Modeling Analog Devices using SV-RNM

Mariam Maurice
Siemens EDA, Mariam_Maurice@mentor.com

2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES
VIRTUAL
FEBRUARY 28 - MARCH 3, 2022

Abstract

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. This paper describes how analog devices such as DACs, ADCs, LDOs, Filters, and PhotoDiodes can be modeled from user-defined resolved nets, boundary elements, and interconnects. Debugging and visualization of RNM constructs are important during the integration of a complete analog device and during connecting an analog device modeled with SV-RNM language to a digital one.

User-Defined Net-types and Resolved Nets (UDN-UDRN)

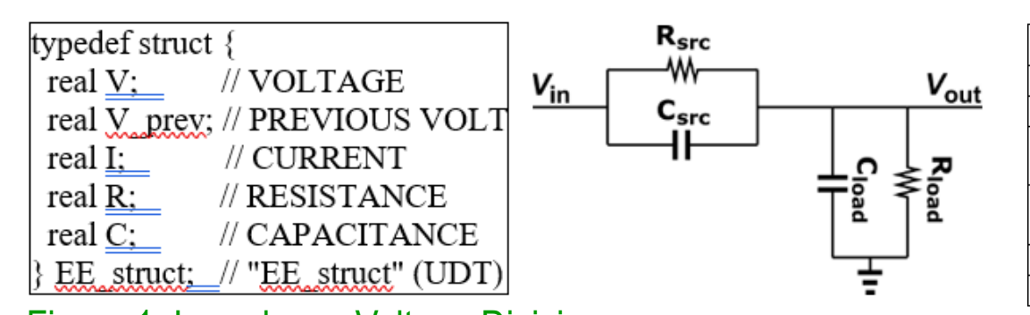
User-Defined Nets (UDN) hold the values of voltage, current, and resistance in only one net. New UDN can also support the capacitance value. Figure 1, illustrates that these nets will be resolved by an impedance voltage division. The output voltage and the current are resolved by the following equations.

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

$$I = \frac{A+C}{A+B+C+D}\begin{pmatrix}(B+D)*vin - (D)*vin_{prev}\\ +(C+D)*vout_{prev}\end{pmatrix} - C*vout_{prev}$$

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

This resolved net can model a Low Pass (LP)/High Pass (HP) filtering effect and a resistance-voltage/capacitance-voltage divider effect. Figure 2, shows which capacitor and resistor need to be set for reaching the desired effect. Figure 3, shows the effect of low and high filtering using this resolved net.

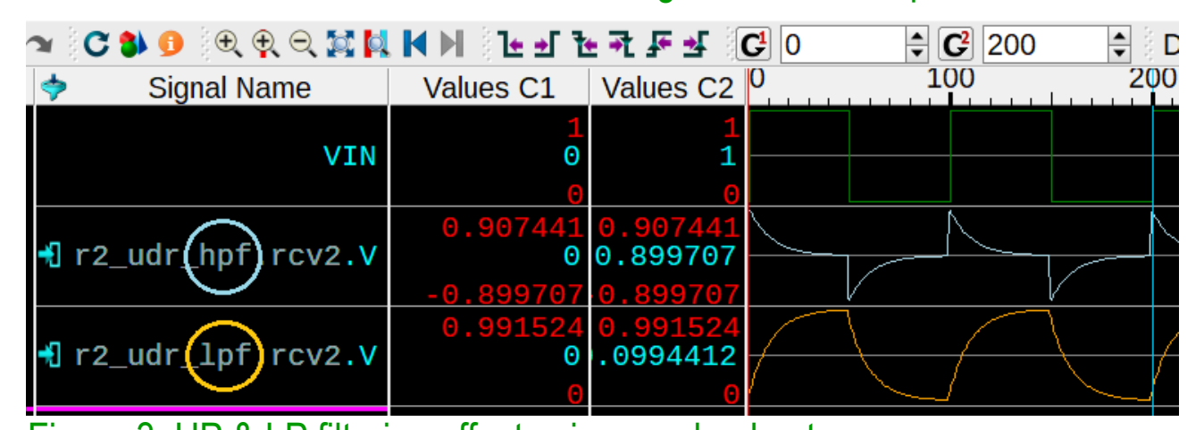


```
typedef struct {
  real V;       // VOLTAGE
  real V_prev; // PREVIOUS VOLT
  real I;        // CURRENT
  real R;       // RESISTANCE
  real C;       // CAPACITANCE
} EE_struct;  // "EE_struct" (UDT)
```

Figure 1: Impedance Voltage Division

	Effect	R_{Load}	R_{src}	C_{Load}	C_{src}
1	High Pass (HP)	Desired Value	≈ ∞	≈ 0	Desired Value
2	Low Pass (LP)	≈ ∞	Desired Value	Desired Value	≈ 0
3	Capacitive voltage division	≈ ∞	≈ ∞	Desired Value	Desired Value
4	Resistive voltage division	Desired Value	Desired Value	≈ 0	≈ 0
5	Capacitive	≈ ∞	≈ 0	Desired Value	≈ 0
6	Resistive	Desired Value	≈ 0	≈ 0	≈ 0

