Functional Verification of Analog Devices modeled using SV-RNM

Mariam Maurice, Siemens DISW, mariam.maurice@siemens.com

***Abstract*— New functional verification models are introduced from the random stimulus, functional coverage, assertions, and UVM on the Analog to Digital Converter (ADC) and Digital to Analog Converter devices, to ensure that the modeled ADC/DAC devices are designed correctly according to the system specifications and requirements.**

I. Introduction

As waiting for the completion of the analog transistor level could extend the time to market for the digital verification engineers to ensure that both the analog and the digital systems will function properly when they are connected, the functional verification of the analog devices that are modeled using the Real Number Modeling (RNM) has become a crucial step of the mixed-signal SoC's validation. The use of the RNM has four benefits.

- The first one is that the Real Number Models are faster than SPICE and VERILOG-AMS electrical models.
- The second advantage is that the Real Number Models are more accurate than the purely digital model and there are a lot of modeling techniques that provide a higher accuracy, and the output could be much equivalent to the output from the transistor netlist [1], [2].
- The third advantage, the RNMs are Top-Down models where the verification starts without finishing the schematic. Furthermore, because they are reuse models, the only significant changes to the same model analog block are in its parameters.
- The fourth advantage is in the top-level System-on-Chip (SoC) verification where the engineers can represent all the electrical signals as RNM equivalents and stay within the digital simulation environment [3],[4].

The third and fourth advantages will be the main emphasis of the paper study, which will build on the work in [1], [2] that addresses the first and second advantages. The precision of the analog simulated signals in the digital environment is also increased by new techniques for modeling the analog devices, as explained in [1]. Therefore, the functional verification could start without finishing the analog schematics. There is no need to wait until finishing the analog devices transistor level, to verify the system design.

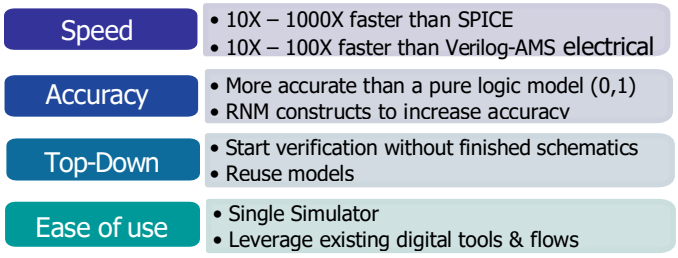


Figure 1. Pros of RNM

The paper contributes with new functional verification models from random stimulus, functional coverage, assertions, and UVM on the Analog-Digital Converter (ADC)/Digital-Analog Converter (DAC) to ensure the functionality of the modeled ADC/DAC. Moreover, there are methods presented in each verification model that increase the verification accuracy according to the system specification and requirements. Therefore, the digital verification engineer will have precious RNM models for the analog devices and variety of the functional verification methods on these RNM models. All of these will be helpful when the mixed devices, the analog and digital devices, are get connected on the transistor level as the high accuracy of modeling and functional verification in the digital environment will reduce the existence of bugs regarding the functionality of both devices' connection together and the data flow through the boundaries of mixed devices connectivity.

II. FUNCTIONAL VERIFICATION

A. *Randomization*

a. Randomization as modeling mismatch effects

Devices like the Flash_analog_to_digital converter (ADC) can have a random mismatch in the resistive divider block. This will result that the voltage at each node of the resistive divider will change. The change can be modeled using the randomization of the reference voltages. Or any device that divides the reference voltage or current through a string of components. Writing constraints can be set, in accordance with the limits of mismatches within the specified block which may be expected from the System-Level engineers to avoid exceeding these limits. Then analyze the outputs to ensure that the block functionality is working correctly, and the driven output values are those expected. Another one might have thought to write the constraints without any limits because the verification engineer was unable to determine the maximum amount of mismatch in the randomized signal. However, after debugging, the verification engineer was able to determine the maximum amount of the mismatch needed to maintain the proper output values. As an additional higher order of thought, the verification engineer may be able to determine the limit of mismatching in accordance with which the constraints are written; however, on occasion, the verification engineer may also harden the limit of mismatch to illustrate the potential consequences for the system and the extent to which it may be functionally flawed.

Figure 2(a). illustrates a resistive divider that could have a voltage mismatch at each node due to its component mismatch. The constraint is written such that the input analog reference voltage will have the quantization noise added or subtracted from it as if the signal exceeds the quantization noise, the output of the ADC will give an error code as illustrated in Figure 2(b). The worst quantization noise is equal to $(\frac{\Delta}{2})$ where (Δ) is equal to $(\frac{V_{ref}}{2^n})$ and n is the number of levels of conversion [5].

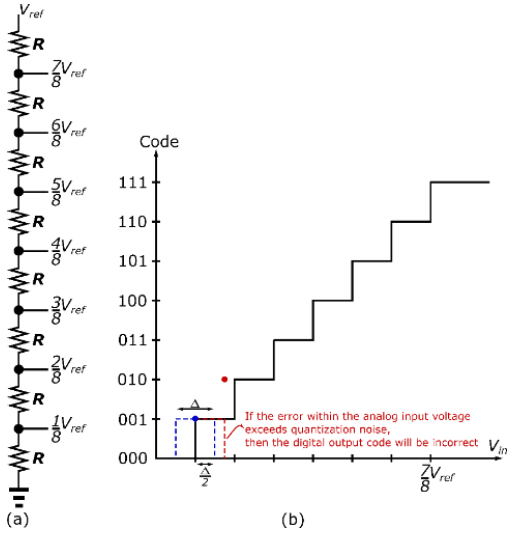


Figure 2. (a) The voltage node of a resistive divider **(b)** The transfer function of ADC

```
class rand_vref #(
      real AVDD = 2.5, // High Supply
      int n = 3,       // Resolution (Accuracy)
      int n_levels = $pow(2,n), // Number levels of conversion
      real delta = AVDD / n_levels;

      rand real VREF;
      constraint c_VREF {VREF inside {[AVDD - (delta/2) : AVDD + (delta/2)]};};
endclass: rand_vref
```

Listing 1. SV class provides randomization constraints on reference voltage

Moreover, the randomization can model the voltage variation in the supply high voltage (AVDD) and the supply low voltage (AGND). The variations in the supply voltage are modelled to make the chips work after fabrication in all the possible conditions as there are a lot of reasons that could make the supply voltage to variate.

```
class rand_supply #(
       real P_TOL = 0.1, // Positive Tolerance
       real N_TOL = 0.1, // Negative Tolerance
       real T_AVDD = 1.0, // Typical AVDD
       real T_AGND = 0.0); // Typical AGND

       rand real AVDD;
       rand real AGND;

       constraint c_AVDD {AVDD inside {[ (T_AVDD-N_TOL) : (T_AVDD+P_TOL) ]}; };
       constraint c_AGND {AGND dist {T_AGND:/10, [P_TOL/100:P_TOL]:/100};};
endclass: rand_supply
```

Listing 2. SV class provides randomization constraints on supply voltages tolerance

a. Randomization of Input Real/logic signals

The input real signal can have a form of DC value, sine/cosine wave, triangular, sawtooth, or noisy signal. All the previous signals can be modeled as their functionality. In verification, it's preferred to generate the real input values as a range of the randomized real data and track the output to ensure the functionality of the system is observed. The input real voltage signal will act as an input to the ADC.

```
rand real vin;

constraint c_vin {vin inside {[-2.5:-0.1] , [0.1:2.5]}; } // Avoiding pure "zero"
constraint c_vin {vin dist {0.0:/5 , [0.1:12]:/50, [-12:-0.1]:/50};}
                                                    //Without Avoiding pure "zero"
```

