How to use the new System Verilog Nettypes to address the Analog SoC Integration Verification

Joachim Geishauser
Thomas Theisen

Freescale Halbleiter Deutschland GmbH
Munich, Germany

www.freescale.com

ABSTRACT

Today's System-on-Chip (SoC) designs contain more and more analog functions. Verilog AMS simulation can be used to address the integration verification but do have their down sides.

An alternative method to do the integration verification was enabled with the new SystemVerilog Nettype constructs. This paper will give some insight in the evaluation of the new SystemVerilog Nettypes for a SoC project.

Table of Contents

Table of Figures

Table of Tables

Table of Listings

1. Introduction

Modern SoCs contain, next to the digital part, many analog Intellectual Properties (IP) such as ADCs or physical layers. While the verification of the analog blocks is done in continuous time simulators, their integration in the SoC is done in the digital domain. Here, complex analog signals are limited to 1 and 0 and the same testbench needs to handle the integration of the digital and analog IPs. While with Verilog 2001 the "real" values got introduced, which enable the modeling of analog behavior, it is not possible to drive or monitor real values from outside the chip. Thus for integration an additional simple model of the IP is necessary which uses digital signal replacement for analog signals.
Another problem is that in digital simulators only a limited number of data types is available and therefore all analog signals are of the same type. Thus it isn't possible to distinguish between different voltage domains. Connecting an analog IP to the wrong voltage domain will destroy the functionality with a high probability. Time-consuming design reviews are required to avoid wrong connections.
With SystemVerilog (SV) 2012 user-defined nettypes are introduced which are able to handle many of the above mentioned problems.

2. The Analog Verification Problem

As mentioned previously the user defined nettype should be able to cover a couple of verification use cases. Before the use cases are detailed, the project that was used for the update is outlined.

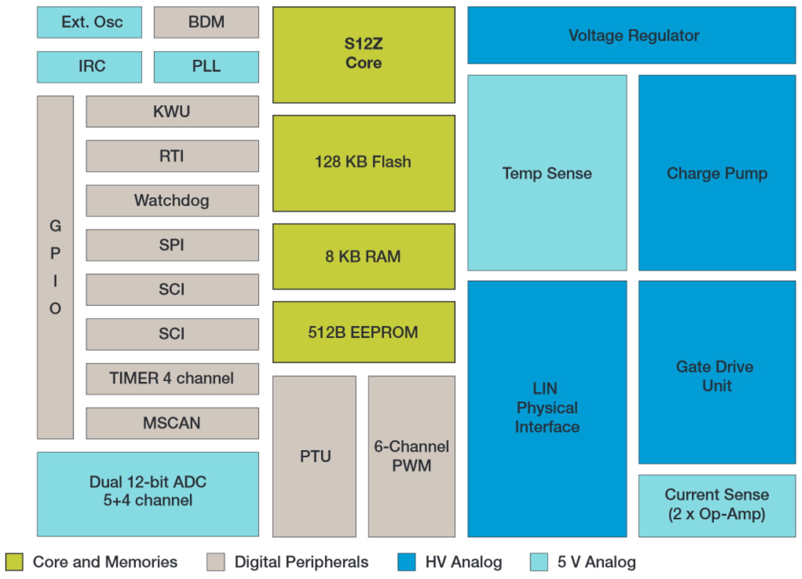

Figure 1 MagniV S12ZVML128 Block Diagram

Figure 1 shows the block diagram of the Freescale MagniV S12ZVML128 MCU. During the project all of the shown analog bocks, as well as the ones that are not shown – e.g. the pads got updated from simple Verilog behavioral models to SV user-defined nettypes.

It is important to note that the design only uses clock gating as power saving strategy. This implies that no IEEE1801-2013 power intent description is needed for the design. Nevertheless the impact of using the design along with a power description is outlined in the conclusion section as a forward looking task we did during the conversion process.

Verification Use Cases

The following sections outline different SoC analog integration verification use cases. These cases are further down the document then mapped to the SV nettype implementation and the identification of use case problems using the nettype implementation is explained.

Wrong analog connectivity
Connecting two signals with e.g. different voltage domains together should lead to an error. As a second case also connecting two signals wrongly must be detectable.