Figure 2: The components of the resolved net that need to be set

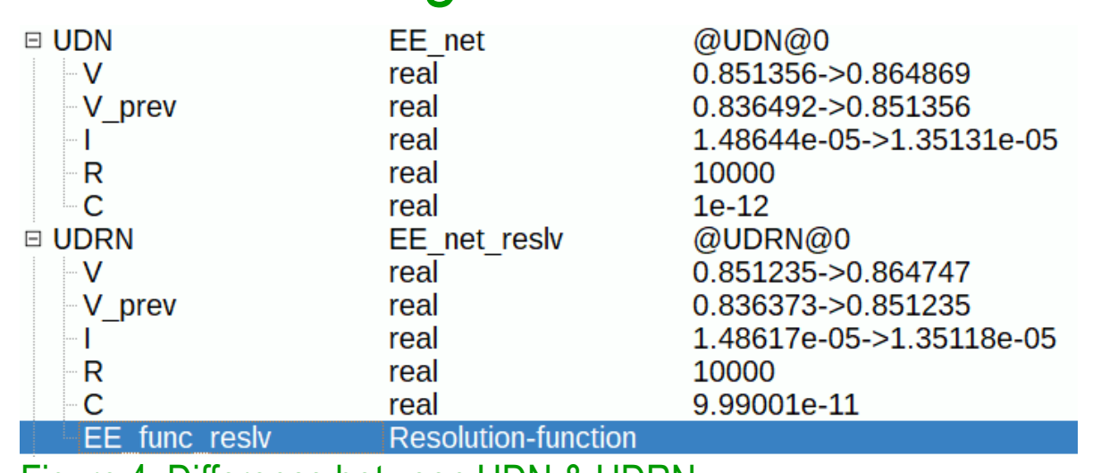


```
Signal Name        Values C1   Values C2
VIN                1 / 0 / 0   1 / 1 / 0
r2_udr(hpf)rcv2.V  0.907441    0.907441
                   0           0.899707
                   -0.899707   0.899707
r2_udr(lpf)rcv2.V  0.991524    0.991524
                   0           .0994412
                   0           0
```

Figure 3: HP & LP filtering effect using resolved net.

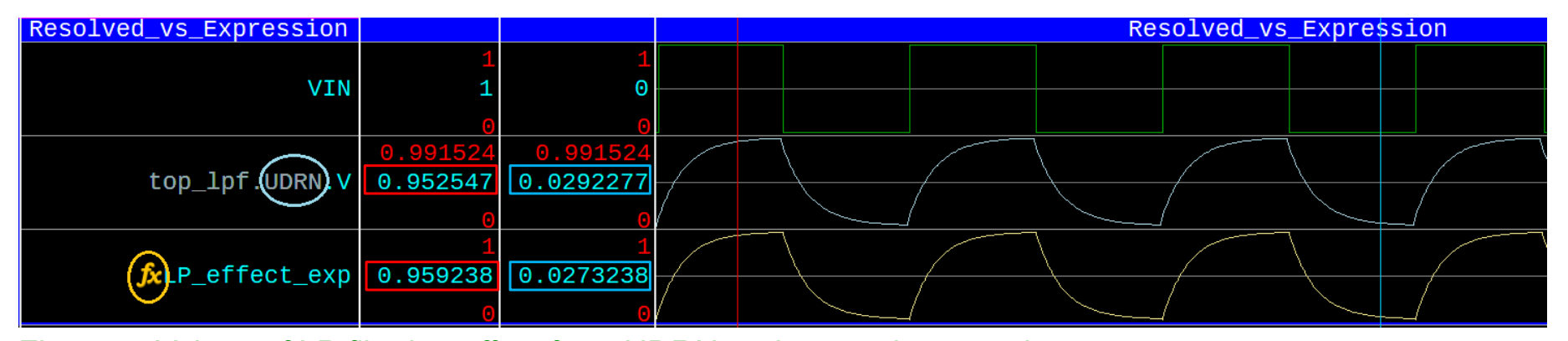
Debugging UDN-UDRN

- Understand the functionality of the resolution function, this is done when the debugging tools can distinguish between UDN and UDRN by declaring the resolution function as a variable of the UDRN as illustrated in Figure 4.
- 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 as shown in Figure 5.



```
⊟ 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 4: Difference between UDN & UDRN



