# Modeling Analog Devices using SV-RNM

Mariam Maurice
Siemens EDA, Cairo-Egypt, Mariam_Maurice@mentor.com

## I. Introduction

Nowadays, both analog and digital system blocks are located on the same chip. Most digital design engineers want to verify digital modeled blocks along with analog models for ensuring the functionality of both systems together. The most familiar language for digital design engineers to model analog blocks in the digital environment is SystemVerilog-Real Number Modeling (SV-RNM). RNM uses real number values to model the voltage and the current behaviors of the analog parts. This paper illustrates important definitions in RNM, like user-defined resolved nets, boundary elements, and interconnects. This paper also describes how analog devices such as DACs, ADCs, LDOs, Filters, and Image Sensors can be modeled with these definitions. Debugging and visualization of RNM constructs are important during the integration of a complete analog device and during connecting an analog device modeled with the SV-RNM language to a digital one.

## II. Definitions And RNM Debugging

### A. *User-Defined Net-types and Resolved Nets*

The usage of User-Defined Nets (UDN) increased because they can hold more than one value. As mentioned in [1], digital design engineers use UDN to hold the values of voltage, current, and resistance in only one net. This paper demonstrates a new modification to the UDN in [1], which also supports the capacitance value. So now the User-Defined Type (UDT) structure consists of five real data types (V, V_prev, I, C, R) and the UDN holds five values. Digital design engineers modify the UDN to be resolved by summation or averaging functions, which are very simple functions. But the idea of this part is to highlight that the resolved nets can have new technical improvements to achieve better functionality for the resolved nets. The example shown in Fig. 1, illustrates that these nets will be resolved by an impedance voltage division. The User-Defined Resolution (UDR) function takes any UDT input connected to the User-Defined Resolved Net (UDRN) from the values of voltages, capacitances, and resistances. And then it presents the value resolved after considering the effect of impedance between the driven input voltage and the output that the user is waiting to see its resolved value. The output voltage and the current are resolved by the equations in (1).

```
typedef struct {
  real V;        // VOLTAGE
  real V_prev; // PREVIOUS VOLT
  real I;         // CURRENT
  real R;        // RESISTANCE
  real C;        // CAPACITANCE
} EE_struct;  // "EE_struct" (UDT)
```

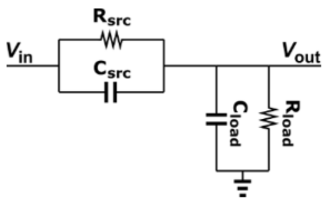


Figure 1. Impedance Voltage Division.

$$V = \frac{1}{A+B+C+D} * \left((B+D)*vin - (D)*vin_{prev} + (C+D)*vout_{prev}\right)$$

$$I = \frac{A+C}{A+B+C+D}\left((B+D)*vin - (D)*vin_{prev} + (C+D)*vout_{prev}\right) - C*vout_{prev} \qquad (1)$$

$$where\ A = \frac{1}{R_{Load}}, \quad B = \frac{1}{R_{src}}, \quad C = \frac{C_{Load}}{dt}, \quad D = \frac{C_{src}}{dt}$$

This resolved net can model a Low Pass (LP) filtering effect, a High Pass (HP) filtering effect, a resistance-voltage divider effect, a capacitance-voltage divider effect, a resistance-voltage effect, and a capacitance-voltage effect. Table I. shows which capacitor and resistor need to be set for reaching the desired effect. Fig. 2 shows the effect of low and high filtering using this resolved net. Also, it can model the effect of loading at any node. It can model the load effect of wires connected between sub-blocks of an analog system where the short wire can be modeled as a small capacitance equivalent to effect '5' in Table I. and the load effects at I/O port pins of an analog system such as ADC/DAC and LDO analog devices as these ports will be connected to other devices.

TABLE I
THE COMPONENTS OF THE RESOLVED NET THAT NEED TO BE SET

| | Effect | $R_{Load}$ | $R_{src}$ | $C_{Load}$ | $C_{src}$ |
|---|---|---|---|---|---|
| 1 | High Pass (HP) | Desired Value | $\approx \infty$ | $\approx 0$ | Desired Value |
| 2 | Low Pass (LP) | $\approx \infty$ | Desired Value | Desired Value | $\approx 0$ |
| 3 | Capacitive voltage division | $\approx \infty$ | $\approx \infty$ | Desired Value | Desired Value |
| 4 | Resistive voltage division | Desired Value | Desired Value | $\approx 0$ | $\approx 0$ |
| 5 | Capacitive | $\approx \infty$ | $\approx 0$ | Desired Value | $\approx 0$ |
| 6 | Resistive | Desired Value | $\approx 0$ | $\approx 0$ | $\approx 0$ |

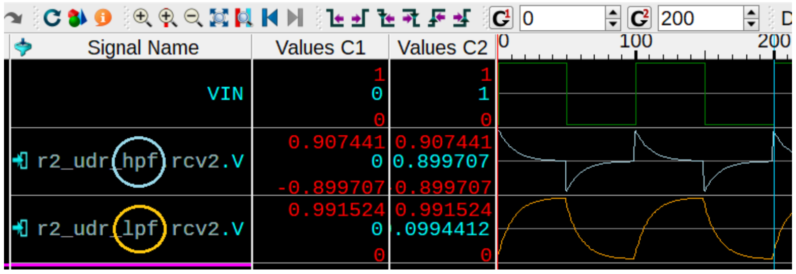
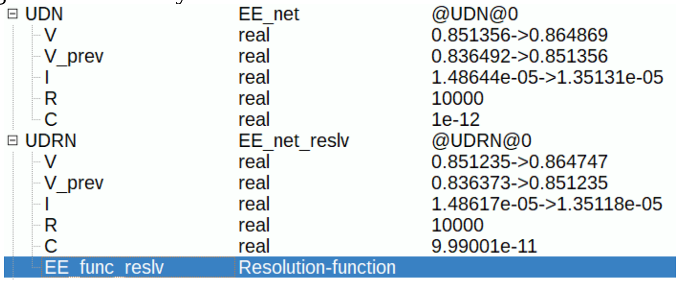


Figure 2. HP & LP filtering effect using resolved Net.

## *B. Debugging User-defined Net-types and Resolved Nets*

It’s important in the resolved nets to understand the functionality of the resolution function, and this is done when the debugging tools can distinguish between UDN and UDRN. Debugging tools should declare the resolution function as a variable of the UDRN that will help in debugging the functional correctness of the resolution function as illustrated in Fig. 3, the UDRN has a resolution function while a UDN does not so debugging begins with tracking and understanding the functionality of this variable which is the resolution function.



```
⊟ UDN            EE_net          @UDN@0
   V             real            0.851356->0.864869
   V_prev        real            0.836492->0.851356
   I             real            1.48644e-05->1.35131e-05
   R             real            10000
   C             real            1e-12
⊟ UDRN           EE_net_reslv    @UDRN@0
   V             real            0.851235->0.864747
   V_prev        real            0.836373->0.851235
   I             real            1.48617e-05->1.35118e-05
   R             real            10000
   C             real            9.99001e-11
   EE_func_reslv Resolution-function
```

Figure 3. Difference between UDN & UDRN.