Broken analog connectivity
Analog modules which are not connected at all should be recognized.

n to 1 instead of 1 to 1 analog connection
It should be possible to detect additional accidentally connected drivers, such as pads or other modules, to one module.

1 to n instead of 1 to 1 analog connection
As final use case it shall be possible to detect drivers which are connected to multiple sinks.

3. Nettype

An Introduction to the Nettype

The new introduced user defined nettypes consist of three elements: A valid data type, a user defined resolution function and the type declaration.
The data type can be amongst others a real or 2/4 state integral type or a more complex data type defined in a structure [1, p. 57].
The main part of the nettype is the definition of the data type. An example for such a data type is shown below

```
typedef struct {
     real data;
               } data_type_t;
```

In addition to the data type SV allows to define a user defined resolution function. This function is called whenever there is a change of the value carried over to user defined nettype. To be more specific, the simulator will call the resolution function in the Active (or Reactive) region of the simulator. This resolution function gets called with the argument of a dynamic array of the user defined nettype. The function itself returns the user defined nettype. In our example the user defined resolution function looks like

```
function automatic data_type_t my_res_function (
    input data_type_t driver[]);
```

Finally the nettype needs to be declared

```
nettype data_type_t data_type with my_res_function;
```

In the declaration a nettype with the name data_type is defined. The new nettype can then be used in the code. Below an example

```
   data_type d;
   wire      w;
```

To connect two modules with ports of the same SV nettype a new net declaration is needed. This new net declaration was also introduced with the release of the IEEE1800-2012 release and is called `interconnect`. Such a connection example is shown in the code below.

```
module A (data_type out);
module B (data_type in);

interconnect      i;
A (.out(i));
B (.in(i));
```

4. The Realnet Nettype

The idea of our implementation of the nettype, called "*Realnet*", is that every driver consists of a voltage source and an output resistance. It enhances the idea developed in [2]. If a node is driven by two or more drivers, the equivalent voltage and resistance is calculated following Thevenins theorem. Load is modeled like a driver but with a voltage of 0 volt ("GND"). In Figure 2 an example is shown: Two drivers V_1 and V_2 with their corresponding output resistance R_1 and R_2 are driving node V_X. As load a resistor R_3 is connected to "GND". It is simplified to a single voltage source V_{eq} and resistance R_{eq}. Alternative concepts are presented in [3] and [4].

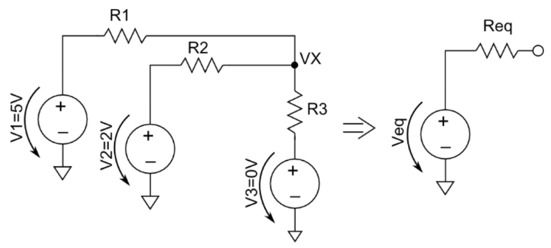

Figure 2 Simplifying a Circuit to a Single Source and Resistor using Thevenins Theorem

For two voltage sources the Thevenins equivalent voltage can be calculated by the following equation

$$V_{eq} = (I_1 + I_2)(R_1||R_2) = (I_1 + I_2)\frac{R_1 R_2}{R_1 + R_2} \tag{1}$$

Replacing currents by voltage divided by resistance gives

$$V_{eq} = (\frac{V_1}{R_1} + \frac{V_2}{R_2})\frac{R_1R_2}{R_1 + R_2} \quad (2)$$

Expanding (2) gives (3)

$$V_{eq} = \frac{V_1R_2 + V_2R_1}{R_1 + R_2} \quad (3)$$

The equivalent resistance is the parallel resistance of the resistors:

$$R_{eq} = \frac{R_1R_2}{R_1 + R_2} \quad (4)$$

Adding now an additional voltage source the same calculation can be done, replacing V_1 and R_1 by the previous calculated values V_{eq} and R_{eq}:

$$V_{eq_{new}} = \frac{V_{eq}R_3 + V_3R_{eq}}{R_{eq} + R_3} \quad (5)$$

and

$$R_{eq_{new}} = \frac{R_{eq}R_3}{R_{eq} + R_3} \quad (6)$$

Out of this it is possible to develop an algorithm.