```
Resolved_vs_Expression                      Resolved_vs_Expression
VIN                 1 / 1 / 0          1 / 0 / 0
top_lpf(UDRN)V      0.991524           0.991524
                    0.952547           0.0292277
LP_effect_exp       0.959238           0.0273238
```

Figure 5: Values of LP filtering effect from UDRN and created expression

Boundary Elements (BEs)

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. The n-bit generic ADC/DAC, generic meaning without a specific topology to ADC/DACs, can be supported by the following equation

$$For\ n-bit\ ADC: L(output_{logic}) = \frac{R(input_{real})}{delta},\qquad For\ n-bit\ DAC: L(output_{logic}) = R(input_{real}) * delta$$

$$where\ L: Logic,\ R: Real,\ delta = \frac{v_{sup}}{2^n},\ v_{sup}: supply\ value,\ 2^n: number\ levels\ of\ conversion,\ n: define\ resolution$$

The SystemVerilog code that supports an n-bit ADC/DAC can be shown in the following table

n-bit_ADC.sv

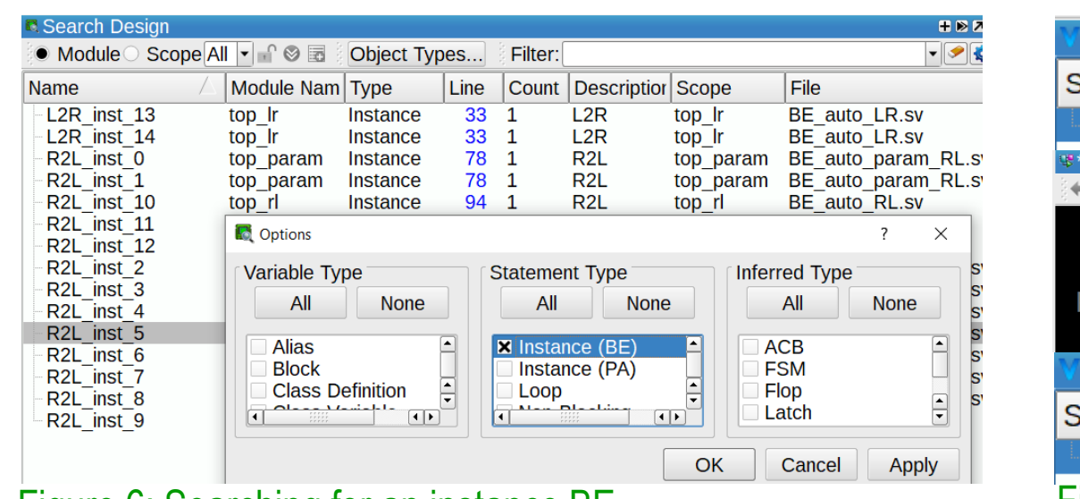
```
// n to befine resolution (accuray)
// number of levels = 2**n
// delta = (high supply voltage) / (number
of levels)

module R2L #(parameter n = 3) (input real
R, output wire [0:(n-1)] L);

parameter real vsup = 2.5;
parameter real vsuplow = 0;
parameter real nlevels = 2**n;
parameter real delta = vsup / nlevels;

reg [0:n-1] R_conv;

   always @ (R) begin
      if      (R === `wrealZState)
         R_conv = 'Z;
      else if (R >= vsup)
         R_conv = '1;
      else if (R === vsuplow)
         R_conv = '0;
      else if ((vsuplow < R) && (R < vsup))
         R_conv = R / delta;
      else
         R_conv = 'X;
   end

   assign L = R_conv;

endmodule
```

n-bit_DAC.sv

```
// n to befine resolution (accuray)
// number of levels = 2**n
// delta = (high supply voltage) / (number
of levels)

module L2R #(parameter n = 3) (input wire
[0:(n-1)] L, output real R);

parameter real vsup = 2.5;
parameter real vsuplow = 0;
parameter real nlevels = 2**n;
parameter real delta = vsup / nlevels;

real L_conv;

   always @ (L) begin
      if      (L === 'Z)
         L_conv = `wrealZState;
      else if (L === '1)
         L_conv = vsup;
      else if (L === '0)
         L_conv = vsuplow;
      else if (('0 < L) && (L < '1))
         L_conv = L * delta;
      else
         L_conv = `wrealXState;
   end

   assign R = L_conv;

