Real Portable Models for System/Verilog/A/AMS

Bill Ellersick

Analog Circuit Works ™, Inc.

www.analogcircuitworks.com

ABSTRACT

A standards-based modelling and simulation methodology for Systems-on-Chips (SoCs) is presented that is portable and efficient. The real-value discrete-time Verilog behavioral models of mixed-signal circuits simulate accurately and efficiently. To enable model portability across variants of the Verilog language, a set of `define *macros are presented. A Verilog-A testbench verifies both the model and the transistor-level design to ensure correspondence. Use of the methodology in this paper increases the choice of mixed-signal circuits and EDA tools for SoC designers, while expanding the markets of circuit and EDA tool providers and improving semiconductor industry efficiency.*

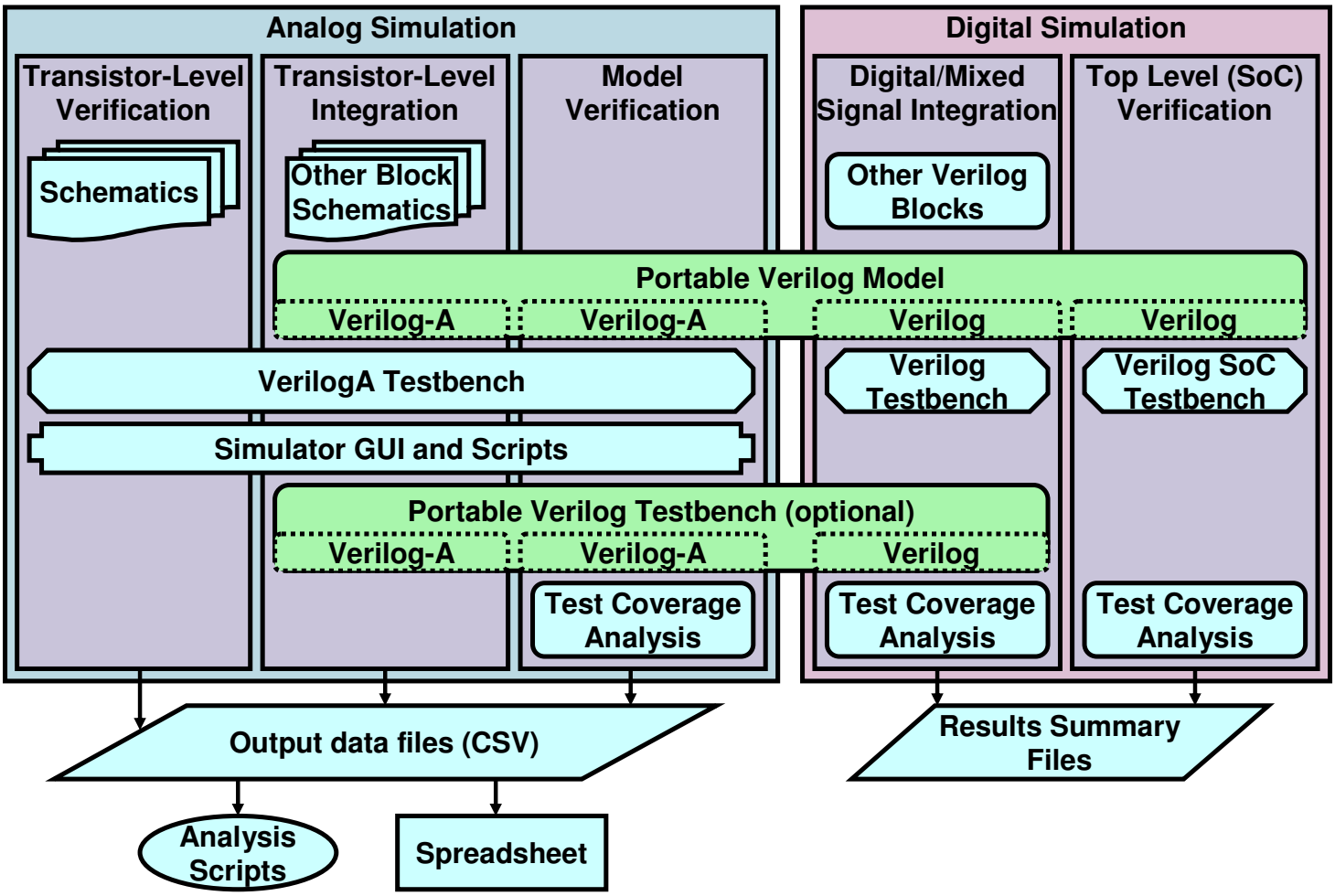
Table of Contents

Table of Figures

1. Introduction

As mixed-signal development tools mature and more widely support standards, the design of Systems-on-Chips (SoCs) can realize some important benefits. The methodology presented here is portable across semiconductor processes and tools and allows rapid behavioral simulation, while supporting comprehensive verification of models against full-transistor simulations. Figure 1 illustrates a proposed verification flow, showing how standards-based models, testbenches and scripts can be used to enhance portability, productivity and reuse.

Figure 1 – Real Behavioral Model and Verification Flow



With standards-based methodology, mixed signal designs can be developed with one set of tools and integrated by another user with a different set of tools. This benefits consumers of mixed signals designs by increasing the number of suppliers they can use, and benefits mixed signal designers and tool providers by enlarging their markets. The broad support of standards by modern tool suites makes them an excellent environment in which to develop and integrate mixed signal circuitry.