Listing 3. Randomization constraints on input real voltage

The input logic signal can be randomized between any value of the start digital code (0) to the end digital code ($2^n - 1$). Or can be randomized sequentially by starting from the first digital code (0) and increasing by the next digital code until reach the last digital code. ($0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \ldots. \rightarrow 2^n - 1$). The input digital code will act as an input to the DAC.

```
parameter n = 3;
rand logic [n-1:0] q;

constraint c_q {q inside {[0:((2**n)-1)]}; }
                      // randomize the digital code within a range
int count=0;
constraint q_c {
    if (count <= ((2**n)-1)) {
        q == count ;
    } else {
        q == 0;
    }
 }                  // randomize the digital code with a defined step

 function void post_randomize;
   $display("count = %0d",count);
   count++;
 endfunction
```

Listing 4. Randomization constraints on input digital code

B. Functional Coverage

Cover an interval of the electrical signal from its low to high amplitude value. The low amplitude could be zero, -ve real analog value, or the low supply value as the ADC input (or the randomized input to ADC), or any modeled electrical signal that its value covers a range of real amplitude datatype. The automatic number of the bins is defined according to the 'real_interval' that passed as a 'type_option' and divided according to the number of real values. Which means, once a bin is created for a value then not another bin is created for this value even if the value is repeated within a different range or declared as a separate value.

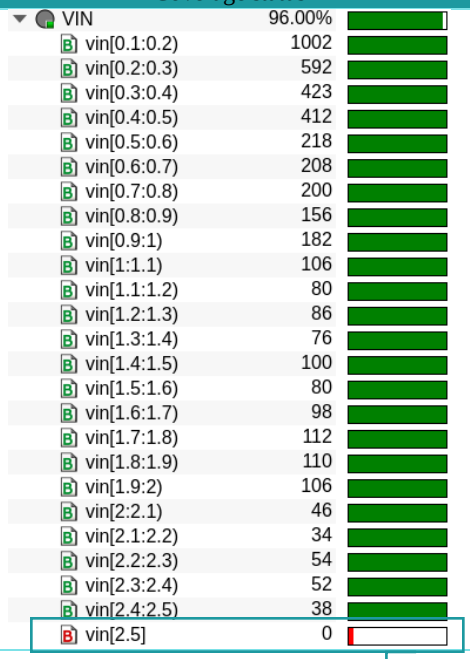
```
// COVERAGE VOLTAGE VALUES WITHIN A CERTAIN RANGE
real min_a = 0.1; // minimum amplitude
real max_a = 2.5; // maximum amplitude

covergroup V_COV @(vout_load);
        option.per_instance = 1;
        coverpoint vout_load {
                type_option.real_interval = 1E-1;
                bins v [] = {[min_a:max_a], min_a, max_a};
        }
endgroup: V_COV
```

Listing 5. Cover an electrical signal

Table 1. Bins creation

type_option.real_interval = x		
bins	values	Coverage status
[min_a : min_a + x]	covers values between min_a to (min_a + x) and the specific value min_a if it's existed	
[min_a + x : min_a + 2x]	covers values between min_a + x to min_a + 2x and the specific value min_a + x if it's existed	
[min_a + 2x : min_a + 3x]	covers values between min_a + 2x to min_a + 3x and the specific value min_a + 2x if it's existed	
…	…	
[max_a – 2x : max_a – x]	covers values between max_a – 2x to max_a – x and the specific value max_a – 2x if it's existed	
[max_a – x : max_a]	covers values between max_a – x to max_a and the specific value max_a – x if it's existed	
[max_a]	covers specific value max_a if it's existed	



From Figure 3, the maximum value of the signal 'vin' is 2.49782. For that the bin 'vin[2.5]' is not hit.

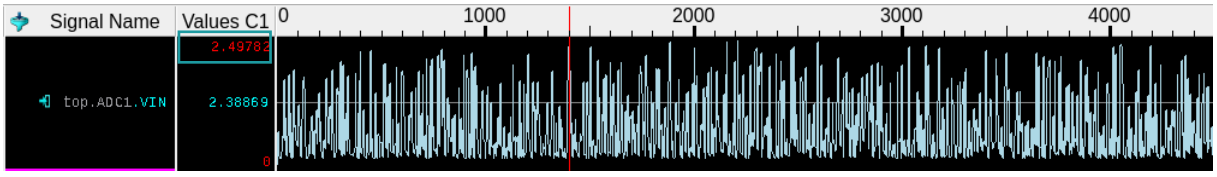


Figure 3. VIN values

C. *Assertions/Checkers*

a. *Assertion on a relation between I/O ports*

Asserting a relation between the output and input signal, helps verifying the functionality of the whole system. For example, the ADC converts an analog input signal to a digital-level code corresponding to the ADC resolution (Accuracy). The accuracy of the conversion increases as the number of conversion levels increases. This can be implemented by declaring a delta variable which is the difference between two levels of conversion. As the number of levels increases, the smaller will be the delta and the more accuracy will be. The number of levels is defined by an (n)-parameter where the number of levels is equal to (2**n) and the delta is equal to the high supply voltage (vsup) divided by the number of levels (2**n). Therefore, the relation between the analog input and digital output could be simply expressed as the following equation [6].

$$Logic\ (output) = \left\lfloor \frac{Real\ (input)}{delta} \right\rfloor \text{ Where, } delta = \frac{v_{sup}}{2^n}. \quad (1)$$

```
// PARAMETERS
        parameter n = 3;  //
        parameter real nlevels = $pow(2,n); // number levels of conversion

// VARIABLES
        real vsuplow = 0; // low supply voltage
        real delta;       // step of converter
        reg [n-1:0] q;    // output code of ADC

// ASSIGNATION
  always @(VSUP)
        delta = VSUP / nlevels; // vsup is supply voltage or full scale voltage
                                // vsup is considered as an input to converter

  always @(VIN) begin
    if (VIN >= vsuplow && VIN < delta) q = '0;
    else if (VIN >= ((nlevels-1)*delta) && VIN <= VSUP)  q = '1;
    else if (VIN >= delta && VIN < ((nlevels-1)*delta))  q = $floor(VIN / delta);
  end

// ASSERTION
    ADC: assert property (@(VIN) ((VIN >= vsuplow) && (VIN <= VSUP)) |-> Q == q);
```

Listing 6. SV code for verifying the ADC functionality

The above SV code can have a little bit of enhancement to model the mismatch within (vsup) using the constraint randomization. Moreover, most ADCs have a sample and hold circuit between the input and the ADC sub-block. Therefore, the assertion property will depend on the clock signal.

```
// VARIABLES
        real clk_period = 2;

// ASSIGNATION
    always @(VIN) begin
        if (VIN >= vsuplow && VIN < delta) #(clk_period) q = '0;
        else if (VIN >= ((nlevels-1)*delta) && VIN <= VSUP) #(clk_period) q = '1;
        else if (VIN >= delta && VIN < ((nlevels-1)*delta)) #(clk_period)
                                                    q = $floor(VIN / delta);
    end

   ADC: assert property (@(posedge CLK) ((VIN >= vsuplow) && (VIN <= VSUP))
                                                          |-> Q == q);
```

Listing 7. Modified SV code for verifying the ADC functionality

The output logic signal Q[2:0] is modeled according to a specific ADC topology which is the FLASH_ADC as illustrated in Figure 4. While q[2:0] is a generic expression of the ADC functionality between the output and input signals. Therefore, the assertion (ADC_ASSERT) is verifying that the FLASH_ADC is passed (P) when the expected functionality of ADC is observed.