endmodule
```

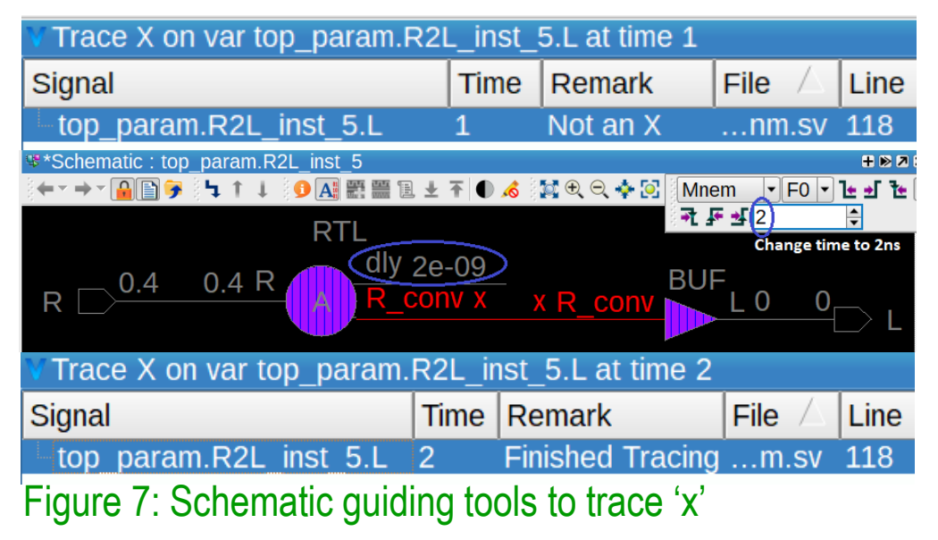
Debugging BEs

- The tool should define it as a separate instance. This helps to know how many BEs are inserted, where they are inserted and what type of the BE is inserted (R2L, L2R) as shown in Figure 6.
- 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. The highest and the lowest BE parameters values are easily visualized but the values between need smart schematic tools to simply give hints when the 'x' propagates as shown in Figure 7.
- Adding the BE input/output to windows like wave window or smart window for debugging event-driven orders will be very helpful as there are a lot of real values changed to logic '1', logic '0' and some can be changed to 'x' as shown in Figure 8.



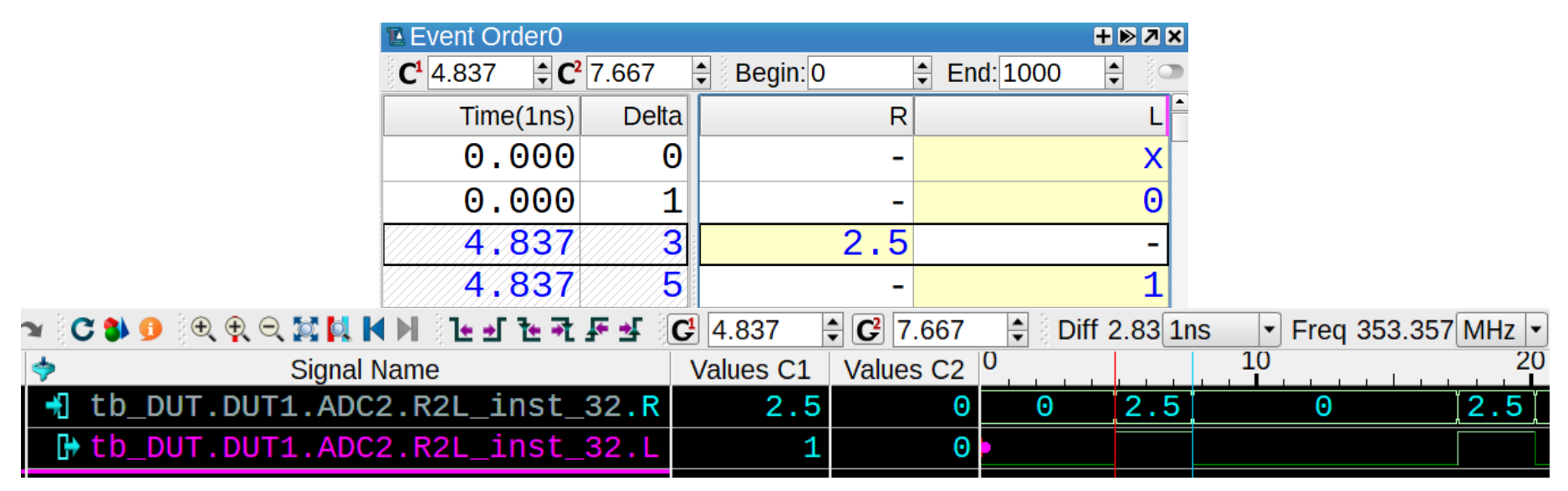
```
Search Design
Name         Module Nam  Type      Line  Count  Descriptio  Scope      File
L2R_inst_13  top_lr      Instance  33    1      L2R         top_lr     BE_auto_LR.sv
L2R_inst_14  top_lr      Instance  33    1      L2R         top_lr     BE_auto_LR.sv
R2L_inst_0   top_param   Instance  78    1      R2L         top_param  BE_auto_param_RL.sv
R2L_inst_1   top_param   Instance  78    1      R2L         top_param  BE_auto_param_RL.sv
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
```

Figure 6: Searching for an instance BE