The debug tool needs to have a strong builder expression that can construct complex functions equivalent to the functionality of the resolution function because this helps in comparing the outputs from the resolution functions to those generated from the builder expression. These expressions generated from the builder expression can be saved in do files (files to save the executed commands, so expressions are saved in the form of executed commands rather than being rebuilt again) and then the parameters of the generated expressions can be easily changed according to the resolved net parameters. Every input signal will be affected by the generated expressions without having to create modules with the same functionality of the resolution function and without having to change the parameters of the modules. The example in Fig. 4, shows the LP filtering effect values generated from the resolved net (UDRN) are compared with the LP filtering effect values created from the builder expression (Fx) and if the numbers are approximately equal that means the values generated from UDRN are correct as illustrated in the red box during charging time (~0.95) and in the blue box during discharging time (~0.02). Building expressions on real data is not something easy at all, it needs a strong debug tool that can create such complex expressions.

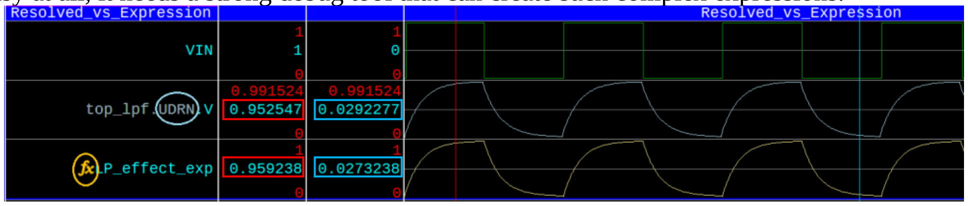


Figure 4. Values of LP filtering effect from UDRN and created expression.

## *C. Boundary Elements*

Most analog system blocks like serial links, image sensors, ADC & DAC can contain digital sub-blocks within the analog system itself such as encoders/decoders, serializers/de-serializers, latches & registers. In a digital environment, data that goes from an analog island (real variable) to a digital island (logic variable) must have some sort of an element to be inserted to convert between them. The element responsible for converting real var to logic var or vise-versa can be called Boundary Element (BE) or Adapter or Connect Modules. The net that shows the inserted BE is called MaSter-Net (MS-Net) because this net connects different data types (real & logic). The BEs are also used to convert power supplies in Analog Mixed Signals (AMS) systems. HDL models the high-supply digital logic with a ‘1’ and low-supply digital logic with a ‘0’. In any IC design, the power supply is always real data so the easiest way to convert it to logic is to insert a BE that can convert a high-supply analog value to logic ‘1’ and a low-supply analog value to logic ‘0’. In the past, HDL designers used to write Verilog code responsible for converting from Real to Logic (R2L) or from Logic to Real (L2R). But now most simulators support the automatic insertion of BEs by providing the MS-Net hierarchical path. The inserted BE can be defined as a 1-bit ADC (in the case of R2L) or as a 1-bit DAC (in the case of L2R) and it is expected in the future that EDA tools will insert n-bit ADC/DAC automatically. Inserting n-bit ADC/DAC automatically can eliminate modeling analog sub-blocks of ADC and DAC with specific topologies in the digital environment especially if the user is not interested in modeling undesirable effects of ADC/DACs and just wants to ensure the functionality of the system. The n-bit generic ADC/DAC, generic meaning without a specific topology to ADC/DACs, can be supported by the following functionality in (2). The SystemVerilog code that supports an n-bit ADC/DAC is shown in Table II.

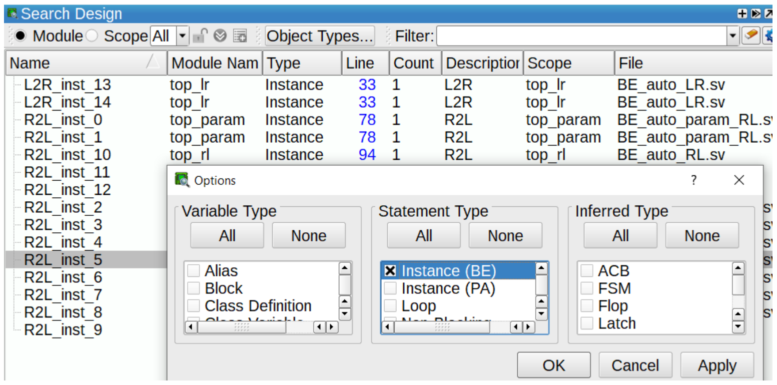
$$For\ n-bit\ ADC\!: L\left(output_{logic}\right) = \frac{R(input_{real})}{delta}, \qquad For\ n-bit\ DAC\!: L\left(output_{logic}\right) = R(input_{real}) * delta \tag{2}$$
$$where\ L\!: Logic,\ \ R\!: Real,\ \ delta = \frac{v_{sup}}{2^n},\ \ v_{sup}\!: supply\ value,\ \ 2^n\!: number\ levels\ of\ conversion,\ \ n\!: define\ resolution$$

TABLE II
SYSTEM VERILOG CODE FOR N-BIT GENERIC ADC/DAC

| n-bit ADC.sv | n-bit DAC.sv |
|---|---|
| // n to befine resolution (accuray) | // n to befine resolution (accuray) |
| // number of levels = 2**n | // number of levels = 2**n |
| // delta = (high supply voltage) / (number of levels) | // delta = (high supply voltage) / (number of levels) |
| | |
| module R2L #(parameter n = 3) | module L2R #(parameter n = 3) |
| (input real R, output wire [0:(n-1)] L); | (input wire [0:(n-1)] L, output real R); |
| | |
| parameter real vsup = 2.5; | parameter real vsup = 2.5; |
| parameter real vsuplow = 0; | parameter real vsuplow = 0; |
| parameter real nlevels = 2**n; | parameter real nlevels = 2**n; |
| parameter real delta = vsup / nlevels; | parameter real delta = vsup / nlevels; |
| | |
| reg [0:n-1] R_conv; | real L_conv; |
| | |
| always @ (R) begin | always @ (L) begin |
| if (R >= vsup) | if (L == '1) |
| R_conv = '1; | L_conv = vsup; |
| else if (R == vsuplow) | else if (L == '0) |
| R_conv = '0; | L_conv = vsuplow; |
| else if ((vsuplow < R) && (R < vsup)) | else if (('0 < L) && (L < '1)) |
| R_conv = R / delta; | L_conv = L * delta; |
| else | else |
| R_conv = 'X; | L_conv = `wrealXState; |
| end | end |
| | |
| assign L = R_conv; | assign R = L_conv; |
| endmodule | endmodule |

## *D. Debugging Boundary elements (BEs)*

The first thought while debugging BEs is about finding the desired BE to be debugged, so the tool should define it as a separate instance. This helps to know how many BEs are inserted, what type of the BE is inserted (R2L, L2R), and where they are inserted. When the BE is defined as a separate unique instance, the search can start to find it as a statement type as shown in Fig. 5.