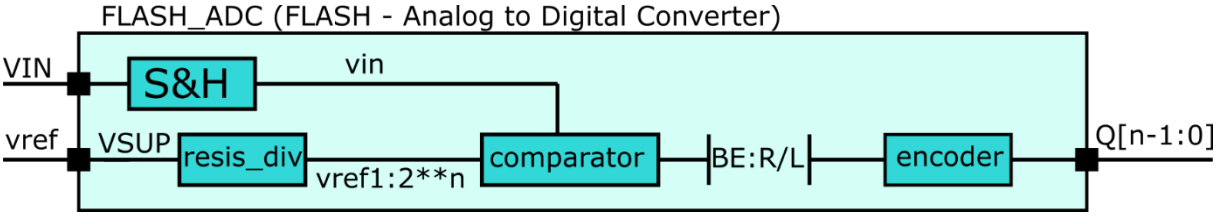


Figure 4. FLASH ADC

Figure 5. illustrates the randomization in the VIN (input real voltage to ADC) and VSUP (input real supply voltage to ADC). The delta value is changed according to the value of VSUP. The assertion (ADC_ASSERT) verifies that the FLASH_ADC is passed (P) when the expected functionality of ADC is observed.

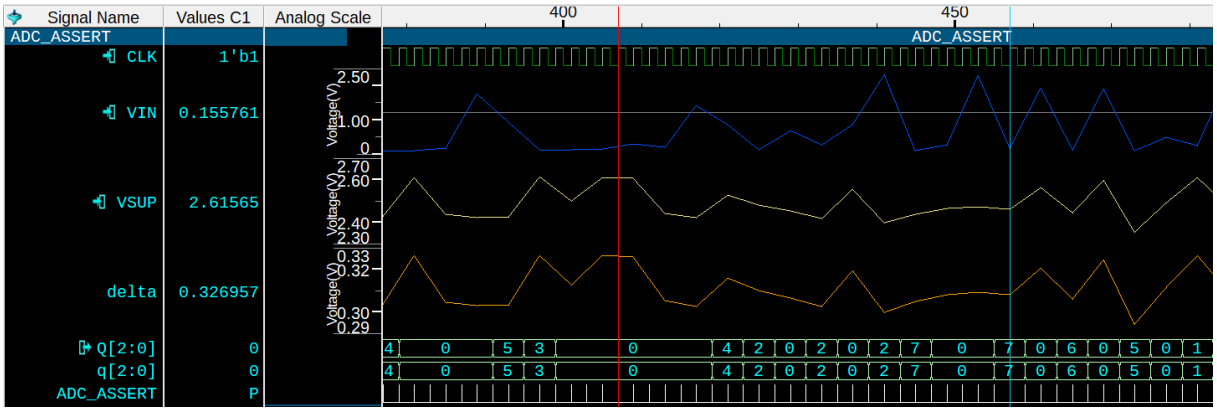


Figure 5. The assertion of ADC

Notice:

- The 'VIN' and 'VSUP' signals, that are mentioned in Listing 6,7, and Figure 5, are declared in listing 1 & 3 respectively.
- The 'vref' and 'VSUP' signals, are the same. Both signals mean the supply voltage or full-scale voltage.
- The 'Q[2:0]', is the digital output code that generated from the DUT itself. The DUT (FLASH_ADC) is mentioned in figure 4. or in [2].
- The 'q[2:0]', is the digital output code that generated from Listing 6. to asset on the functionality correctness of the DUT (FLASH_ADC).

Similarly, the DAC converts the digital input level code to an analog output value corresponding to the DAC resolution (Accuracy). The relation between the digital input and analog output could be simply expressed as the following equation.

$$Real\ (output) = Logic\ (input) \times delta \tag{2}$$

The output analog signal (VOUT_A) is modeled according to a specific DAC topology which is R_STRING_DAC as illustrated in Figure 6. While vout_a, the generated output from equation (2), is a generic expression of the DAC functionality between the output and input signals.

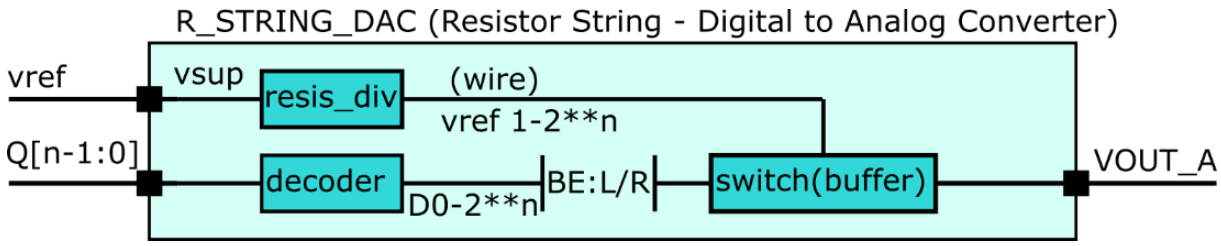


Figure 6. R-String DAC

```
// PARAMETERS
        parameter n = 3;
        parameter real nlevels = $pow(2,n);

// VARIABLES
        real delta;
        real vout_a;

//ASSIGNATION
       always @(VSUP)
         delta = VSUP / nlevels;

        always @(Q) begin
                if (Q > '0 && Q < '1) vout_a = Q * delta;
                else if (Q == '1)  vout_a = ((nlevels-1)*delta);
                else if (Q == '0)  vout_a = 0;
        end

   DAC: assert property (@(Q) ((Q >= '0) && (Q <= '1)) |-> VOUT_A == vout_a);
```

Listing 8. SV code for verifying the DAC functionality

Notice:

- The ‘Q’ signal, that is mentioned in Listing 8., is declared in listing 4.
- The ‘vref’ and ‘VSUP’ signals, are the same. Both signals mean the supply voltage or full-scale voltage.
- The ‘VOUT_A’, is the analog output voltage that generated from the DUT itself. The DUT (R_STRING_DAC) is mentioned in figure 6. or in [2].
- The ‘vout_a’, is the analog output voltage that generated from Listing 8. to asset on the functionality correctness of the DUT (R_STRING_DAC).

D. UVM

The UVM is the most widely used verification standard for the modern digital circuits. However, it has been heavily utilized in the mixed-signal applications lately. The digital scenario uses the UVM elements including the constrained-random stimulus generation, verification planning, assertions, and coverage metrics production. The verification of the analog part of the mixed-signal systems, is usually achieved by hard approaches such as the directed testing, corner analysis, and Monte Carlo simulations.

As a result, integrating the UVM and Real Number Modeling (RNM) is a crucial tactic for creating a fast and reliable verification environment for the mixed-signal devices. Just one environment enables the verification of the analog devices from the Constrained random Verification (sequence item through the virtual interface of the input/output ports), modeling the internal mismatches which in part ‘II.A.a’ (sequence items through the virtual interface of the internal signals ‘bind’), Assertions/Checkers (scoreboards), and Functional Coverage (subscribers).

In this work, a full analog-mixed ADC/DAC modeled DUT is verified using the UVM. There are two agents, two subscribers, and a scoreboard in the UVM environment. The sequence items are received by the sequencer in

the active agent, which forwards them to the driver. The interface then delivers the sequence items to the DUT. There is a monitor component in the passive agent. Using the virtual interface (Pin Wiggles), the monitor records the DUT signals and transforms them into sequence items (transactions) that are sent to the subscriber and scoreboard as illustrated in Figure 7.