```
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

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 7: Schematic guiding tools to trace 'x'



```
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

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 8: The event order for a 'real' variable

Interconnects

Some devices are structurally wire-based, or the extracted netlist connections of a device are wire-based. If the system is of real datatype, then there will be incompatibility errors during the simulation process. The solution could be:

- Insert two BEs, one R2L and one L2R, so that the wire is between the two inserted BEs (real -- R2L -- wire -- L2R -- real). Only two real values for the highest and lowest data will be received.
- Include n-bit ADC and n-bit DAC. 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.
- Convert these wires to interconnect. Interconnect can hold any value according to the type of data connected to it. Therefore, it is the best solution as shown in Figure 9.

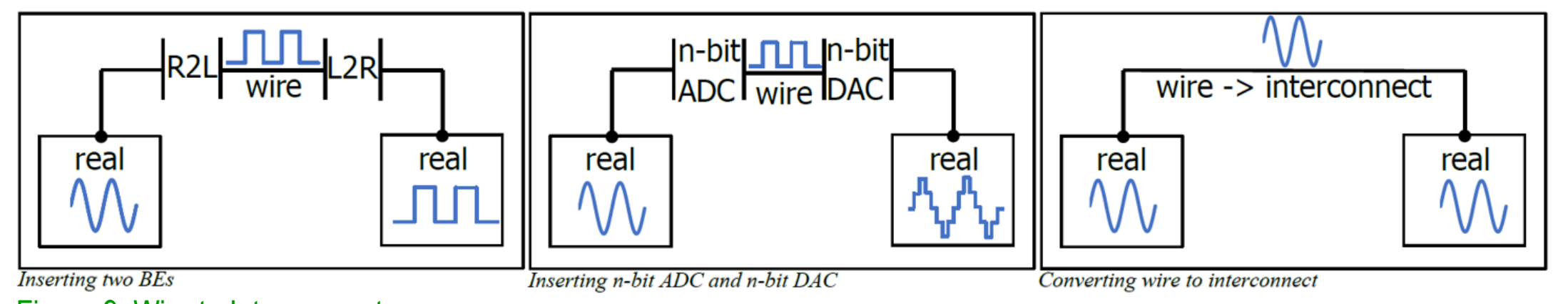
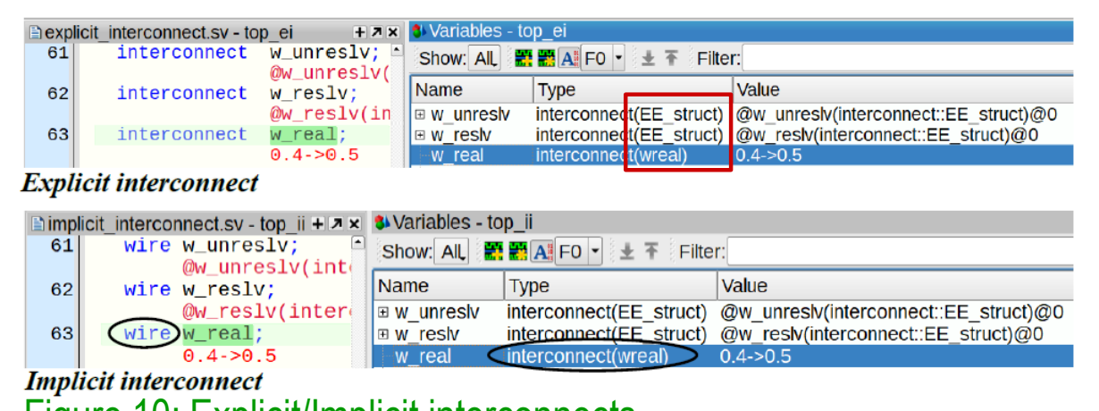


Figure 9: Wire to Interconnect

Debugging Interconnects

- Differentiate between interconnects implicitly defined by the tool and those explicitly defined by the user as in Figure 10.
- Highlight the data type driven by the interconnect, the red box in Figure 10 shows the data type that will be driven by the interconnect between brackets '()'.
- Show the drivers/receivers of interconnect. Notice that this interconnect connects UDRN because the datatype of the port has a resolution function as shown in Figure 11.



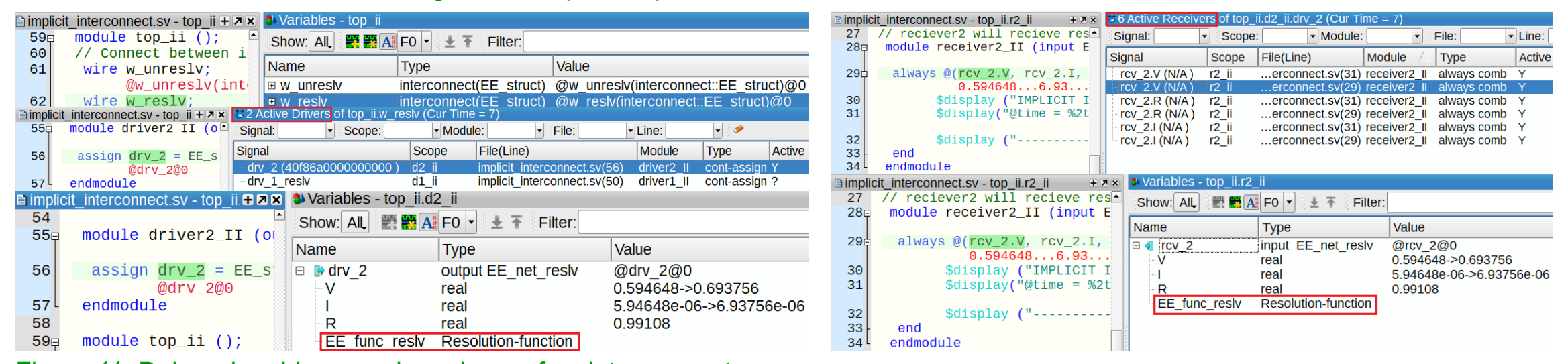
```
explicit_interconnect.sv - top_ei
61  interconnect w_unreslv;  @w_unreslv(
62  interconnect w_reslv;    @w_reslv(in
63  interconnect w_real;     0.4->0.5