Search Design

● Module ○ Scope All | Object Types... | Filter:

| Name | Module Nam | Type | Line | Count | Descriptior | Scope | File |
|---|---|---|---|---|---|---|---|
| L2R_inst_13 | top_lr | Instance | 33 | 1 | L2R | top_lr | BE_auto_LR.sv |
| L2R_inst_14 | top_lr | Instance | 33 | 1 | L2R | top_lr | BE_auto_LR.sv |
| R2L_inst_0 | top_param | Instance | 78 | 1 | R2L | top_param | BE_auto_param_RL.s |
| R2L_inst_1 | top_param | Instance | 78 | 1 | R2L | top_param | BE_auto_param_RL.s |
| R2L_inst_10 | top_rl | Instance | 94 | 1 | R2L | top_rl | BE_auto_RL.sv |
| R2L_inst_11 | | | | | | | |
| R2L_inst_12 | | | | | | | |
| R2L_inst_2 | | | | | | | |
| R2L_inst_3 | | | | | | | |
| R2L_inst_4 | | | | | | | |
| R2L_inst_5 | | | | | | | |
| R2L_inst_6 | | | | | | | |
| R2L_inst_7 | | | | | | | |
| R2L_inst_8 | | | | | | | |
| R2L_inst_9 | | | | | | | |

Options

| Variable Type | Statement Type | Inferred Type |
|---|---|---|
| All / None | All / None | All / None |
| ☐ Alias | ☒ Instance (BE) | ☐ ACB |
| ☐ Block | ☐ Instance (PA) | ☐ FSM |
| ☐ Class Definition | ☐ Loop | ☐ Flop |
| Class Variable | Non-Blocking | ☐ Latch |

OK | Cancel | Apply

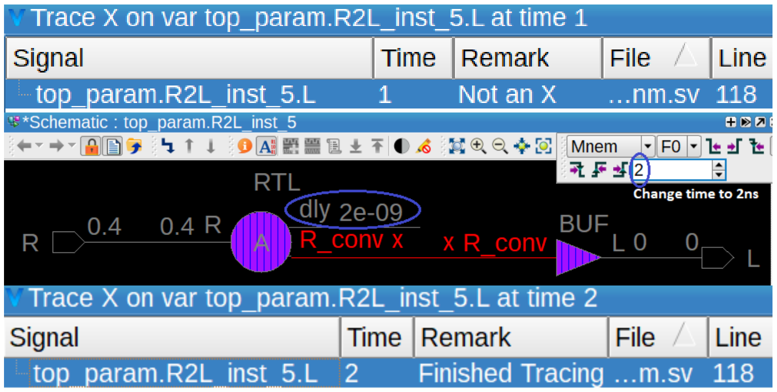
Figure 5. Searching for an instance BE.

After selecting the desired BE that needs to be debugged, the BE parameters must be well understood because the designer can change the default values of the BE inserted through configurable files and these values will be propagated through all the design. Table III. may simplify the BE parameters whose values need careful debugging.

TABLE III
BE PARAMETERS

| Name | Description |
|---|---|
| Supply value (vsup) | The greatest supply value |
| Threshold high value (vthi) | From this value 'vthi' up to 'vsup' consider high |
| X delay (txdel) | The values between vthi to vtlo consider don't know "x". It can also be delayed by a value of time secs. |
| Threshold low value (vtlo) | From this value 'vtlo' down to 'vsuplow' consider low |
| Supply Low value (vsuplow) | The lowest supply value |

It is easy to visualize the highest and the lowest values but the values between need smart schematic tools to simply give hints when the 'x' happens. And then change the time and trace when the BE output becomes 'x' as shown in Fig. 6.


Trace X on var top_param.R2L_inst_5.L at time 1

| Signal | Time | Remark | File | Line |
|---|---|---|---|---|
| top_param.R2L_inst_5.L | 1 | Not an X | ...nm.sv | 118 |

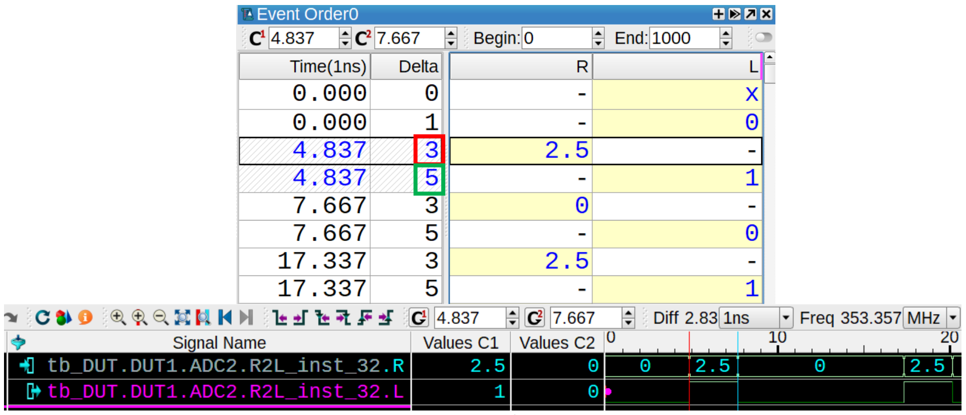
*Schematic : top_param.R2L_inst_5

Trace X on var top_param.R2L_inst_5.L at time 2

| Signal | Time | Remark | File | Line |
|---|---|---|---|---|
| top_param.R2L_inst_5.L | 2 | Finished Tracing | ...m.sv | 118 |

Figure 6. Schematic guiding tools to trace 'x'.

Adding the inserted BE input and output to a window that can show its changes over time will be very helpful. Since for any R2L instance there are a lot of real values changed to logic '1', some can be changed to logic '0' and some can be changed to 'x'. These windows can be a wave window or a smart window for debugging event-driven orders. In the example shown in Fig. 7, from the event order window, R (real) will be changed to '1' at 4.837 ns until the next event which is 7.667 ns where R will be changed to '0' that means R equals '1' from time 4.837 ns to time 7.667 ns or can be understood that R is converted at time 4.667 ns to logic '1' and to logic '0' at time 7.667 ns. R changed to '2.5' at time 4.837 ns, and a delta '3' in the red box means that R is the third signal to be changed in the database file at time 4.837 ns. R (real) converted to L =1 (logic) at time 4.837 ns, and delta '5' in the green box which means that L is the fifth signal to be changed in the database file at time 4.837 ns.