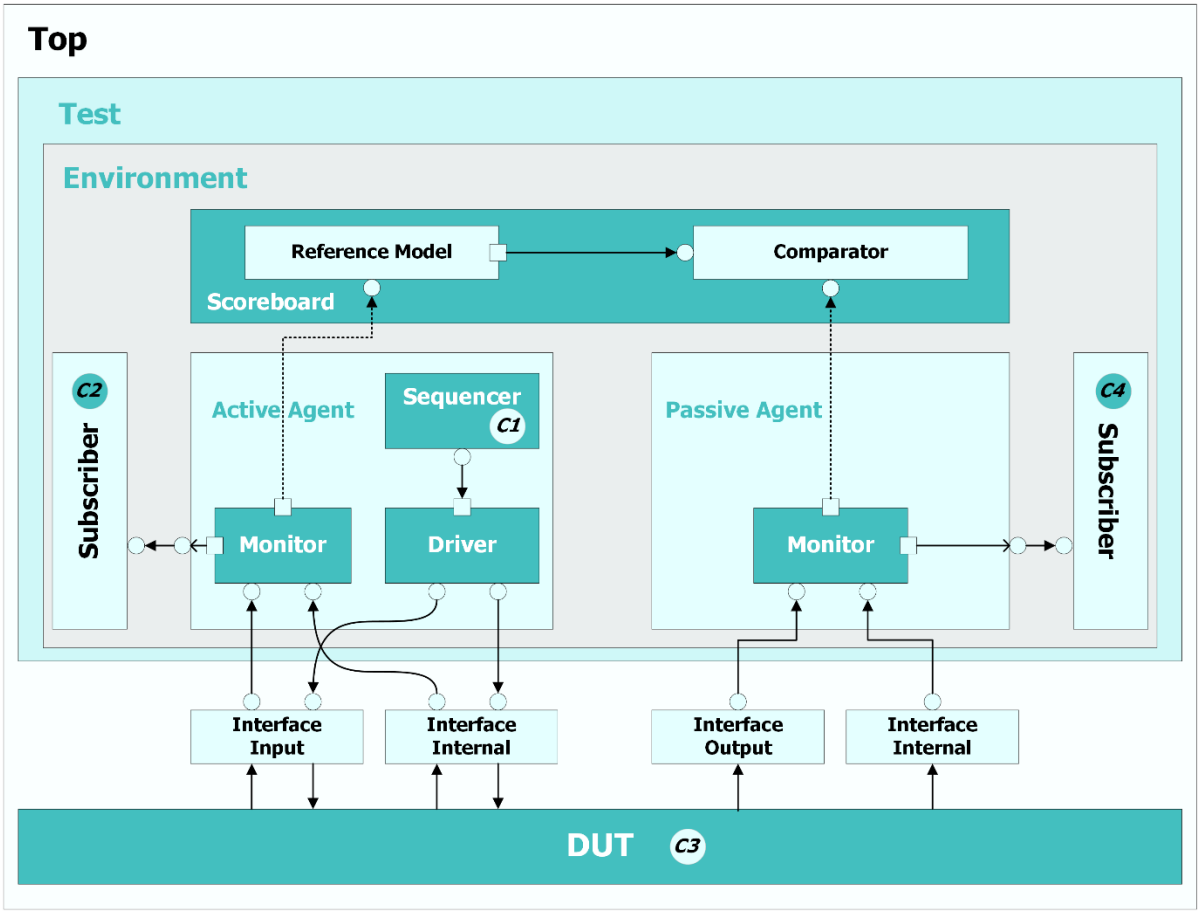


Figure 7. UVM testbench environment

a. Interfaces

- Interface – Input ports: Interface for accessing the DUT's input ports

ADC Interface_In	DAC Interface_In

`interface ADC_IF_IN (input CLK);`
`  real VIN;`
`  real VREF;`
`endinterface: ADC_IF_IN`
	`interface DAC_IF_IN (input CLK);`
	`  parameter int n = 3;`
	`  logic [n-1 : 0] Q;`
	`endinterface: DAC_IF_IN`

Listing 9. Interface for Accessing DUT's Input Ports

- Interface – Output ports: Interface for recording the DUT's output ports

ADC Interface_Out	DAC Interface_Out

`interface ADC_IF_OUT (input CLK);`
`  logic [2:0] Q;`
`endinterface: ADC_IF_OUT`
	`interface DAC_IF_IN (input CLK);`
	`  real VOUT_A;`
	`endinterface: DAC_IF_IN`

Listing 10. Interface for Recording DUT's Output Ports

- Interface – Internal signals: Interface for accessing or recording the internal signals of the DUT by binding it to the DUT. This is helpful to force an internal signal by a value or by the randomized value within a constraint random range.

ADC Interface_Internal

```
import volt_pkg::*; // pkg has a UDN
interface ADC_IF_INT (
          input logic CLK,
          input real vref1,
// To access resistive
// reference voltages
          input real vref2,
          input real vref3,
          input real vref4,
          input real vref5,
          input real vref6,
          input real vref7,
          output volt D1,
// UDN holds resolved voltage
// To record comparator output
          output volt D2,
          output volt D3,
          output volt D4,
          output volt D5,
          output volt D6,
          output volt D7
          );
endinterface
```

Binding Declaration of DUT to Internal Interface

```
module top ();
…
// Binding DUT to internal interface
bind FLASH_ADC ADC_IF_INT int_if
(.CLK(CLK),
 .vref1(FLASH_ADC.RES_DIV.vref1),
 .vref2(FLASH_ADC.RES_DIV.vref2),
 .vref3(FLASH_ADC.RES_DIV.vref3),
 .vref4(FLASH_ADC.RES_DIV.vref4),
 .vref5(FLASH_ADC.RES_DIV.vref5),
 .vref6(FLASH_ADC.RES_DIV.vref6),
 .vref7(FLASH_ADC.RES_DIV.vref7),
 .D1(FLASH_ADC.COMP.D1),
 .D2(FLASH_ADC.COMP.D2),
 .D3(FLASH_ADC.COMP.D3),
 .D4(FLASH_ADC.COMP.D4),
 .D5(FLASH_ADC.COMP.D5),
 .D6(FLASH_ADC.COMP.D6),
 .D7(FLASH_ADC.COMP.D7));

always @(CLK) begin
        force FLASH_ADC.RES_DIV.vref1 =
FLASH_ADC.int_if.vref1;
        force FLASH_ADC.RES_DIV.vref2 =
FLASH_ADC.int_if.vref2;
        force FLASH_ADC.RES_DIV.vref3 =
FLASH_ADC.int_if.vref3;
        force FLASH_ADC.RES_DIV.vref4 =
FLASH_ADC.int_if.vref4;
        force FLASH_ADC.RES_DIV.vref5 =
FLASH_ADC.int_if.vref5;
        force FLASH_ADC.RES_DIV.vref6 =
FLASH_ADC.int_if.vref6;
        force FLASH_ADC.RES_DIV.vref7 =
FLASH_ADC.int_if.vref7;
 end


// base pointer "virtual interface" in
configuration database with a field name :
"adc_vi_int"
        uvm_config_db #(virtual
ADC_IF_INT)::set(null, "uvm_test_top",
"adc_vi_int", FLASH_ADC.int_if);
…
endmodule
```

Listing 11. Interface for access/record internal signal and bind declaration of DUT to internal interface

b. Sequence generation

The sequence-item, sequence, sequencer, and driver are the four classes needed for the entire sequence generation process. The sequences are reusable, and the environment testbench build has no bearing on the stimulus generation. The data fields needed to generate the stimulus make up the sequence-item (transaction). It is necessary to randomize the sequence-item's variables in order to generate the stimulus. As a result, the sequence-item's data properties are declared as rand and subject to randomization constraints.