Variables - top_ei
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 interconnect

```
implicit_interconnect.sv - top_ii
61  wire w_unreslv;  @w_unreslv(int
62  wire w_reslv;    @w_reslv(inter
63  wire w_real;     0.4->0.5

Variables - top_ii
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
```

Implicit interconnect

Figure 10: Explicit/Implicit interconnects



```
implicit_interconnect.sv - top_ii
59  module top_ii ();
60  // Connect between i
61  wire w_unreslv;  @w_unreslv(int
62  wire w_reslv;

Variables - top_ii
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

2 Active Drivers of top_ii.w_reslv (Cur Time = 7)
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
54
55  module driver2_II (o
56    assign drv_2 = EE_s  @drv_2@0
57  endmodule
58
59  module top_ii ();

Variables - top_ii.d2_ii
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
27  // reciever2 will recieve res
28  module receiver2_II (input E
29    always @(rcv_2.V, rcv_2.I,  0.594648...6.93...
30      $display ("IMPLICIT I
31      $display("@time = %2t
32      $display ("---------
33    end
34  endmodule

6 Active Receivers of top_ii.d2_ii.drv_2 (Cur Time = 7)
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
rcv_2.R (N/A)   r2_ii  ...erconnect.sv(31)  receiver2_II  always comb  Y
rcv_2.R (N/A)   r2_ii  ...erconnect.sv(29)  receiver2_II  always comb  Y
rcv_2.I (N/A)   r2_ii  ...erconnect.sv(31)  receiver2_II  always comb  Y
rcv_2.I (N/A)   r2_ii  ...erconnect.sv(29)  receiver2_II  always comb  Y

Variables - top_ii.r2_ii
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 11: Debugging drivers and receivers of an interconnect

DUT (Generator)

The function generator is used to generate electrical waves such as sine, square, sawtooth, and triangular waves. These waves can be modeled according to the following equations. Any of these waves will be connected to another analog block, for that reason considering its loading effect could be important.

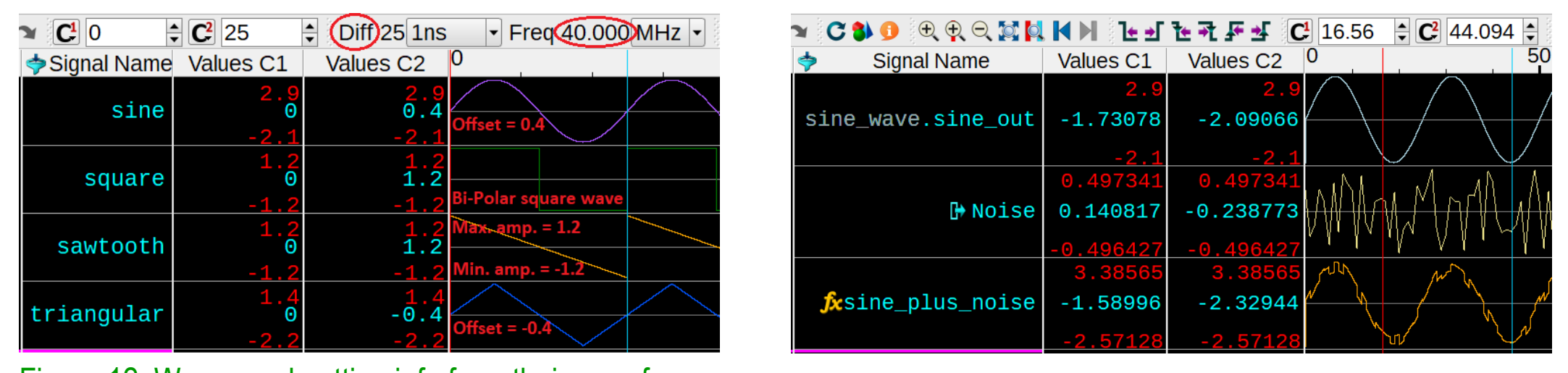
$$sine_{wave} = offset + ampl * \$\sin(2*pi*freq*\$time*unit_{time})$$

$$square_{wave} = offset + ampl * (sgn(\$\sin(2*pi*\$time*unit_{time})))$$

$$sawtooth_{wave} = offset + ((2*ampl)/pi) * \$atan\,(1/\$\tan(pi*freq*\$time*unit_{time}))$$

$$triangular_{wave} = offset + ((2*ampl)/pi) * \$asin\,(1/\$\sin(pi*freq*\$time*unit_{time}))$$

As shown in Figure 12, the tool 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.