Event Order0

C1 4.837 C2 7.667 Begin: 0 End: 1000

| Time(1ns) | Delta | R | L |
|---|---|---|---|
| 0.000 | 0 | - | x |
| 0.000 | 1 | - | 0 |
| 4.837 | 3 | 2.5 | - |
| 4.837 | 5 | - | 1 |
| 7.667 | 3 | 0 | - |
| 7.667 | 5 | - | 0 |
| 17.337 | 3 | 2.5 | - |
| 17.337 | 5 | - | 1 |

C1 4.837 C2 7.667 Diff 2.83 1ns Freq 353.357 MHz

| Signal Name | Values C1 | Values C2 |
|---|---|---|
| tb_DUT.DUT1.ADC2.R2L_inst_32.R | 2.5 | 0 |
| tb_DUT.DUT1.ADC2.R2L_inst_32.L | 1 | 0 |

Figure 7. The event order for a 'real' variable.

## *E. Interconnects*

Some devices are structurally wire-based or the extracted netlist connections of a device can be wire-based which means that the wire type will be used to connect the sub-blocks of that system. If the system has analog (real) ports and the connection between its sub-blocks of wire net type, then there will be incompatibility errors during the simulation process as the real data type is connected directly with the wire net type. One can think that the solution is to insert two BEs, one R2L and one L2R, so that the wire is between the two inserted BEs (real --- R2L --- wire --- L2R --- real). This solution will result in a huge information loss and the real data will not be received as the desired value. Only two real values for the highest and lowest data will be received. Another solution to be considered is including n-bit ADC and n-bit DAC. Also, this solution will have information loss according to the number of levels of conversion. It is a hard solution if the tool does not support the automatic insertion of n-bit ADC/DAC and will also change the design hierarchy. The Best solution is to convert these wires to interconnect. From [2], since the interconnect is a type-less net and it can hold any value according to the type of data connected to it. Therefore, the best solution is to convert these wires that connect a real data type into interconnect as shown in Fig. 8.

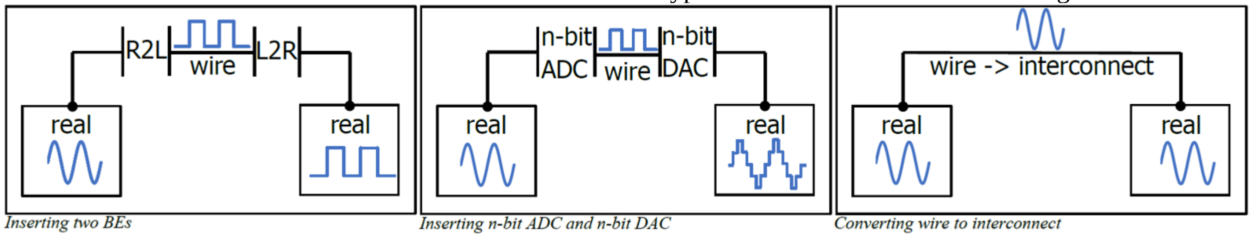
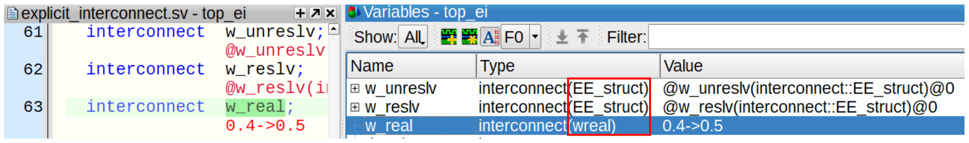


Figure 8. Wire to interconnect.

## *F. Debugging Interconnects*

The tool must differentiate between interconnects implicitly defined by the tool and those explicitly defined by the user because those implicitly defined by the tool convert the wire to interconnect as these wires are connected to real port data. Fig. 9 shows that the interconnect is implicitly defined in the Variables window as interconnect while the designer defined it in the Source window as wire. The debugger here can understand that the tool changes the wire to interconnect.


explicit_interconnect.sv - top_ei

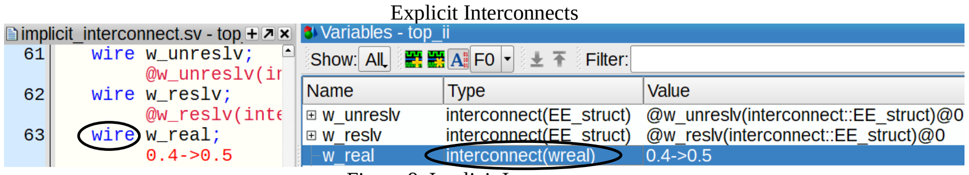
```
interconnect   w_unreslv;
               @w_unreslv
interconnect   w_reslv;
               @w_reslv(i
interconnect   w_real;
               0.4->0.5
```

Variables - top_ei

Show: All F0 Filter:

| Name | Type | Value |
|---|---|---|
| ⊞ w_unreslv | interconnect(EE_struct) | @w_unreslv(interconnect::EE_struct)@0 |
| ⊞ w_reslv | interconnect(EE_struct) | @w_reslv(interconnect::EE_struct)@0 |
| w_real | interconnect(wreal) | 0.4->0.5 |

Explicit Interconnects


implicit_interconnect.sv - top

```
wire w_unreslv;
     @w_unreslv(ir
wire w_reslv;
     @w_reslv(inte
wire w_real;
     0.4->0.5
```

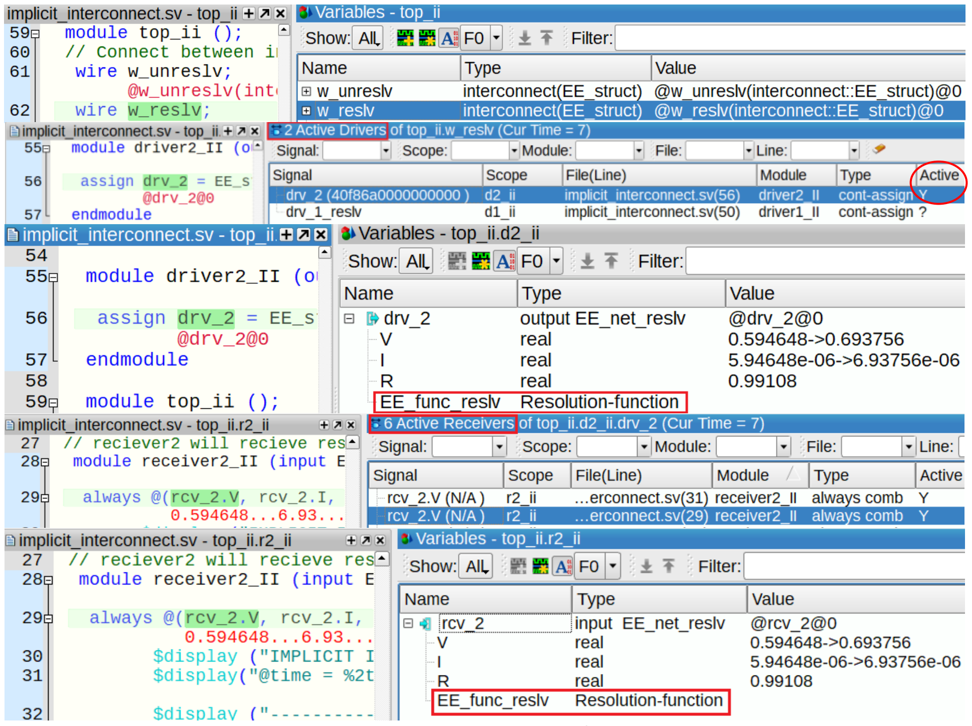
Variables - top_ii

Show: All F0 Filter:

| Name | Type | Value |
|---|---|---|
| ⊞ w_unresv | interconnect(EE_struct) | @w_unreslv(interconnect::EE_struct)@0 |
| ⊞ w_reslv | interconnect(EE_struct) | @w_reslv(interconnect::EE_struct)@0 |
| w_real | interconnect(wreal) | 0.4->0.5 |

Figure 9. Implicit Interconnects.

