Novel Mixed Signal Verification Methodology Using Complex UDNs

Rakesh Dama, Ravi Reddy, Andy Vitek
Roche Sequencing Solutions
Outline

• Introduction
• Modeling Analog Nets using UDNs
• Defining a Generic UDN
• Modeling of SAR-ADC
• Power Aware Verification
• Q&A
Introduction

• Mixed signal designs are increasing by number and complexity

• Modeling approach for analog design is determined by
  – Accuracy (SPICE > Verilog-AMS > SV-RNM)
  – Simulation perf (SV-RNM >> Verilog-AMS > SPICE)
  – Effort (SV-RNM > Verilog-AMS > SPICE)

• SV-RNM is ideal for SOCs due to:
  – Simulation performance (approx. 300x Spice, 100x Verilog-AMS)
  – Portability of models
  – Acceptable accuracy and effort
SV-RNM and UDN

- Simple ‘real’ nets
  - Single Driver nets
  - Straight forward and simple

- User Defined Nettype (UDN)
  - Multiple Driver nets
  - Multiple values can be passed
    (such as Voltage, Current, Impedance)
  - A User Defined Resolution function
    computes the resultant value

From: How To’s of Advanced Mixed Signal Verif, DVCon 2015
Declaring UDN with Resolution

```verilog
nettype Type my_udn with my_udr; // complex UDN (struct type UDT)

typedef struct {
    real voltage;
    real current;
} Type; // UDT

function automatic Type my_udr(input Type drivers[]);
    int n_drivers = 1;
    my_udr.voltage = 0;
    my_udr.current = 0;

    foreach (drivers[ii]) begin
        my_udr.voltage += drivers[ii].voltage;
        my_udr.current += drivers[ii].current; // sum
        n_drivers++;
    end

    my_udr.voltage = my_udr.voltage/n_drivers; // avg
Endfunction // UDR

SV has restrictions on the data-types that can be used in a UDT.

A UDN is unique and connects to only its type (same UDT and UDR)
**UDN Usage**

```vlsm
nettype Type my_udn with my_udr; // complex UDN (struct type UDT)
typedef struct { real voltage; real current;} Type; // UDT

module receiver (input VIN);
   my_udn VIN;
   real power;
   // model
   always @(*) begin
      power = VIN.voltage*VIN.current;
   end
endmodule

module driver1 (output VOUT);
   my_udn VOUT;
   // model
   assign VOUT = Type'{volts_t, amps_t};
   ....
endmodule

Any change in the drivers (in I1 and I2) of the UDN (my_udn) will trigger the UDR function (my_udr) and the resolved value is computed and made available to the modules.
Motivation for a Generic UDN

• In an analog circuit
  – A net has different types of drivers (voltage, current, capacitive, switches...)
  – The same net has to be resolved in different ways at run time

• UDN nets must connect to ‘same type’ ports – i.e. same UDT + UDR
• No dynamic selection of UDR allowed

• Should multiple UDN types be declared?
• Even so, how to connect port of different types?

• While EE_net (Cadence) implements a generic electric equivalent, it cannot handle capacitance and more – how to implement ideal switch?
Concept of a Generic UDN

- A generic UDN that can replace all other UDNs across the design
  - Only a single UDT and a single UDR function
  - Can be used to connect any set of ports in a design
  - Can resolve different scenarios at run time
  - Provides extensibility for new use cases (when the need arises)
  - Ability to compute resultant currents and voltages (like EE_net)
  - Offers modeling efficiency by standardizing and abstracting out UDR
Generic UDN – UDT and UDR

- Analog port types are defined
  - V_OUT, I_OUT > V_PT, I_PT > V_CAP > V_IN, I_IN
  - UDT has a field, `port_type`, to assign port type for a port
- A UDR function is written to resolve based on port types of all ports connected to the net.
  - The dominant port type is resolved, i.e. strongest driver
  - Based on the resolved `port_type`, values for V and I are resolved
  - When the strongest driver is `Z`, then the next dominant driver determines the resolved values

```c
// UDT for generic UDN
typedef struct {
    // V_OUT, V_PT, V_CAP...
    PORT_TYPE port_type;

    // Driven values on ports
    real v_value;
    real i_value;
    real g_value;
    real c_value;

    // vars for active drivers
    real v_active;
    real i_active;
    real g_active;
    real c_active;
} GENERIC_PORT;
```
// Resolution function for generic UDN
function automatic GENERIC_PORT GEN_RES (input drv[]);
  // code to resolve using port_types
  // ........
endfunction