```
C1 0   C2 25   Diff 25 1ns   Freq 40.000 MHz
Signal Name  Values C1        Values C2
sine         2.9 / 0 / -2.1   2.9 / 0.4 / -2.1
square       1.2 / 0 / -1.2   1.2 / 1.2 / -1.2
sawtooth     1.2 / 0 / -1.2   1.2 / 1.2 / -1.2
triangular   1.4 / 0 / -2.2   1.4 / -0.4 / -2.2

C1 16.56   C2 44.094
Signal Name          Values C1   Values C2
sine_wave.sine_out   2.9         2.9
                     -1.73078    -2.09066
                     -2.1        -2.1
Noise                0.497341    0.497341
                     0.140817    -0.238773
                     -0.496427   -0.496427
sine_plus_noise      3.38565     3.38565
                     -1.58996    -2.32944
                     -2.57128    -2.57128
```

Figure 12: Waves and getting info from their waveforms

DUT (ADC/DACs)

The DUT in Figure 13 has the following devices: two Generators, two ADCs of 4-bit FLASH_ADC, Logical Unit (LU), Arithmetic Unit (AU), one DAC of 4-bit R_STRING DAC and 4-bit generic DAC.

- 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 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 wire to interconnect is needed to receive the desired real data between 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. Then, the LU output will go through the R_STRING_DAC and the DAC's output is a resolved net. Inside the DAC there are BEs inserted of L2R type. The DAC sub-blocks are structurally wire-based connections, so a wire to interconnect is now required. The AU output will go through the n-bit DAC acts as a generic DAC and its output is also resolved net.