Since the interconnects can connect any type of data whether it is user-defined or not, the tool should highlight the data type driven by the interconnect. The red box in Fig.9, shows the data type that will be driven by the interconnect between brackets ‘()’. As shown in Fig.9, that one interconnect will drive a real data type and the other will drive ‘EE_struct’ which is likely to be a UDT. Select ‘w_reslv(EE_struct)’ as shown in Fig. 10, to show its drivers. There is an active driver, the status of Active is “Y”, then show the variables of this driver. You can also show the receivers of this driver, and then notice that this interconnect connects UDRN because the datatype of the port has a resolution function.



implicit_interconnect.sv - top_ii

```
module top_ii ();
// Connect between i
 wire w_unreslv;
      @w_unreslv(int
 wire w_reslv;
```

Variables - top_ii

Show: All F0 Filter:

| Name | Type | Value |
|---|---|---|
| ⊞ w_unreslv | interconnect(EE_struct) | @w_unreslv(interconnect::EE_struct)@0 |
| ⊞ w_reslv | interconnect(EE_struct) | @w_reslv(interconnect::EE_struct)@0 |

implicit_interconnect.sv - top_ii

```
module driver2_II (o
 assign drv_2 = EE_s
          @drv_2@0
endmodule
```

2 Active Drivers of top_ii.w_reslv (Cur Time = 7)

Signal: Scope: Module: File: Line:

| Signal | Scope | File(Line) | Module | Type | Active |
|---|---|---|---|---|---|
| drv_2 (40f86a0000000000 ) | d2_ii | implicit_interconnect.sv(56) | driver2_II | cont-assign | Y |
| drv_1_reslv | d1_ii | implicit_interconnect.sv(50) | driver1_II | cont-assign | ? |

implicit_interconnect.sv - top_ii

```

module driver2_II (o

 assign drv_2 = EE_s
          @drv_2@0
endmodule

module top_ii ();
```

Variables - top_ii.d2_ii

Show: All F0 Filter:

| Name | Type | Value |
|---|---|---|
| ⊟ drv_2 | output EE_net_reslv | @drv_2@0 |
| V | real | 0.594648->0.693756 |
| I | real | 5.94648e-06->6.93756e-06 |
| R | real | 0.99108 |
| EE_func_reslv | Resolution-function | |

implicit_interconnect.sv - top_ii.r2_ii

```
// reciever2 will recieve res
module receiver2_II (input E

 always @(rcv_2.V, rcv_2.I,
         0.594648...6.93...
```

6 Active Receivers of top_ii.d2_ii.drv_2 (Cur Time = 7)

Signal: Scope: Module: File: Line:

| Signal | Scope | File(Line) | Module | Type | Active |
|---|---|---|---|---|---|
| rcv_2.V (N/A ) | r2_ii | ...erconnect.sv(31) | receiver2_II | always comb | Y |
| rcv_2.V (N/A ) | r2_ii | ...erconnect.sv(29) | receiver2_II | always comb | Y |

implicit_interconnect.sv - top_ii.r2_ii

```
// reciever2 will recieve res
module receiver2_II (input E

 always @(rcv_2.V, rcv_2.I,
         0.594648...6.93...
      $display ("IMPLICIT I
      $display("@time = %2t

      $display ("----------
```

Variables - top_ii.r2_ii

Show: All F0 Filter:

| Name | Type | Value |
|---|---|---|
| ⊟ rcv_2 | input EE_net_reslv | @rcv_2@0 |
| V | real | 0.594648->0.693756 |
| I | real | 5.94648e-06->6.93756e-06 |
| R | real | 0.99108 |
| EE_func_reslv | Resolution-function | |

Figure 10. Debugging drivers and receivers of an interconnect.

## III. DUT (Device Under Test)

The purpose of this section is to show that there are a lot of analog devices that can be modeled with all the definitions defined in the previous sections from UDNs, BEs, and interconnects. These devices can be generators, ADCs, DACs, LDOs, Image Sensors, and Filters.

### A. *Generators*

The function generator is used to generate electrical waves such as sine, square, sawtooth, and triangular waves. These waves can be modeled according to (3). Any of these waves will be connected to another analog block, for that reason considering its loading effect could be important. From Fig. 11, the waveform can provide information about frequency, max/min amplitude, and the offset of the wave. The random Generator can be useful for modeling the noise input wave or adding an amount of noise to the input wave.

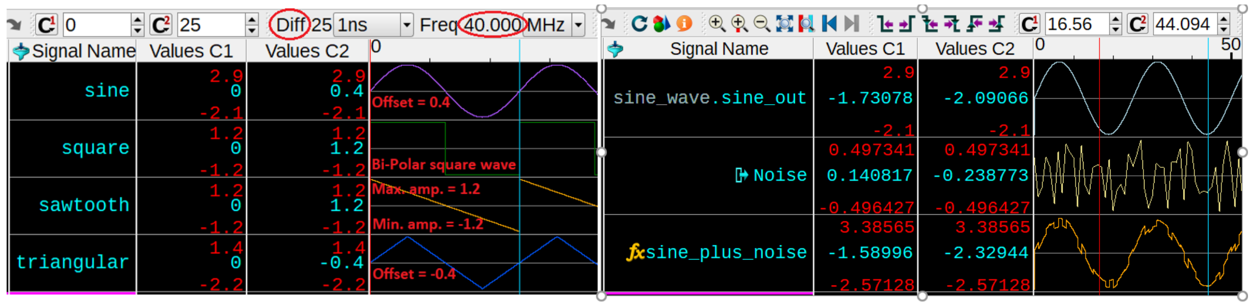


Figure 11. Waves and getting info from their waveforms.

$$\begin{gathered} sine_{wave} = offset + ampl * \$\sin(2 * pi * freq * \$time * unit_{time}) \\ square_{wave} = offset + ampl * (sgn(\$\sin(2 * pi * \$time * unit_{time}))) \\ sawtooth_{wave} = offset + ((2 * ampl)/pi) * \$atan\,(1/\$\tan(pi * freq * \$time * unit_{time})) \\ triangular_{wave} = offset + ((2 * ampl)/pi) * \$asin\,(1/\$\sin(pi * freq * \$time * unit_{time})) \end{gathered} \tag{3}$$

## B. *ADCs/DACs*

The DUT shown in Fig. 12, has the following devices: two Generators, two ADCs of 4-bit FLASH_ADC topology in [3], Logical Unit (LU), Arithmetic Unit (AU), one DAC of 4-bit R_STRING DAC topology in [4], and 4-bit generic DAC which is explained in part [II.C]. The inputs (wave1 and wave2) to the ADC (Analog to Digital Converter) can be real sine, triangular, sawtooth, square, or random voltage wave which is explained in part [III.A].