// Generic UDN for generic UDN
nettype GENERIC_PORT GENPORT with GEN_RES;

// Module definitions
module V (output GENPORT v_src);
  real v_val;
  assign v_src = { V_OUT, v_val, `Z, `Z, ....};
  // port_type, volts, amps, impedance, ...

  always
  #1 v_val = 0.5*sin(2*PI*100*time);
endmodule

// TB
module TB ();
  wire net1; //net coerced to GENPORT
  wire net2; //net coerced to GENPORT
  V  v1  (net1);
  SW sw1 (net2, net1);
  C  c1  (net2);
  R  r1  (net2);
endmodule
Modeling a SAR ADC
**Simulation Results of SAR ADC**

- Vcommon node is the capacitive node where charge re-distribution happens
- Conversion starts 2-clk cycles after the falling edge of ‘Start’ signal and takes 4-clks for data_out
- Capacitors are connected to Vref in a sequence (MSB to LSB), this changes the Vcommon volts
- Digital logic determines if capacitor stays connected to Vref or Gnd
Comparison with SPICE

<table>
<thead>
<tr>
<th></th>
<th>RNM</th>
<th>Spice</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNR (dB)</td>
<td>20.05</td>
<td>18.16</td>
</tr>
<tr>
<td>THD (%)</td>
<td>10.9</td>
<td>10.2</td>
</tr>
<tr>
<td>SINAD</td>
<td>16.63</td>
<td>15.89</td>
</tr>
<tr>
<td>ENOB</td>
<td>3.03</td>
<td>2.72</td>
</tr>
<tr>
<td>Sim Time (seconds)</td>
<td>4.7</td>
<td>1183.3</td>
</tr>
</tbody>
</table>

Sim Performance improvement of 250x over SPICE
Bug

- Switch polarity reversal for MSB cap
  - Due to schematic error, the Vref and GND connections for MSB were interchanged
  - Resulting in 0.5*Vref or 8-LSBs of error
  - Lowering model boundary can expose this bug.

Model Boundary without the ability to model capacitors.
Bug Exposed

- Switch polarity reversal for MSB cap
  - Resulting in 0.5*Vref or 8-LSBs of error
- Lowering model boundary enabled easy detection of the bug
Assumptions and Limitations

• To simplify, all active source are ideal (zero output resistance)
  – Non-ideal behavior has to be implemented in the source driver model
• V_OUT and I_OUT connections aren’t allowed

• Implementation done in Cadence Incisive simulator only
• For efficiency reasons, generic UDN to be used to augment simple UDN
• Certain port combinations are yet to be implemented
  – Series capacitance, chaining of two-way switches (V_PT, I_PT ports)
• Debug of generic UDR is tricky due to limited debug hooks
Future Enhancement - Power Aware Ports

- Adding attributes to the UDT – PD_name and PD_state
- Power Domain information is available at run time
- Checkers inside the models utilize this PD info to verify power intent

```
PD = V_LOW
PD_state = OFF

vdd'
// Check vdd supply domain
assert (vdd.PD == V_LOW)
  else `PA_err("Domain err");

// Drive power state info
out_a.PD = PD;

{V_LOW, OFF,...}
Module A

Module B

PD = V_HIGH
PD_state = ON

in_a
// Checker to verify the pin 'in_a'
always @(*) begin
  // LSF check
  if ($volts(PD) != $volts(in_a.PD))
    `PA_err("Missing LSF");
  // Iso check
  if (in_a.PD_state == OFF
    && in_a.v_value != 0 ) // iso-0
    `PA_err("ISO failure");
  // Iso needed - log information
  if ($less_on(PD, in_a.PD)
    `PA_log("Iso needed @%m.in_a"); end
```
Questions?
Thanks!
Additional – Bi-directional Switch

- Presents the resolved resistance, capacitance, current and voltage of all external sources from one port to the other port using V_PT ports
- Accurate when connecting two current branches
A switch connecting two current carrying branches is tricky to implement to reflect true currents on either side.