Definition in SystemVerilog

As outlined before a nettype in SV consist of three elements: A datatype (real, struct, etc.), a resolution function and the definition of the nettype.
As datatype a structure with three elements is created:

```
typedef struct {
     real v_eq;         //Thevenins theorem equivalent voltage
     real r_eq;         //Thevenins theorem equivalent resistance
     real load;         //Number of drivers
                        } voltage_t;
```

v_eq and r_eq contain the above explained Thevenins equivalent values. The load value contains the number of drivers of this node. Additionally two important defines are provided:

```
`define realnetZState 1e23
`define realnetXState 1e22
```

These defines can carry digital values and help when solving the resolution function.

Next the resolution function is defined:

```
1.  function automatic voltage_t res_electrical(input voltage_t driver[]);
2.       voltage_t actualDriver;
3.       voltage_t resultOld;
4.       voltage_t result = '{0.0,`realnetZState, 0.0};
5.
6.       foreach (driver[i]) begin
7.          actualDriver = driver[i];
8.          //if there is an unknown driver coming in…
9.          if((`ABS(actualDriver.r_eq-`realnetZState)<1.0) || ((`ABS(actualDriver.v_eq-
 `realnetZState))<1.0)) begin
10.             // high imp driver has no influence to node
11.             // if there is only one driver to the node (open load),
12.             // the result will be high imp since result is init with
13.             // Z!
14.         end
15.         if((`ABS(actualDriver.r_eq-`realnetXState)<1.0) ||
16.            (`ABS(actualDriver.v_eq-`realnetXState)) < 1.0) begin
17.             //...the result will be unknown as well!
18.             result = '{0.0, `realnetXState,0.0};
19.             break;
20.         end
21.         else if((`ABS(result.r_eq-`realnetZState)<1.0) ||
22.                 (`ABS(result.v_eq-`realnetZState) < 1.0 )) begin
23.  //If result is high Imp. (=no infuence to node) the actual driver
24.  //will drive the node !!! result is init. as Z, so we go here when we
25.  //enter the first time the foreach loop!!!
26.             result = actualDriver;
27.         end
28.         else begin
29.             resultOld   = result;
30.             result.v_eq = (actualDriver.v_eq*resultOld.r_eq +
31.                            resultOld.v_eq*actualDriver.r_eq)/
32.                           (actualDriver.r_eq+resultOld.r_eq);
33.             result.r_eq = resultOld.r_eq*actualDriver.r_eq/
34.                          (resultOld.r_eq+actualDriver.r_eq);
35.         end
36.      end     //foreach(driver[i])
37.      result.load          = driver.size;
38.      // Init R to high impedance
39.      if (($realtime == 0) && (result.r_eq == 0))
40.        result.r_eq = `realnetZState;
41.      res_electrical = result;
42. endfunction : res_electrical
```

Listing 1 Realnet Resolving Function

The driver array contains all voltage sources which drive a certain node. The size of the driver array is stored in the load value of the structure, line 37. This load value can be used to check for connectivity errors.
The return value is the calculated voltage of the node. The calculation is done with a temporary variable "result" which is assigned to the return value "res_electrical" at the end. This seems to be less (tool) error prone. If the driver is a high ohmic signal it is ignored, line 9 to 14. If it is an unknown signal, the result voltage is unknown as well and further calculation is stopped, line 15 to 20. Running the loop the first time, indicated by the fact that the result is equal to Z, and the driver is a regular signal the result is equal to the driver. If only one driver is assigned to the node there is no further calculation necessary. The core algorithm presented in equation (1) to (6) is defined from line 28 to 35. For every additional driver these lines will be repeated. Then the number of drivers is saved and the temporary variable is assigned to the return variable.