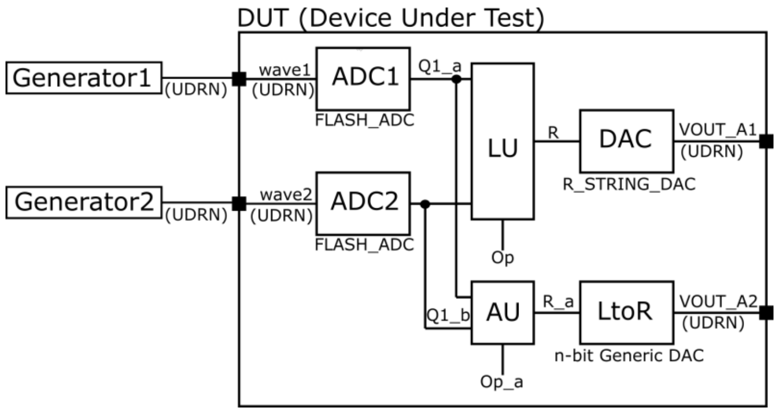


Figure 12. Analog-Mixed Signal DUT.

Wave1 and wave2 are driven by resolved net-types to model the load effect at the analog part, and this loading can reduce the value of the analog voltage that will be driven by the ADC. The reduction in these values can cause the ADC to convert an undesired analog voltage to digital. The system engineers can then re-design or re-model the loading effect from the generator part or the ADC part. Fig. 13 shows the structure of the FLASH_ADC. Inside the FLASH_ADC there are BEs inserted of R2L type, to convert the real output voltage value of the comparator to the digital input logic value of the encoder. The ADC sub-blocks are structurally wire-based connections, so a conversion from wire to interconnect is needed to receive the desired real data between the sub-blocks. The outputs of the two ADCs will go through the Logical Unit (LU) and the Arithmetic Unit (AU) which are purely digital devices.

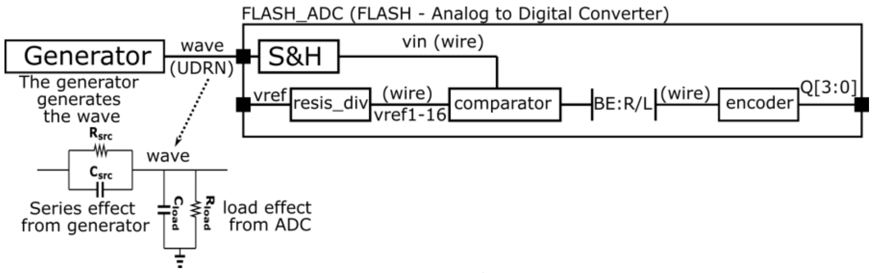
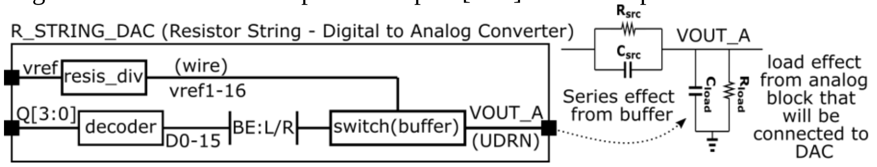


Figure 13. The structure of FLASH_ADC.

Then, the LU output will go through the R_STRING_DAC and the DAC's output is a resolved net. Fig. 14 shows the structure of R_STRING_DAC. Inside the DAC there are BEs inserted of L2R type. The DAC sub-blocks are structurally wire-based connections, so a conversion from wire to interconnect is required. The AU output will go through the n-bit generic DAC which is explained in part [II.C] and its output is also resolved net.

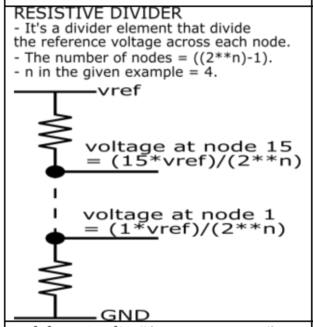
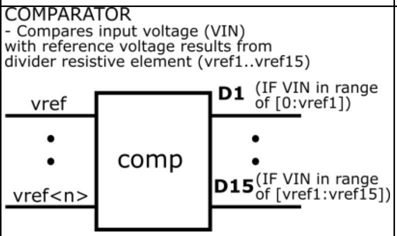
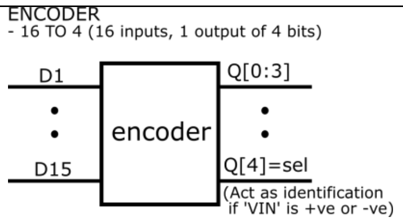


R_STRING_DAC (Resistor String - Digital to Analog Converter)
vref
resis_div
(wire)
vref1-16
Q[3:0]
decoder
D0-15
BE:L/R
switch(buffer)
VOUT_A
(UDRN)
R_src
C_src
VOUT_A
Series effect from buffer
C_load
R_load
load effect from analog block that will be connected to DAC

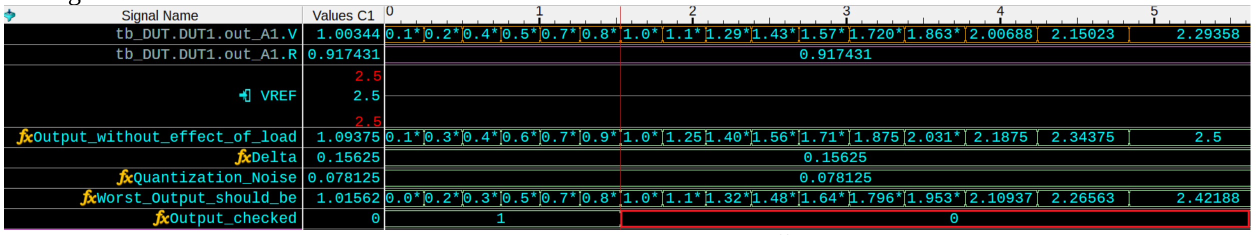
Figure 14. The structure of R_STRING_DAC.

Table IV. has the SystemVerilog code for FLASH_ADC sub-blocks, to explain the functionality of each sub-block and how to think about creating its behavioral model.