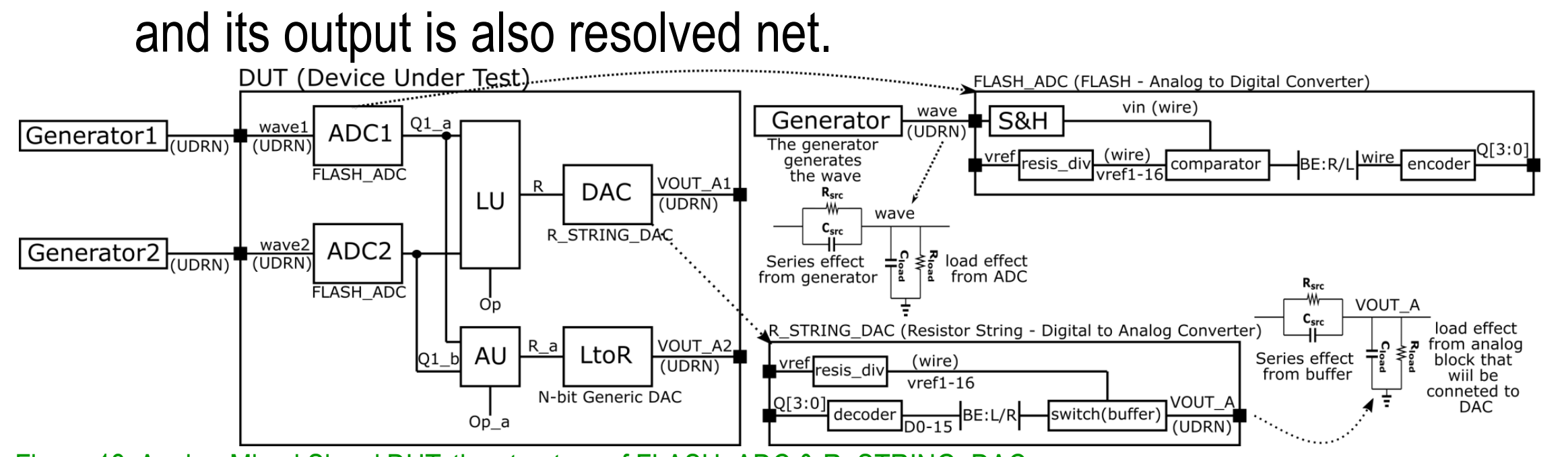
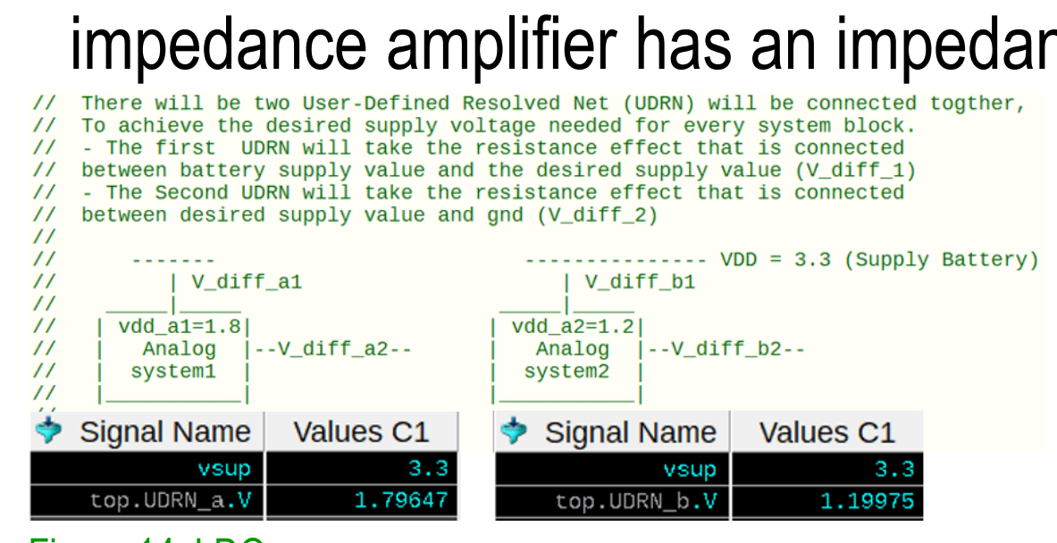


Figure 13: Analog-Mixed Signal DUT, the structure of FLASH_ADC & R_STRING_DAC

DUT (LDOs) / (PhotoDiodes)

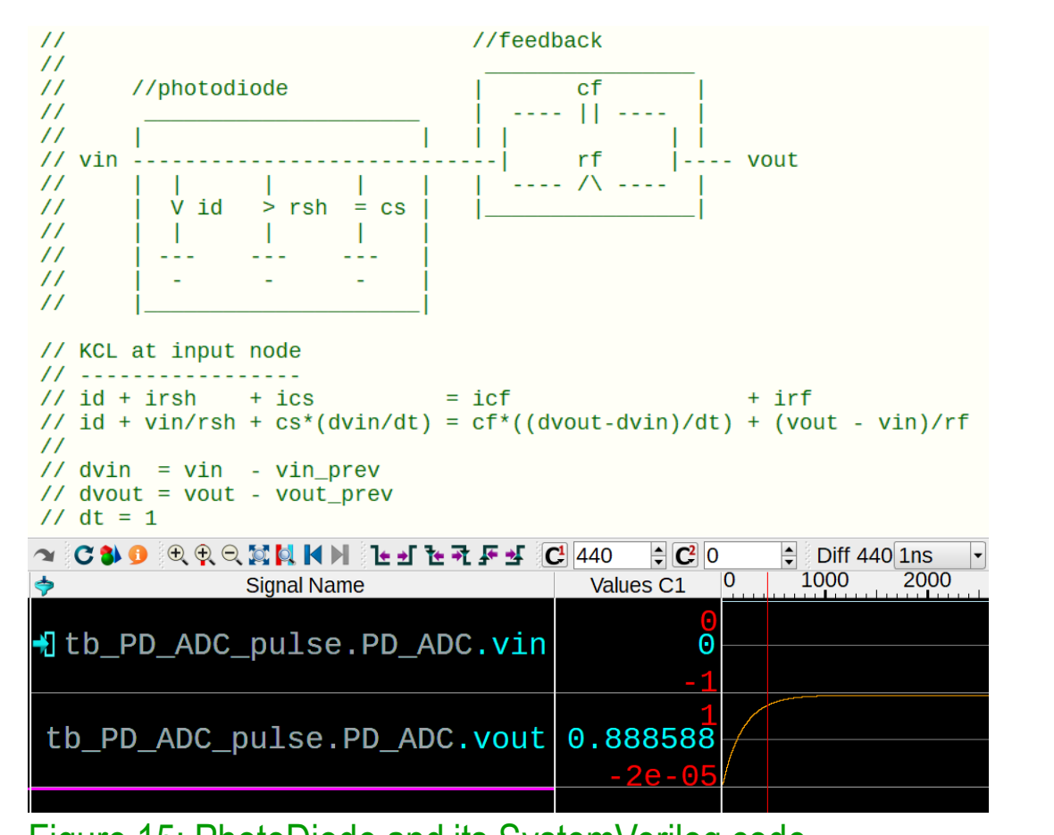
Figure 14: The concept of LDO (Low Drop-Out voltage) can be simply modeled as a resistive voltage divide. The LDO reduces the value of the supply battery to the desired analog system supply voltage. It can be modeled with UDRN.

Figure 15: PhotoDiode (PD) can be modeled as a current source with approximate infinity resistance (rsh) and small capacitor due to the effect of the materials (cs). The photodiode is then connected to a trans-impedance amplifier to convert current into voltage. The trans-impedance amplifier has an impedance effect of (cf//rf).



```
//  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 14: LDO



```
//                                  //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 / 0 / -1
tb_PD_ADC_pulse.PD_ADC.vout  1 / 0.888588 / -2e-05
```

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, samplying_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 15: PhotoDiode and its SystemVerilog code.

Conclusion

A lot of analog devices can be modeled using SV-RNM and in the future much of the technical thinking will be towards modeling analog complex devices in the digital environment and will be reflected in the EDA tools from simulation, debugging, and verification of such complex devices. Future work will be on how to verify these devices from UVM-based verification, assertions, and functional coverage.