Finally the net needs to be defined:

```
nettype voltage_t realnet with res_electrical;
```

Listing 2 shows an example of how the circuit in Figure 2 can be defined. Several ways of defining are used on purpose to show different possibilities. It is possible to assign a voltage directly, here the 5V, or define a voltage/resistance as real, V_2, and assign it per concatenation or define an additional net and assign this net like V_3.

```
realnet Vx;
real V2 = 2.0;
realnet V3 = '{0.0, 2000, 1};

assign Vx = '{5.0, 2000, 1};
assign Vx = '{V2, 2000, 1};
assign Vx = V3;
```

Listing 2 Assigning Voltages using Realnet

The voltage V_2 can now change within initial/always blocks as shown in the following example. The use of the variable V2 illustrates how real numbers inside an existing behavioral model can be connected to nettype connection to carry the information across to another module. This makes converting existing real model behaviors easier and is explained in more detail in chapter 5.

Comment [GJ1]: This builds the bridge to „reuse a model with real values for the use with nettype"

```
always @(posedge clk) begin
   V2 +=2.0;
end
```

The same concept can also be applied for current driven connections, such as current mirrors.

The Real Convergence Problem

As described in the last section, the backbone of the nettype is a structure that consists of real numbers. These real numbers cover a wide range of values but also introduce a problem known as rounding error. These rounding errors can lead to an infinite loop during simulation and make the simulation get stuck at a certain point in time.
To understand the root cause for this it is important to understand the function of the SV nettype. As described in the previous section the user defined resolution function is called whenever a change of the user defined net happens. If the modeled design does not only consist of point to point connections, then the return value of the called resolution function will trigger other nodes resolution functions. These calls will happen initially because of the changed inputs and will continue until a stable condition of the calculated system is reached. If now the numeric value of the stable condition is not covered in the real value range, the user defined function will return a real value which differs from the exact numeric value. This error will now propagate through the resolution functions and can cause that they will be called over and over again. As a result the simulation get stuck in an infinite loop.
The solution we use to overcome this effect is to convert the real value to a discrete value and reduce the accuracy. Below a code example

```
res_electrical = result;
`ifdef TRUNC
     res_electrical.v_eq = $bitstoreal(
         ($realtobits(res_electrical.v_eq) & 64'hFFFFFFFFFFFFFF00));
`endif
```

Listing 3 Example Avoiding Convergence Problems

Technology Dependent Nettypes

One of the use cases is to detect wrong connection types. One good practice is to catch a problem at compilation time, rather than to catch it at run time to save verification time. This practice is implemented in our implementation through the use of technology specific nettypes. In the last section the base nettype Realnet was shown. Based on this nettype we defined for example the following technology specific nettypes

```
package nettype_l118_uhv_pkg;
   import nettype_pkg::*;

   function automatic voltage_t res_electrical_1v8(
                                        input voltage_t driver[]);
      res_electrical_1v8 = res_electrical(driver);
     endfunction : res_electrical_1v8

   // AMS: electrical_5v0
   function automatic voltage_t res_electrical_5v0(
                                        input voltage_t driver[]);
      res_electrical_5v0 = res_electrical(driver);
   endfunction : res_electrical_5v0
   // ---------------------------------------------------------------
   // Nettype defintions
   // AMS: electrical_1v8
   nettype voltage_t sv_electrical_1v8 with res_electrical_1v8;
   // AMS: electrical_5v0
   nettype voltage_t sv_electrical_5v0 with res_electrical_5v0;
```

Listing 4 Technology Dependent Realnet Definition

The base nettype is imported into this package through the import statement. The example shows that the technology specific nettypes are based on the Realnet data type as well as the Realnet resolution function. If now these technology specific nettypes are used in behavioral models and for example an electrical_1v8 port is connected to an electrical_5v0 port, a compile error will be issued.

One thing that was realized to be helpful to ease the creation of such nettype specializations would be the support of nettypes as parameter for modules. This enhancement request is already discussed in the IEEE standard committee. Until this enhancement is part of the SV standard and supported by the different vendors the structure shown above is a feasible workaround.

5. Using the Nettype

In the past only behavioral Verilog models were used for digital simulation. These behavioral Verilog models will continue to exist for some more time until all tools and models got updated to support the SV nettype. Therefore this section outlines how an existing behavioral Verilog model was updated to support the SV nettype with the goal that the model can also still be used as a behavioral model.