TABLE IV
SYSTEM VERILOG CODE FOR FLASH ADC

| resis_div.sv | comparator.sv | encoder.sv |
|---|---|---|
|  RESISTIVE DIVIDER<br>- It's a divider element that divide the reference voltage across each node.<br>- The number of nodes = ((2**n)-1).<br>- n in the given example = 4.<br>vref<br>voltage at node 15 = (15*vref)/(2**n)<br>voltage at node 1 = (1*vref)/(2**n)<br>GND |  COMPARATOR<br>- Compares input voltage (VIN) with reference voltage results from divider resistive element (vref1..vref15)<br>vref<br>vref<n><br>comp<br>D1<br>(IF VIN in range of [0:vref1])<br>D15<br>(IF VIN in range of [vref1:vref15]) |  ENCODER<br>- 16 TO 4 (16 inputs, 1 output of 4 bits)<br>D1<br>D15<br>encoder<br>Q[0:3]<br>Q[4]=sel<br>(Act as identification if 'VIN' is +ve or -ve) |
| module resis_div #(parameter n = 4)<br>(vref1, … , vref15, vref, GND);<br>always @(vref) begin<br>vref1 = (1*vref) / (2**n);<br>vref2 = (2*vref) / (2**n);<br>…<br>vref15_temp = (15*vref) / (2**n);<br>end<br>endmodule | module comparator (vref1, … vref15, D1, … , D15, VIN, sel, vref);<br>if ((0 <= VIN) && (VIN < vref1)) begin<br>D1 = 0; sel = 0;<br>end<br>…<br>else if ((vref15 <= VIN) && (VIN <= vref)) begin<br>D15 = vref; sel = 0;<br>end<br>…<br>else if ((-vref1 >= VIN) && (VIN > -vref2)) begin<br>D1 = vref; sel = 1;<br>end<br>…<br>endmodule | module ENCODER_16_TO_4 (D1, …, D15, sel, Q);<br>logic [3:0] Q_a;<br>always @(D1, … , D15, sel) begin<br>Q_a[3] = D8 \| D9 \| D10 \| D11 \| D12 \| D13 \| D14 \| D15;<br>Q_a[2] = D4 \| D5 \| D6 \| D7 \| D12 \| D13 \| D14 \| D15;<br>Q_a[1] = D2 \| D3 \| D6 \| D7 \| D10 \| D11 \| D14 \| D15;<br>Q_a[0] = D1 \| D3 \| D5 \| D7 \| D9 \| D11 \| D13 \| D15;<br>end<br>assign Q[3:0] = Q_a;<br>assign Q[4] = sel;<br>endmodule |

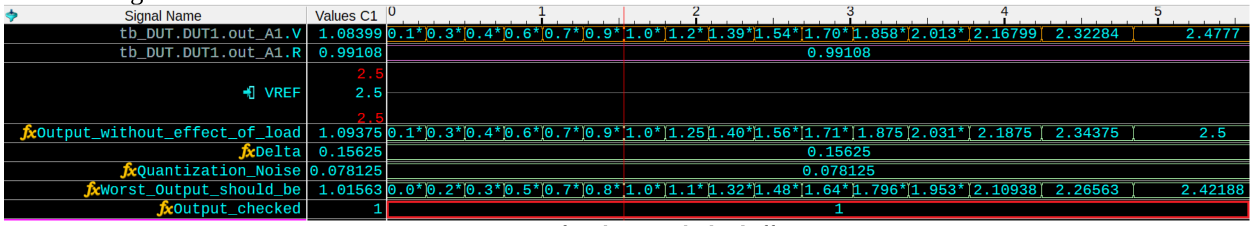
To check whether this loading effect is desirable or not, the designer can use a builder expression or a calculator to take the effect of the quantization noise on the net that will be affected by the load. Taking the worst quantization noise on this net will show how much the voltage on this net should be lowered or increased. In Fig. 15, the expression 'Output_Checked' means if this check gives a value of zero, all the values in the red box must be redesigned.



```
Signal Name                     Values C1  0      1      2      3      4      5
tb_DUT.DUT1.out_A1.V            1.00344    0.1* 0.2* 0.4* 0.5* 0.7* 0.8* 1.0* 1.1* 1.29* 1.43* 1.57* 1.720* 1.863* 2.00688  2.15023  2.29358
tb_DUT.DUT1.out_A1.R            0.917431   0.917431
VREF                            2.5
                                2.5
                                2.5
Output_without_effect_of_load   1.09375    0.1* 0.3* 0.4* 0.6* 0.7* 0.9* 1.0* 1.25 1.40* 1.56* 1.71* 1.875 2.031* 2.1875  2.34375  2.5
Delta                           0.15625    0.15625
Quantization_Noise              0.078125   0.078125
Worst_Output_should_be          1.01562    0.0* 0.2* 0.3* 0.5* 0.7* 0.8* 1.0* 1.1* 1.32* 1.48* 1.64* 1.796* 1.953* 2.10937  2.26563  2.42188
Output_checked                  0          1                    0
```

Figure 15. Build an expression to ensure the modeling effect is correct or not.

Therefore, there is something to change in modeling the loading effect of this net. For correction there are two ways, change the load effect of the DAC or change the load effect that will be connected to the DAC. Changing the DAC load means more complexity for the buffer sub-block specification of the DAC. Changing the load effect of the device to be connected to the DAC, means more complexity of the specifications of the connected device. After changing the load effect of the connected device to DUT. The 'Output_Checked' value is now all '1' values as shown in Fig. 16.

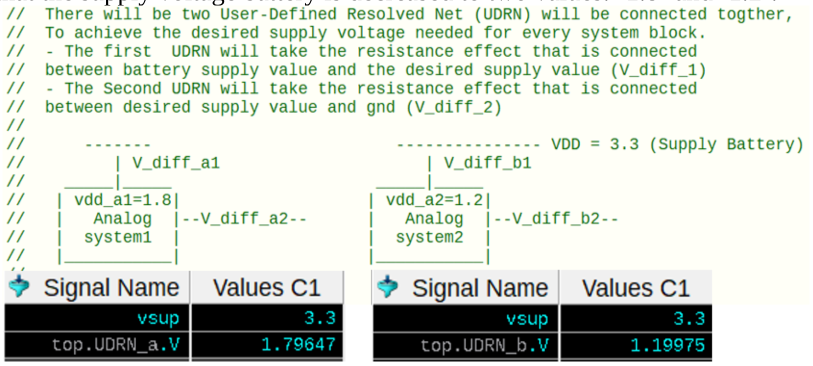


```
Signal Name                     Values C1  0      1      2      3      4      5
tb_DUT.DUT1.out_A1.V            1.08399    0.1* 0.3* 0.4* 0.6* 0.7* 0.9* 1.0* 1.2* 1.39* 1.54* 1.70* 1.858* 2.013* 2.16799  2.32284  2.4777
tb_DUT.DUT1.out_A1.R            0.99108    0.99108
VREF                            2.5
                                2.5
                                2.5
Output_without_effect_of_load   1.09375    0.1* 0.3* 0.4* 0.6* 0.7* 0.9* 1.0* 1.25 1.40* 1.56* 1.71* 1.875 2.031* 2.1875  2.34375  2.5
Delta                           0.15625    0.15625
Quantization_Noise              0.078125   0.078125
Worst_Output_should_be          1.01563    0.0* 0.2* 0.3* 0.5* 0.7* 0.8* 1.0* 1.1* 1.32* 1.48* 1.64* 1.796* 1.953* 2.10938  2.26563  2.42188
Output_checked                  1          1
```

Figure 16. After changing the load effect.