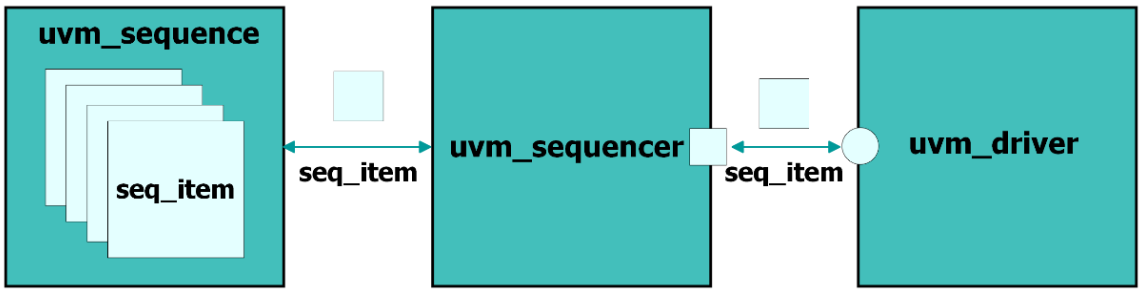


Figure 8. Sequence Items Generation

For the ADC device the reference voltage (VREF) constraint (c_VREF) will have the maximum quantization noise added or subtracted from it as if the signal exceeds the quantization noise, the output of the ADC will give an error code. The input voltage (VIN) to the ADC that will be converted to a digital code, which will be a randomized input from low voltage (VL) to high voltage (VH).

ADC sequence_item

```
class     adc_transaction_in     extends
uvm_sequence_item;
      …
// Randomization of Input Signals
   rand real VREF;

   parameter real A_VDD = 2.5;
         // Ideal High Supply
   parameter int n = 3;
         // Resolution (Accuracy)
   int n_levels = $pow(2,n);
         // Number levels of conversion
   real delta = (A_VDD / n_levels);

   constraint c_VREF {VREF inside
         {[A_VDD - (delta/2) :
           A_VDD + (delta/2)]};};

   rand real VIN;

   real VL = 0.01;
   real VH = 2.5;

   constraint c_VIN
{V_in dist {0.0:/5 ,   [VL:VH]:/50,   [-
VH:-VL]:/50};}
      …
endclass: adc_transaction_in
```

DAC sequence_item

```
class dac_transaction_in extends
uvm_sequence_item;
      …
      parameter int n = 3;
         // Resolution (Accuracy)
      int n_levels = $pow(2,n);
         // Number levels of conversion

      rand bit [n-1:0] Q;


      constraint c_Q {Q inside
                {[0 :(n_levels-1)]};};
      …
endclass: dac_transaction_in
```

Listing 12. SV code for uvm_sequence_item of ADC/DAC (randomize of input ports)

The ‘uvm_sequence_item’ class has also the randomization of the internal signals that model the random mismatch within the ADC components.

ADC sequence_item (randomization of internal signals)

```
class adc_transaction_in extends uvm_sequence_item;
      …
// Randomization of Internal Signals
    rand real vref1;
    constraint c_vref1 { vref1 inside {[(1*VREF)/n_levels - (delta/2) :
```

```
(1*VREF)/n_levels + (delta/2)]}; };

    rand real vref2;
    constraint c_vref2 { vref2 inside {[(2*VREF)/n_levels - (delta/2) :
(2*VREF)/n_levels + (delta/2)]}; };

    rand real vref3;
    constraint c_vref3 { vref3 inside {[(3*VREF)/n_levels - (delta/2) :
(3*VREF)/n_levels + (delta/2)]}; };

    rand real vref4;
    constraint c_vref4 { vref4 inside {[(4*VREF)/n_levels - (delta/2) :
(4*VREF)/n_levels + (delta/2)]}; };

    rand real vref5;
    constraint c_vref5 { vref5 inside {[(5*VREF)/n_levels - (delta/2) :
(5*VREF)/n_levels + (delta/2)]}; };

    rand real vref6;
    constraint c_vref6 { vref6 inside {[(6*VREF)/n_levels - (delta/2) :
(6*VREF)/n_levels + (delta/2)]}; };

    rand real vref7;
    constraint c_vref7 { vref7 inside {[(7*VREF)/n_levels - (delta/2) :
(7*VREF)/n_levels + (delta/2)]}; };
…
endclass
```

Listing 13. SV code for uvm_sequence_item of ADC (randomize of internal signals)

c. Monitors

The environment here has two UVM monitors:

- The first one captures the input signals to the DUT through the virtual interface responsible for accessing the DUT's input ports. Moreover, it captures the internal signals of the DUT through the virtual interface responsible for accessing the DUT's internal signals that the verification engineer needs to force their values instead of defined values to these signals. This monitor is inside an active agent.
- The second one captures the output signals to the DUT through the virtual interface responsible for recording the DUT's output ports. Moreover, it captures the internal signals of the DUT through the virtual interface responsible for recording the DUT's internal signals that the verification engineer needs to ensure their values are within the expected values range. This monitor is inside a passive agent.

d. Coverage

Since it is preferred to have functional cover points in four places for any verification environment.

- The first one is placed close to the randomization to guarantee that the system is set to all expected randomized values within a given range. As a result, the sequence declares the cover points. *(C1)*
- The second one is located close to the system's input to make sure that all expected values are present at the DUT's input ports. Since the entire system will be using these values, all expected test values will be passed through the design. The uvm_subcriber, which was designed to handle data that are tracked from the input interface, can accomplish this. *(C2)*
- The third one is located within the DUT to guarantee that the internal signals contain the right values. *(C3)*
- The final one is located close to the system's output ports to guarantee that it covers all anticipated output values. The verification engineer then ensures that the system's output values are being observed from this functional cover point. The uvm_subcriber, which is designed to cover the data monitored from the output interface, can accomplish this. *(C4)*

adc_uvm_subscriber_in	dac_uvm_subscriber_in
`class adc_subscriber_in extends uvm_subscriber #(adc_transaction_in);`	`class dac_subscriber_in extends uvm_subscriber #(dac_transaction_in);`

```
   …
   real VREF;
   real VIN;

   parameter real A_VDD = 2.5;
       // High Supply
   parameter int n = 3;
       // Resolution (Accuracy)
   int n_levels = $pow(2,n);
       // Number levels of conversion
   real delta = (A_VDD / n_levels);

 covergroup cover_real;
  option.per_instance = 1;
  coverpoint VREF {
   type_option.real_interval = 0.001;
   bins vref [] = {[(A_VDD-(delta)) :
                  (A_VDD +( delta))]};
      }

    coverpoint VIN {
    type_option.real_interval = 0.1;
        bins vin [] = {[0: A_VDD]};
      }
    endgroup: cover_real

 function void write
              (adc_transaction_in t);
  …
     VREF = t.VREF;
     VIN = t.VIN;
     cover_real.sample();
 endfunction: write
 …
endclass: adc_subscriber_in
```

```
    …
    parameter int n = 3;
       // Resolution (Accuracy)
    int n_levels = $pow(2,n);
       // Number levels of conversion

    logic [n-1:0] Q;

covergroup cover_logic;
  option.per_instance = 1;
  coverpoint Q {
    bins q [] = {[0: (n-1)]};
      }
endgroup: cover_logic


 function void write
               (dac_transaction_in
t);
      …
     Q = t.Q;
     cover_logic.sample();
 endfunction: write
 …
endclass: dac_subscriber_in
```