As a general method definition it was decided to use a define to switch between the two use modes of a behavioral model. This allows to place Verilog preprocessor code into one behavioral model used for both use modes. Common code of the model is then automatically shared. The debugging of these macros got much better these days **-** thanks to the heavy use of macros in the UVM base class library.

The define used was defined upfront and was used on all models. One common code added to all modules is

```
`ifdef USE_REALNET
   import nettype_pkg::*;
`endif
```

In the following examples an ADC Verilog behavioral model is updated to System Verilog nettype. The update is done to the base nettype Realnet, but it can also be done to the technology specific implementations.

The ADC Behavioral Model

Port Definition

Whenever a port is used with analog functionality the definition needs to be changed. There was no other clever way found to update the models port definition other than to make the code conditional. Below is an example for this

```
module adc12b9c_5m1t (
…
  vrh0,
  vrh1,
  vrl0,
  vrl1,
…
);
`ifdef USE_REALNET
   inout  realnet vrh0;
   inout  realnet vrh1;
   inout  realnet vrl0;
   inout  realnet vrl1;
`else
   inout  vrh0;
   inout  vrh1;
   inout  vrl0;
   inout  vrl1;
`endif
```

The **underlined** code shows the update required to have the additional definition in the model.

After the definition for the ports it makes sense to add an initial check into the model for 1 to 1 connections. This can be easily done using the load variable of our Realnet structure. Below an example for this

```
1.  initial
2.    begin : load_check
3.      #0;
4.      if (vrh0.load > MAXLOAD)
5.        $display("ERROR, vrh0 load > MAXLOAD");
6.    end : load_check
```

Line 3 is required to allow the simulator to call the resolution function. As a side effect, the load value can also be used nicely in the wave display to check the number of drivers on a net.

Real Numbers

The updated ADC model already had real numbers, used to define the voltage that was sampled. Below is an example of the code fragments that were updated for the high reference voltage. The update does extend the model such that the reference voltage is now taken from the module ports rather than a fixed value or a given define.

```
module adc12b9c_5m1t (
…
  vrh0,
  vrh1,
  vrl0,
  vrl1,
…
);

`ifdef USE_REALNET
   inout  realnet vrh0;
   inout  realnet vrh1;
   inout  realnet vrl0;
   inout  realnet vrl1;
   realnet vrh;
   `define vrefh_val  vrh.v_eq
`else
   inout  vrh0;
   inout  vrh1;
   inout  vrl0;
   inout  vrl1;
   wire vrh;
   real       vrefh_val;
   `define vrefh_val  vrefh_val
`endif

assign vrh = adc_vrh_sel_scan ? vrh0 : vrh1;
…
always @(posedge sar_az1_scan)
  vin_analog_val= vin_analog_val + `vrefh_val/256.0;
`endif
```

Listing 5 ADC Realnet Updates

The updated code to introduce the updated functionality is shown in **underlined**. Especially the use of the define for the vrefh_val has shown to keep the code changes minimal. The update was just to add the " ` " before the variable instead of having the whole code line being duplicated with ifdef around them.

Verilog Primitives

Some models do use Verilog primitives to model the behavior. To be able to keep the current model in the plain digital function almost unchanged, Realnet versions of those models where created.

```
module adc12b9c_5m1t (
// outputs
  adc_temp_sense_out,
…
);
inout  adc_temp_sense_out;
wire adc_temp_sense_out;
real adc_temp_sense_out_val;

`ifdef USE_REALNET
  realnet adc_temp_sense_out_r;
  assign  adc_temp_sense_out_r = '{adc_temp_sense_out_val,1000,1};
  realnet_nmos nmos_temp_sense_out (
       adc_temp_sense_out, adc_temp_sense_out_r, 1'b1);
`else
  nmos(adc_temp_sense_out, adc_temp_sense_out_i, 1'b1);
`endif
…
// temp_sense ------------------------------------------
initial  adc_temp_sense_out_i = 1'b0;
initial  adc_temp_sense_out_val = 0.0;
always @(pmp_en_scan)
begin
  if (pmp_en_scan)
  begin
    adc_temp_sense_out_val = 1.4;
    adc_temp_sense_out_i = 1'b1;
  end
  else
  begin
    adc_temp_sense_out_val = 0.0;
    adc_temp_sense_out_i = 1'b0;
  end
end
```

