operation, jitter has been introduced on the parallel data bus and the read data strobe. BP_SLV_BQ is a 32 bit read data bus and BP_RDQS_SLV_T is the read strobe without any jitter coming from the slave/agent. BP_DQ is the 32 bit data bus output which has varying amounts of jitter/skew introduced on every bit of read data bus BP_SLV_DQ. BP_RDQS_delay_T is the read strobe obtained after introducing jitter on BP_RDQS_SLV_T. At the receiving end (DUT), the read data bus BP_DQ is sampled with respect to the read strobe BP_RDQS_delay_T. The jitter added is based on the parameters in the configuration object.

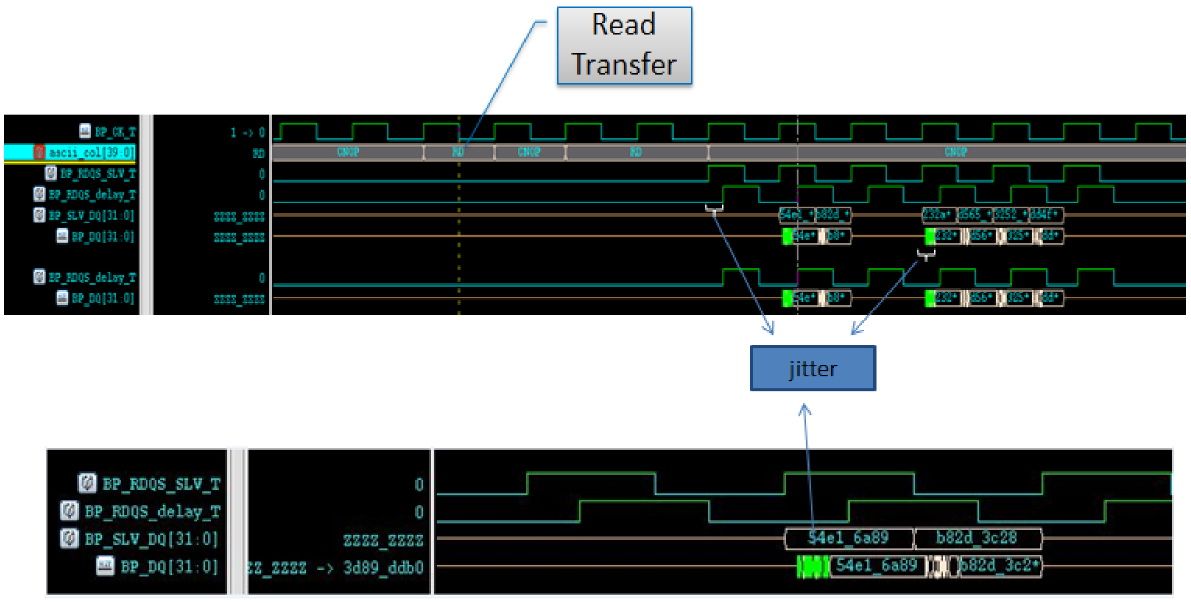


Figure 13: Jitter on a parallel interface on the read path.

*B.*BUGS ON A SERIAL INTERFACE

Table 1 captures the bugs found on a serial interface with our proposed approach.

Table 1: Bugs.

Category	**Description of the bug**
Received data corrupted	Final bit of a HS sync pattern is received in the first bit of a UI where 2 bits are processed. If this happens, the 2nd bit doesn't get written into the elastic buffer, but is discarded.
False assertion of receive error	Noise at the end of a HS chirp response aliases to a start-of-packet pattern. This falsely asserts receive error to the controller.

Figure 14depicts the first bug mentioned in the table, which is received data corruption on a serial interface. In this figure,dp/dm refer to a pair of differential serial data lines. rx_data is a 8-bit parallel data obtained after the clock-data recovery of the serial data. At the position indicated by the marker, the data bit doesn't get written into the elastic buffer, but is discarded. This results in thefirst byte obtained on rx_data being incorrect.

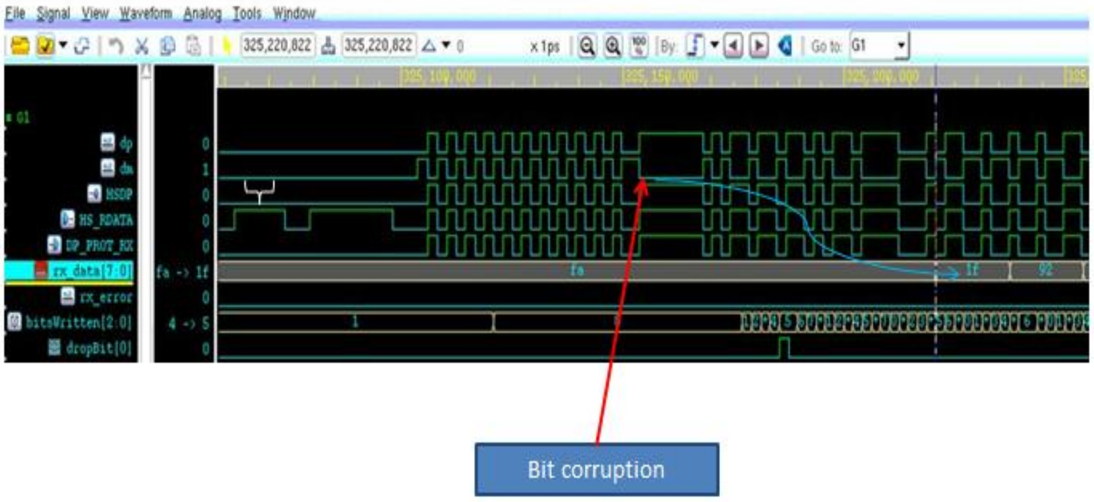


Figure 14: Received data corruption on a serial interface.

XII. CONCLUSION AND FUTURE WORK

A separate jitter model for each new interface lacks reusability. With our proposed approach, any new interface can be generated and the jitter model can be quickly integrated in the verification environment. Due to the configurability, the model can be reused across any serial or parallel interface. The model is dynamic in nature and can be disabled when not needed. This approach saved us 2 weeks worth of effort when developing a jitter model for a new interface.
In the future, we want to incorporate modeling clock jitter as a part of this solution. This will enable us to generate jitter on the clock in a controlled fashion. We also want to develop the solution as a Verification IP (VIP).

ACKNOWLEDGMENT

We would like to acknowledge our colleaguesToni Simov, Atanaska Yanachkova, and Govinda Raju at AMD for their valuable suggestions during this effort.

REFERENCES

[1] Pavan Kumar Hanumolu, Bryan Casper, Randy Mooney, Gu-Yeon Wei, and Un-Ku Moon, "Jitter in high-speed serial and parallel links,"ISCAS 2004, pp. IV-425-428, 2004.

[2] Nelson Ou, Touraj Farahmand, Andy Kuo, Sassan Tabatabaei, and André Ivanov,"Jitter Models for the Design and Test of Gbps-Speed Serial Interconnects," IEEE Design & Test of Computers, pp 302-313, July-August 2004.

[3] UVM Cookbook. [Online]. Available:www.verificationacademy.com.Retrieved September, 2014.

[4] Cadence,Mentor Graphics, Synopsys, "Accellera : Universal Verification Methodology (UVM) 1.1 User's Guide," May 18,2011.

ATTRIBUTION