adc_uvm_subscriber_out

```
class    adc_subscriber_out    extends
uvm_subscriber
#(adc_transaction_out);
   …

  parameter int n = 3;
       // Resolution (Accuracy)
  int n_levels = $pow(2,n);
       // Number levels of conversion

    logic [2:0] Q;

    covergroup cover_bus;
      option.per_instance = 1;
      coverpoint Q {
        bins q [] = {[0:(n-1)]};
      }
    endgroup: cover_bus

    function void write
              (adc_transaction_out t);
      …
      Q = t.Q;
      cover_bus.sample();
    endfunction: write

endclass: adc_subscriber_out
```

dac_uvm_subscriber_out

```
class    dac_subscriber_out    extends
uvm_subscriber
#(dac_transaction_out);
    …
   real VOUT_A;

   parameter real A_VDD = 2.5;
        // High Supply
   parameter int n = 3;
        // Resolution (Accuracy)
   int n_levels = $pow(2,n);
        //    Number    levels    of
conversion
   real delta = (A_VDD / n_levels);

 covergroup cover_real;
   option.per_instance = 1;
   coverpoint VREF {
    type_option.real_interval        =
0.001;
    bins  vref  []  =  {[(A_VDD-(delta))
:
                    (A_VDD            +(
delta))]};
      }
 endgroup: cover_real
```

```
 function void write
              (adc_transaction_in t);
     …
     VOUT_A = t.VOUT_A;
     cover_real.sample();
 endfunction: write

endclass: dac_subscriber_out
```

Listing 14. SV code for uvm_subscriber of ADC/DAC

e. Scoreboards

The UVM Scoreboard is a verification component that checks, ensures, and verifies the functionality of the DUT. It receives the transactions from the monitor that are captured from the interfaces of DUT for both input and output ports. After receiving the transactions from the DUT input and the ones that came from the DUT output. It performs, builds, or models simple calculations from the input transaction and provides the expected output transaction. This operation is carried out using a reference model. Where, the reference model is a simple representation to the functionality of the DUT that links the input data of the DUT with a simple relation between the expected output and the input received from the DUT. The output transaction from the reference model is then compared with the output transaction received by the DUT.

ADC

```
class refmod extends uvm_component;
     `uvm_component_utils(refmod);
     …
     int n = 3;
     int nlevels = 2**n;
//No_levels_of_conversion

virtual task run_phase(uvm_phase phase);
 super.run_phase(phase);
 forever begin
  in.get(tr_in);
   if (tr_in.VIN >= vsuplow && VIN <
               ((tr_in.VREF)/n_levels))
   begin tr_out.Q = '0; end

   else if (tr_in.VIN >= ((n_levels-1)
            && tr_in.VIN <= tr_in.VREF)
   begin tr_out.Q = '1; end

   else if ((tr_in.VIN >=
             ((tr_in.VREF)/n_levels))
          && (tr_in.VIN < ((n_levels-1)
           * ((tr_in.VREF)/n_levels))))
   begin
    tr_out.Q = $floor(tr_in.VIN /
              ((tr_in.VREF)/n_levels));
   end
 out.put(tr_out);
 end
endtask: run_phase

endclass: refmod
```

DAC

```
class refmod extends uvm_component;
     `uvm_component_utils(refmod);
     …
     int n = 3;
     int nlevels = 2**n;
//No_levels_of_conversion

virtual task run_phase(uvm_phase phase);
 super.run_phase(phase);
 forever begin
  in.get(tr_in);
   if (tr_in.Q > '0 && tr_in.Q < '1)
   begin
    tr_out.vout_a = Q
             *((tr_in.VREF)/nlevels);
   end

   else if (tr_in.Q == '1)
   begin
    tr_out.vout_a = ((nlevels-1)
       * ((tr_in.VREF)/nlevels));
    end

   else if (tr_in.Q == '0)
   begin
    tr_out.vout_a = 0;
   end
  out.put(tr_out);
 end
endtask: run_phase

endclass: refmod
```

Listing 15. SV code for reference model of ADC/DAC

III. Check Linearity of Converter

The functional correctness of the converter is measured by the quantization error. The quantization error is the difference between the infinite resolution and the actual characteristics. It's equal to $\frac{1}{2}\,LSB = \pm\frac{\Delta}{2} = \frac{V_{FS}}{2^{n+1}}$ and considered as a noise added to the signal. The converter errors should be less than the quantization noise. The circuit errors are due to:

- The Component random mismatch due to the fabrication tolerances.
- The limitation in build block specification like gain, bandwidth, linearity, …

Checking the effect of the quantization errors in the converter functionality, can be derived by measuring the ADC linearity. The ADC linearity means measuring the transition deviations of the converter from the ideal characteristics. This is known by finding the INL and DNL.

A. Integral Non-Linearity (INL)

Maximum deviation of the code transitions from their ideal values in LSB. A converter is guaranteed to be monotonic if the maximum INL is less than ± 0.5 LSB. An Analog-to-Digital Converter (ADC) is monotonic if, for increasing analog voltage input, the digital output code increases and vice versa. Monotonic behavior does not guarantee that there will be no missing codes.

$$INL\ (in\ LSB) = \frac{V_{i\_real} - V_{i\_ideal}}{\Delta}$$

B. Differential Non-Linearity (DNL)

Maximum deviation in the step width from the ideal values of Δ in LSB. If $|DNL| \geq 1\,LSB$, this will result in a missing code.