In the example above the code that needed to be changed is shown in **underlined**.
The Verilog primitive is defined to be

```
pmos p1 (out, data, control);
```

The implementation of the primitive is quite simple and shown for completeness below

```
module realnet_nmos
    (out,
     data,
     control);

    import nettype_pkg::*;

    input  wire    control;
    input  realnet data;
    output realnet out;

    assign out         = control ? data : '{0,`realnetZState,1};

endmodule : realnet_nmos
```

Listing 6 Nmos Primitive for Realnet

The Pad Behavioral Model

Pads are a very complex part in a mixed signal SoC: They need to transfer digital as well as analog signals and therefore contain

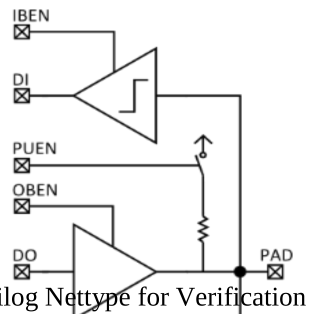

Figure 3 Simple Pad Model

driver, receiver, pull-up/down resistance, transmission gates and so on. And since on an MCU many functions are mapped to one pad, selectable by the user, it is necessary to have one behavior which is able to handle analog as well as digital signals. Figure 3 shows a schematic of a typical pad, including the previously mentioned elements.
Since we like to transfer analog values it is clear that the port "pad" as well as port "analog" are from type Realnet. The direction of the signal flow for both is unknown and thus they are inouts. The output of the buffer is an output logic while the input of the driver and the other configuration ports are input logic.

Digital Connection

The implementation of the digital parts is pretty straightforward. First the output buffer is explained: If the output buffer is enabled, the digital value is converted to the corresponding Realnettype, otherwise the output is high impedance.

```
assign pad = oben ? (do===1'bz ? '{gnd,`realnetZState, 1.0} :
                                (do===1'bx ? '{gnd,`realnetXState, 1.0} :'{do*vdd,300.0, 2.0})) :
                                             '{gnd,`realnetZState, 3.0};
```

The input buffer is realized as 1bit ADC. If the buffer is enabled, and the voltage above/under a certain voltage, then a 1/0 is driven into the SoC. High ohmic/undefined state is considered as well as shown in Listing 7.

```
always @*  begin
    if(iben) begin
        if((analogIn.r_eq - `realnetZState) < 1.0)
            di_reg = 1'bz;
        else if((analogIn.r_eq - `realnetXState) < 1.0)
            di_reg = 1'bx;
        else if(analogIn.v_eq < Vth)
            di_reg = 1'b0;
        else
            di_reg = 1'b1;
    end
    else begin
        di_reg = 1'bz;
    end     //if
end //always
```

Listing 7 Input Comparator

The implementation of the pull up is trivial:

```
assign pad = puen ? '{vdd, 5000.0,1.0} : '{gnd, `realnetZState,1.0};
```

Analog Connection

The analog port is connected to the pad via a transmission gate, thus it is a real bidirectional connection with an on/off resistance between the two ports. Following two implementations are shown and their advantages/disadvantages are explained.
The first implementation is the "self-assignment" as shown in Listing 8. Both Realnets referring to each other over transmission gates with on resistance of 300 Ohm. If the transmission gate is disabled it is high ohmic. The advantage of this implementation is that it is a real bidirectional connection. The disadvantage is the longer simulation time since this solved by Iteration and not,

like in an analog simulator, with differential equations. How big the performance impact is, is not analyzed yet and also depends on the accuracy, see chapter 4 - The Real Convergence Problem.

```
assign pad =  anaen ? '{ana.v_eq,300.0,0.0} : '{gnd,`realnetZState,0.0};
assign ana =  anaen ? '{pad.v_eq,300.0,0.0} : '{gnd,`realnetZState,0.0};
```