## C. *LDOs*

The concept of LDO (Low Drop-Out voltage) can be simply modeled as a resistive voltage divider. Since there is only one supply battery in any IP, the LDO model will be designed to reduce the value of the supply battery to the required supply voltage for each analog system. Also, it can be modeled with UDRN which is explained in part [II.A]. Fig. 17 shows that the supply voltage battery is decreased to two values: ‘1.8’ and ‘1.2’.



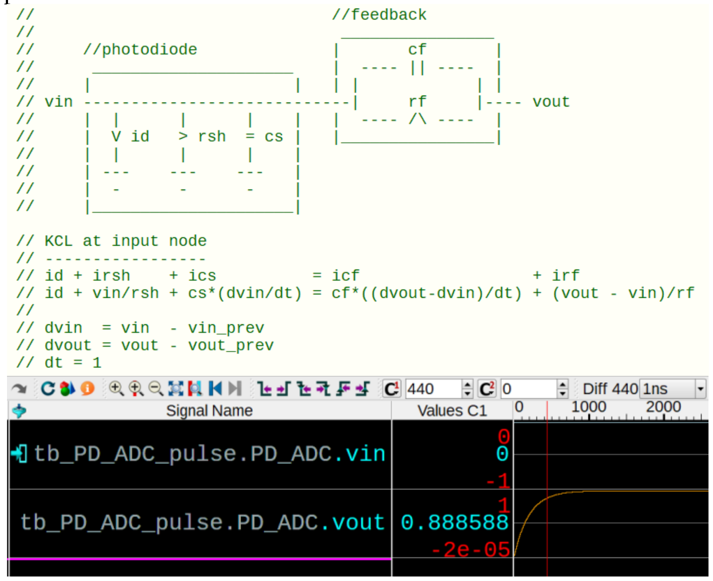
```
//  There will be two User-Defined Resolved Net (UDRN) will be connected togther,
//  To achieve the desired supply voltage needed for every system block.
//  - The first  UDRN will take the resistance effect that is connected
//  between battery supply value and the desired supply value (V_diff_1)
//  - The Second UDRN will take the resistance effect that is connected
//  between desired supply value and gnd (V_diff_2)
//
//      -------                         --------------- VDD = 3.3 (Supply Battery)
//         | V_diff_a1                       | V_diff_b1
//    _____|_____                       _____|_____
//   | vdd_a1=1.8|                     | vdd_a2=1.2|
//   |   Analog  |--V_diff_a2--        |   Analog  |--V_diff_b2--
//   |  system1  |                     |  system2  |
//   |___________|                     |___________|
```

```
Signal Name        Values C1      Signal Name        Values C1
       vsup              3.3             vsup              3.3
 top.UDRN_a.V        1.79647      top.UDRN_b.V        1.19975
```

Figure 17. LDO.

## D. *Image Sensors*

A digital image sensor circuit can be simply modeled as a PhotoDiode (PD) followed by an ADC that will convert the PD's analog output into a digital output. The PD can be modeled as a current source with approximate infinity resistance (rsh) and a small capacitor due to the effect of the materials (cs). The PD will be connected to a trans-impedance amplifier that converts its current output into voltage. The trans-impedance amplifier has an impedance effect of a capacitance parallel to a resistance (cf//rf). Fig. 18 shows the PD output when the input to PD is a pulse wave.



```
//                              //feedback
//                               _______________
//     //photodiode             |       cf      |
//      _____________________   |  ---- || ---- |
//     |                     |  | |           | |
// vin ---------------------------|     rf      |---- vout
//     |  |      |      |    |  |  ---- /\ ---- |
//     |  V id   > rsh  = cs |  |_______________|
//     |  |      |      |    |
//     | ---    ---    ---   |
//     |  -      -      -    |
//     |_____________________|

// KCL at input node
// -----------------
// id + irsh     + ics           = icf                   + irf
// id + vin/rsh + cs*(dvin/dt) = cf*((dvout-dvin)/dt) + (vout - vin)/rf
//
// dvin  = vin  - vin_prev
// dvout = vout - vout_prev
// dt = 1
```

```
C1 440   C2 0   Diff 440 1ns
Signal Name                     Values C1
tb_PD_ADC_pulse.PD_ADC.vin      0
tb_PD_ADC_pulse.PD_ADC.vout     0.888588
```

PhotoDiode.sv

```
module beh_pd (vout, vin);
  output vout; input  vin;
  real vout; real vin;
  real vout_prev; real vin_prev;
 // Parameters of photodiode
 parameter real id = 50E-6;
 parameter real cs = 100E-12; // series capacitance
 parameter real rsh = 1E9;     // shunt resistance
 // Parameters of feedback
 parameter real rf = 20E3;     // feedback resistance
 parameter real cf = 10E-12; // feedback capacitance
//
 parameter real TS = 1;
 parameter real TU = 1E-9;
 real dt = TS * TU;
 // Constants
 real A, B, C, D;
  always @(vin, sampling_clock) begin
   A = 1/rf; B = 1/rsh; C = cf/dt; D = cs/dt;
   vin_prev = vin;
   vout_prev = vout;
   vout = (1/(A+C)) * ( ((A+C+B+D) * vin) + (-(D+C) *
vin_prev) + (C * vout_prev) + id );
 end
endmodule
```

Figure 18. Photodiode and its SystemVerilog code.

## IV. Conclusion

Many analog devices can be modeled using the SV-RNM language. In the future much of the technical thinking will be towards modeling analog complex devices in the digital environment and will be supported by the EDA tools to enable simulation, debugging, and verification of such complex devices. Future work will be on how to verify these devices with UVM-based verification, assertions, and functional coverage.

## References

[1] John Brennan, Thomas Ziller, Kawe Fotouhi, Ahmed Osman, “The How To’s of Advanced Mixed-Signal Verification”, pp. 34-38, 2015.
[2] Stuart Sutherland, and Tom Fitzpatrick, “Keeping Up with Chip — the Proposed SystemVerilog 2012 StandardMakes Verifying Ever-increasing Design Complexity More Efficient”, pp. 5, February 28 – March 1, 2012
[3] K.Sai Kumar, K.Lokesh Krishna, K. Sampath Raghavendra, K.Harish, “A High Speed Flash Analog to Digital Converter”, pp.2, 2018.
[4] Jørgen Andreas Michaelsen, “Digital to Analog Converters”, pp. 9, spring 2013