$$DNL\ (in\ LSB) = \frac{V_i - V_{i-1}}{\Delta} - 1$$

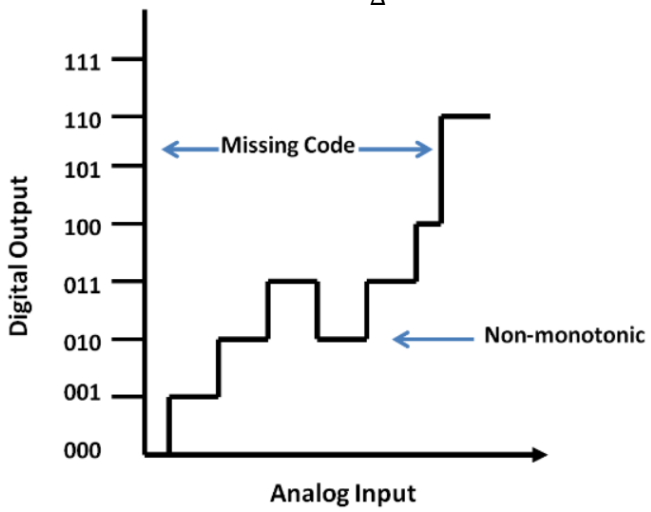


Figure 9. Non-Monotonic and Missing Code ADC

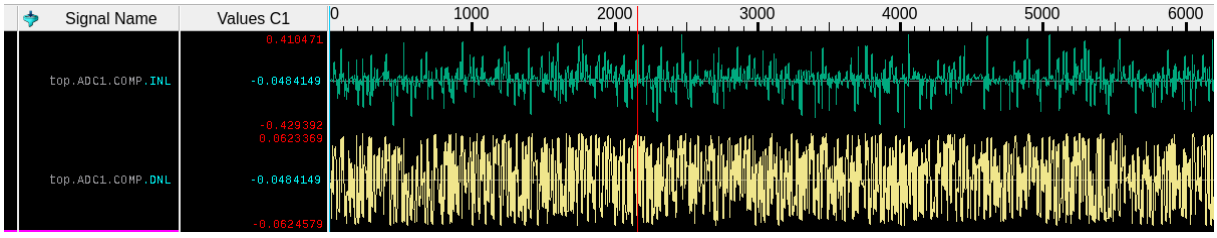


Figure 10. INL & DNL

Figure 10. illustrates the INL and DNL values without modeling the component mismatch, or without forcing the internal signals with the randomized values. If the sv code in listing 13. is applied with the content defined that the reference or supply voltage is randomized with the extent of the quantization noise and each voltage node at the resistive divider is reached also the extent of the quantization noise. Then the device could be non-monotonic or could have a missing code. Figure 11. the input voltage decreases from '1.75469' to '1.494', and the output code increases from '100' to '110'. The linearity is not maintained as the input voltage decreases while the output code increases and provides non-monotonicity in the ADC device as illustrated the red region of the failure assertion in figure 11. From Figure 12. the input voltage is equal to '1.04839' that should be coded to '100' but the code is missed as the DNL exceeds 1 LSB.

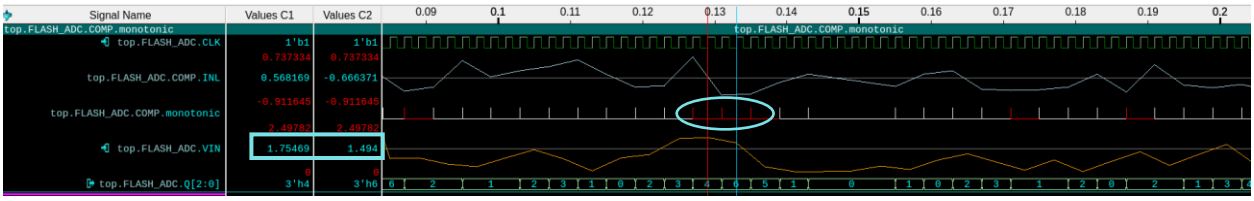


Figure 11. Non monotonic behavior

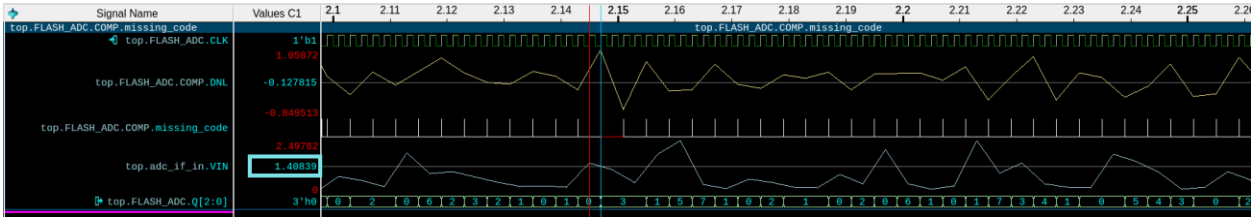


Figure 12. Missing Code

IV. Results

Figure 13. illustrates the transactions that are received at the reference model from the 'get_in(tr_in)' function; the input transaction has the randomized input voltage (tr_in.VIN) and reference/supply voltage (tr_in.VREF). The output transaction has the output calculated code from the reference model 'tr_out_refmod.Q' then this output compared with the output code from dut. The 'm_matches' signal is the number of the equivalent values and 'm_mistmatches' is the number of the non-equivalent values. If the extent of the quantization noise is applied on the reference/supply voltage and at each voltage node of the resistive divider, this will lead to make the ADC behave incorrectly and not as expected for that the number mismatches is increasing. The design engineer has to re-model the ADC and take into consider that the extent in the quantization noise for all the ADC components will make the ADC non-monotonic and could have missing codes.

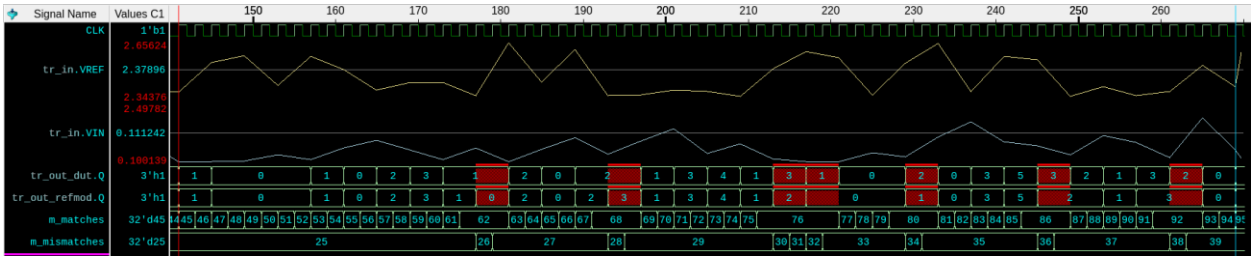


Figure 13. Scoreboard outputs due to max. limit model of quantization noise

Figure 14. illustrates if the quantization noise is modeled by an acceptable range that does not change the ADC output code. The number of matches output code is increasing with the simulation time while there are no mismatches between the output code from dut (real model) and the output code from reference model (ideal model).

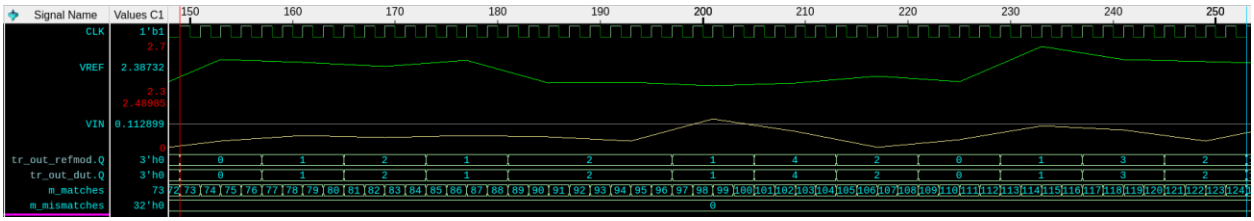


Figure 14. Scoreboard outputs with an acceptable quantization model range

The ‘coverpoints’ covered in the ‘uvm_subscriber’ can be debugged to know to what extent the values are covered. The ‘Coverage Summary’ window provides 95% are covered for three tests. The three errors are due to an assertion error in each test as illustrated in figure 16.

The ‘Covergroups’ windows can be used more to understand what are not covered. For example, the ‘vin’ does not have a zero value. Therefore, the user can rewrite the constraint on ‘vin’ to have zero value if this value is really needed.