Listing 8 Nettype Self Assignment

An alternative implementation is shown in Listing 9. Here a "direction detector" is implemented and depending on the direction the assignment is done. The direction detector is within the always block. First, a little time is consumed to let the simulator converge. Now the voltages at the pad and ana port should be stable. Next, it is checked on which side a driver is connected, indicated by r_eq unequal high ohmic, and on which side not, r_eq equal high ohmic. Depending on this information and the analog enable signal the assignment is done. This implementation is faster since only one iteration is done but it is no real bidirectional connection and more error prone. For instance if the pull up is activated and a signal is driven into the ana port the direction detection will fail.

```
   always @(pad.v_eq,pad.r_eq,ana.v_eq,ana.r_eq)
      begin
             #0; //consume time to let the simulator step and converge
             if (pad.r_eq == `realnetZState &&
                 ana.r_eq != `realnetZState) begin
               ana_drv = 1;
               pad_drv=0;
             end
             else if (ana.r_eq == `realnetZState &&
                      pad.r_eq != `realnetZState ) begin
               pad_drv = 1;
               ana_drv=0;
             end
 else begin
 //error: direction not clear
 end
      end // always @ (p,n)

    assign   pad = (anaen & ana_drv) ?
                    ana :  '{0.0,`realnetZState,0.0};
    assign   ana = (anaen & pad_drv) ?
                    pad :  '{0.0,`realnetZState,0.0};
```

Listing 9 Assignment using Direction Detection

Interface to Digital Signal

Sometimes in a behavioral model a former digital value is used to determine the behavior of the model. This will cause a compile error as user defined nettypes cannot be used in equations with built in nettypes. This is addressed by adding converter modules. A use case for such a converter module usage is shown below

```
1.  nmos            (pad_i, ipp_do_i, ipp_obe_xxi);
2.  bufif1 (weak0,weak1)   (pad_i, ipp_pus, ipp_pue);

3.  `ifdef USE_REALNET
4.  wire pad_pad_i;
5.  nmos            (pad_pad_i, pad_i, 1'b1);
6.  wire_to_realnet w2r_pad_i(.in_w (pad_pad_i),
7.                            .out_r(pad));
8.  `else
9.  nmos            (pad, pad_i, 1'b1);
10. `endif
```

The function of this model uses the nmos primitive at line 5, the pad however is a Realnet signal. For this reason the converter module is added at line 6. This converter module needs to consider the Verilog drive strength model in order to not break the modeling done based on Verilog Primitives.
The converter module is inserted in line 6 and converts a wire to a Realnet. A simplified implementation of this converter module is shown below

```
module wire_to_realnet
#(LOW = 0, HIGH = 5, RES = 100)
(in_w, out_r);
…
  input  wire     in_w;
  output realnet out_r;
  real    v_driver = LOW;
  real    r_driver = `realnetXState;
  assign out_r = '{v_driver,r_driver,1.0};
  always@(in_w) begin
  if(in_w === 1'b1) begin
    v_driver = HIGH;
    r_driver = RES;
  end else if(in_w=== 1'b0) begin
    v_driver = LOW;
    r_driver = RES;
  end else if(in_w=== 1'bz) begin
…
end
```

6. Testbench Structure

The testbench structure is important to be considered when moving to a SV nettype based verification. This is often missed as the analog behavior is mainly a function carried out in the design. However, to verify the design, it is important to get the values passed into and monitored and checked by the testbench. Similar to the design behavioral model updates, the testbench was updated to allow nettype simulations as well a plain 4 state stimulations.