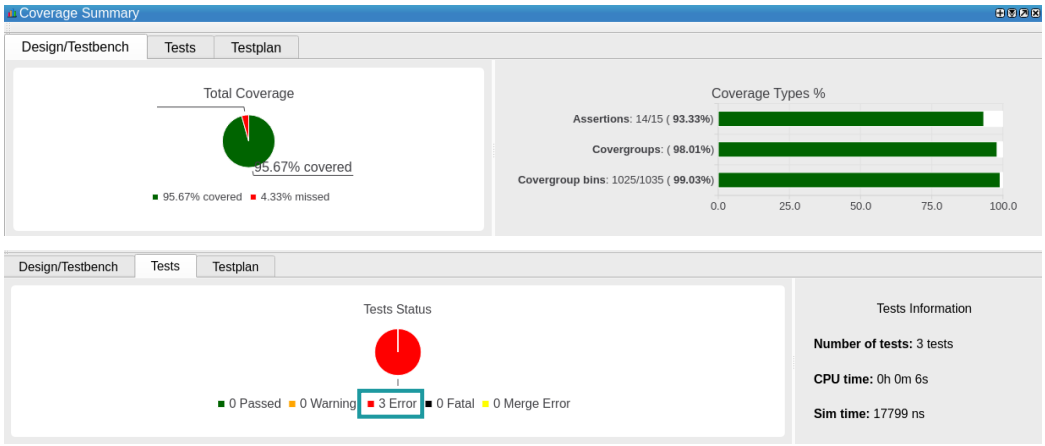
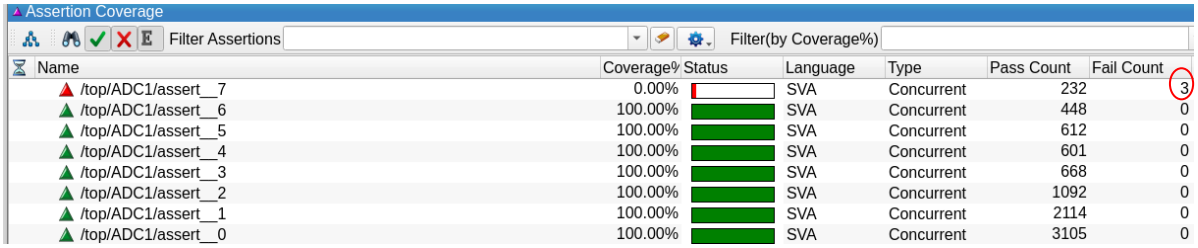
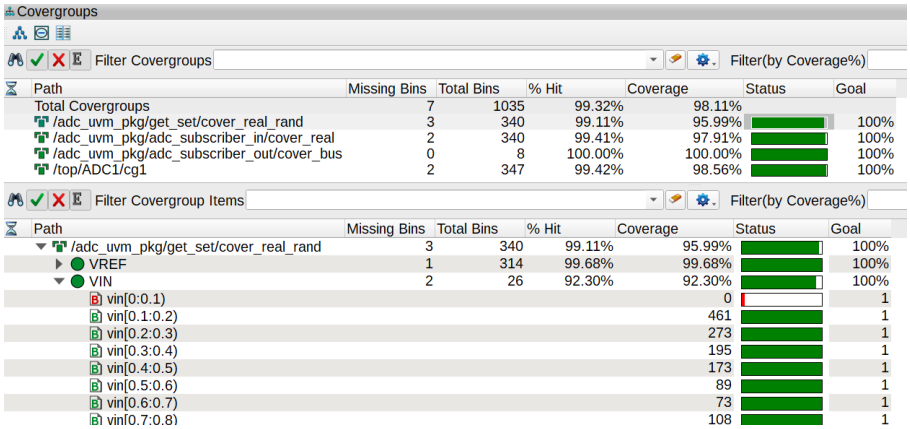


Figure 15. Coverage Summary



Name	Coverage%	Status	Language	Type	Pass Count	Fail Count
/top/ADC1/assert__7	0.00%		SVA	Concurrent	232	3
/top/ADC1/assert__6	100.00%		SVA	Concurrent	448	0
/top/ADC1/assert__5	100.00%		SVA	Concurrent	612	0
/top/ADC1/assert__4	100.00%		SVA	Concurrent	601	0
/top/ADC1/assert__3	100.00%		SVA	Concurrent	668	0
/top/ADC1/assert__2	100.00%		SVA	Concurrent	1092	0
/top/ADC1/assert__1	100.00%		SVA	Concurrent	2114	0
/top/ADC1/assert__0	100.00%		SVA	Concurrent	3105	0

Figure 16. Assertion Coverage



Path	Missing Bins	Total Bins	% Hit	Coverage	Status	Goal
Total Covergroups	7	1035	99.32%	98.11%		
/adc_uvm_pkg/get_set/cover_real_rand	3	340	99.11%	95.99%		100%
/adc_uvm_pkg/adc_subscriber_in/cover_real	2	340	99.41%	97.91%		100%
/adc_uvm_pkg/adc_subscriber_out/cover_bus	0	8	100.00%	100.00%		100%
/top/ADC1/cg1	2	347	99.42%	98.56%		100%

Filter Covergroup Items — Filter(by Coverage%)

Path	Missing Bins	Total Bins	% Hit	Coverage	Status	Goal
/adc_uvm_pkg/get_set/cover_real_rand	3	340	99.11%	95.99%		100%
VREF	1	314	99.68%	99.68%		100%
VIN	2	26	92.30%	92.30%		100%
vin[0:0.1)				0		1
vin[0.1:0.2)				461		1
vin[0.2:0.3)				273		1
vin[0.3:0.4)				195		1
vin[0.4:0.5)				173		1
vin[0.5:0.6)				89		1
vin[0.6:0.7)				73		1
vin[0.7:0.8)				108		1

Figure 17. Covergroups

The Simulator used in the paper, allows the randomization of the real type variables by default. The simulator has a switch to stop the randomization on the real datatype. The Debug environment enables debugging of the real signals and applies all the debug features without extra licenses.

V. Conclusion

In the digital environment, there are a lot of verification techniques that can be used to find bugs within a system. This makes the digital verification is always preferred as its reliability, and usage. Only one environment and one event-driven simulator can provide these verification techniques in an automated way. The paper illustrates:

- The constrained random verification. Such as randomizing the input components of ADC/DAC as modeling quantization mismatch in ADC/DAC. From this step, the digital verification engineer will be able to know the extent of the variation that each component can hold else the system will behave incorrectly.
- The functional coverage ensures that the electrical voltage is covered under a certain amplitude range.
- Assertions are built to check the functionality of the whole system if there is a simple relation between the output and input of a system.
- The UVM-based verification is one testbench environment that provides classes to support the randomization of ports through 'uvm_sequence' class. Supports the functional coverage through 'uvm_subscriber' class and assertions through 'uvm_scoreboard' class.

References

[1] Mariam Maurice, Mohamed Dessouky, and Ashraf Salem, "Increasing the Modeling Accuracy of an Analog PLL Device Executed With an Event-Driven Simulator," *IEEE Access*, vol. 11, pp. 79721-79793, July 2023.

[2] Mariam Maurice, "Modeling Analog Devices using SV-RNM," *Design and Verification Conference and Exhibition (DVcon), USA*, March 2022.

[3] Nikolaos Georgoulopoulos, Ioannis Giannou, and Alkiviadis Hatzopoulos, "UVM-Based Verification of a Mixed-Signal Design Using SystemVerilog," *International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, July 2018.

[4] Nikolaos Georgoulopoulos, and Alkiviadis Hatzopoulos, "UVM-based Verification of a Digital PLL Using SystemVerilog", *International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, July 2019.

[5] Bryan Lizon, Christopher Hall, Ryan Andrews, & Joachim Wuerker, "Fundamentals of Precision ADC Noise Analysis", *TEXAS INSTRUMENTS*, September 2020.

[6] Vivek Modi, & Cherry Bhargava, "Review of flash-sar ADC with pipeline operation suitable for better performance", *International Journal of Control Theory and Applications*, September 2016, pp. 5139-5148.