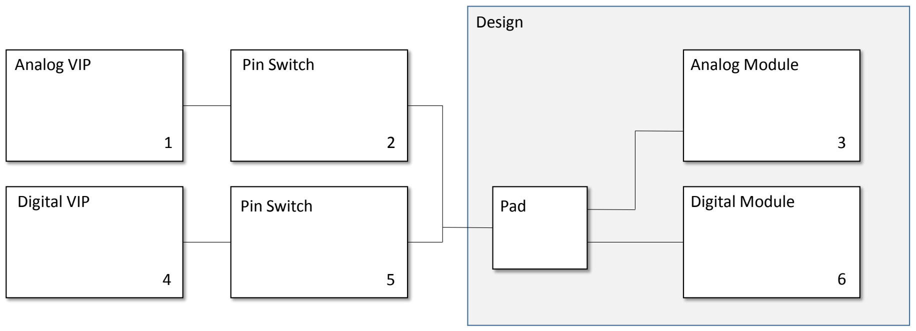

Figure 4 Testbench Structure

Figure 4 shows the testbench structure. On the design side one pad is connected to an analog module as well as digital module (3, 6). On the testbench side the setup is duplicated with an analog and a digital verification IP (1, 4) connected to the design via the pin switch modules (2, 5). Since there is no way to handle pads with only digital function differently than pads that have also analog function, it was decided to have the connection between the testbench pin switch module (2, 5) and the pad behavioral models of the design defined as SV nettype connections. The updated connection required the pin switch to be updated to translate SV nettype to the standard 4 state values required for the digital VIP (4). For the analog VIP (1) a new pin switch was created to support the routing of nettype values.

7. Simulator Invocation

In order to use the SV nettype function the following VCS switch has to be used

```
-sverilog,-sv_interconnect
```

One big requirement is that there shall be no RTL design changes required in order to use the SV nettypes. This makes it mandatory to use the VCS switch

```
-xlrm coerce_nettype
```

This switch analyzes the design for SV nettypes and replaces internally all SV wire declarations with SV interconnect declarations.
To enable the SV nettype function within our behavioral models, the following define needs to be provided

```
+define+USE_REALNET
```

8. Summary

	Load Value	Specific nettype	Real Value Modeling
Wrong connectivity		X	X

Broken connectivity	X		X
1to n connection	X		X
N to 1 connection	X		X

Table 1 Verification Use Case Coverage

The table above summarizes the different methods shown in the document and what kind of verification use cases can be covered by them.
The load value of the `voltage_t` type definition can be used as a simple checking mechanism, but has its limitation in the checking quality.
The technology specific nettypes shown in the section "Technology Dependent Nettypes" are a simple extension of the base nettype Realnet and provide a good compile time check.
Finally the usage of real value models along with the connection of them using the net SV nettypes provide the best checking results but also need the most modeling effort. However the advantage is that there are already some behavioral models that use the real values to model their behave and then it is just a matter of extending these models to make the real values available through the new SV nettype ports. How such a model update can be done is shown in section "Using the Nettype".

9. Conclusions

The usage of the SV nettype allows to verify analog integration behaviour in a single digital simulator engine. We have been able to update several behavioral models and simulate them at module level. On SoC the update did take a bit longer as we had to update 24 modules and faced some tool issues. This proved the known best practice – divide and conquer. For example it may have accelerated the conversion process if the clock and reset module would have been updated on the module level, similar to the ADC. For this reason we did not do any benchmarks so far but there are benchmark results available – see [5].
The tool issue we faced were

- Difficult debugging support for modeling problems, e.g. Realnet connected to a register
- Some features of our testbench structure are not yet supported, e.g. two dimensional array of nettype

We work closely with SNPS R&D to resolve the issues and get very good support.
It needs to be noted that SV nettype are not natively supported together with power definition languages as IEE1801-2013. IEE1801-2013 works on one special structure called **`supply_net_type`** but this is not a SV nettype, it is just a **`struct`**. Beside the supply nets, the normal digital nets are assumed to be built in nettypes, but they also can be user defined nettypes and therefore IEE1801-2013 needs an enhancement to handle them.

10. Acknowledgements

Thanks to Peter Broderick for reviewing the content of the paper as native English speaker.

11.References

[1] IEEE Standards Association, IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language, IEEE Std 1800™-2012, New York, Feb 2013.

[2] C. Chang, "System Modeling and Verification," Carinthia University of Applied Science,

Villach, 2012.

[3] S. Liao and M. Horowitz, “A Verilog Piecewise-Linear Analog Behavior Model for Mixed-Signal Validation,” IEEE, 2013.

[4] M. Schubert, "Mixed Analog-Digital Signal Modeling Using Event-Driven VHDL," SBCCI, Gramado, 1997.

[5] P. Hardee, “Real datatypes and tools enable fast mixed-signal simulation,” [Online]. Available: http://www.techdesignforums.com/practice/technique/real-wreal-datatypes-fast-mixed-signal-simulation/.