This paper presents a methodology for the development, verification and integration of mixed signal circuits that achieves valuable benefits. Chapter 2 describes and provides examples of

Verilog behavioral models with internal real variables. Chapter 3 presents Verilog-A testbenches for transistor-level circuit designs that are also used to verify the behavioral models. Chapter 4 presents simulation results and comparison of execution time of models, and Chapter 5 concludes the paper.

2. Real Behavioral Models in Verilog

Real variables can model analog currents and voltages in discrete time with accuracy approaching that of analog simulators[1][2]. In fact, analog simulators only evaluate at discrete time steps as well, so the difference between the two approaches is less than it appears. The largest difference may be in how designs are described to the simulators. Discrete-time real models use difference equations, while analog representations use differential equations that require slow iterative solving. Contrast the discrete-time model for a voltage source driving an RC filter:

Figure 2 – Voltage Source Driving RC Filter

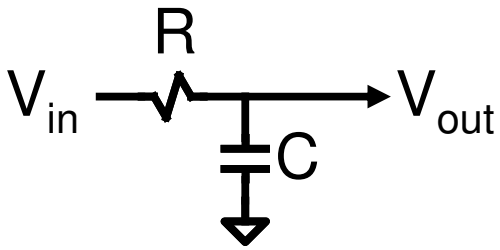


Figure 3 – Difference vs. Differential Equations for Voltage Source Driving RC Filter

Difference Equations	Differential Equations
$I_r = (V_{in} - V_{out}) / R$	$I_r = (V_{in} - V_{out}) / R$
$\Delta V_{out} = \Delta t\ I_r / C$	$dV = dt\ I_r / C$

As long as Δt is a small enough fraction of the signal frequency, discrete-time difference equations can be as accurate as periodically solved differential equations. And since the difference equations don't require iterative convergence and need only be evaluated when their inputs have changed, they can execute orders of magnitude faster than differential equations. It turns out that Verilog is a great language in which to express discrete-time difference equations, with the above represented by:

Figure 4 – Verilog for Voltage Source Driving RC Filter

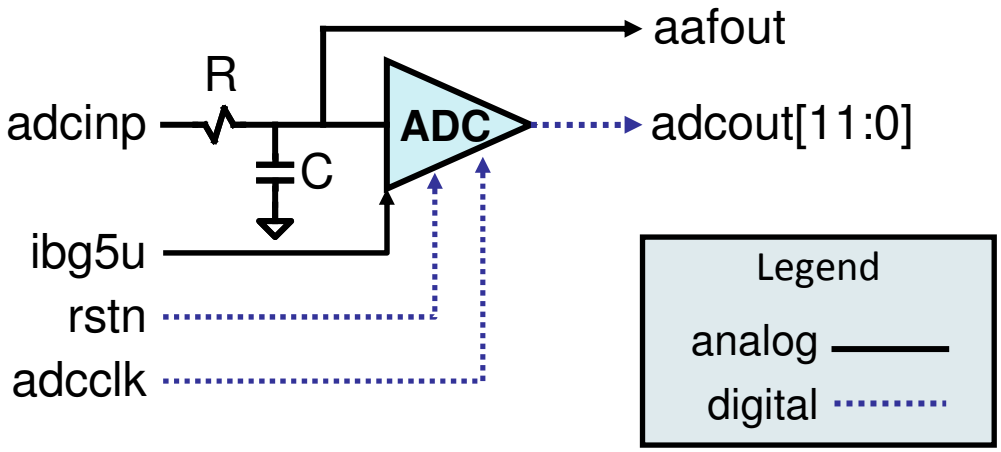
```
input Vin;    // input voltage
real Ir;      // resistor current
real Vout;    // output voltage
real lsttim;  // last time model was evaluated

initial begin
  Vout = 0.0;
  lsttim = 0.0;
end

Ir = (Vin - Vout) / C;                        // resistor model
Vout = Vout + Ir * ($abstime - lsttim) / C;   // cap model
lsttim = $realtime;                           // save time
```

Note that dV/dt becomes $\Delta V_{out}/\Delta t$. Similarly, $\int Idt$ would become $I\Delta t$. Thus, inductors and even parasitic coupling aren't much more difficult to model with difference equations than resistors and capacitors, and even large circuits are guaranteed to converge in finite time.

Recent standardization efforts to support discrete-time real models have resulted in the ability of most SystemVerilog, Verilog and Verilog-AMS simulators to transfer real values on input and output ports. While this falls short of true electrical modeling of both the current and voltage on each port, it is of great practical use, since most ports transfer either a voltage or a current*. No longer must mixed-signal designers carefully review top level modules to ensure that analog ports are properly connected – these connections can be simulated and the simulations checked with verification coverage tools.

To illustrate further, a more comprehensive example comprises an anti-alias filter at the input of an ADC, with the anti-alias filter voltage output for debugging purposes. This example includes an analog voltage input and output, a current input, logic and clock inputs and a logic vector output, as shown in the figure below.

Figure 5 – adcX Example Block Diagram: Anti-alias Filter into ADC



The discrete-time Verilog model for this ADC example, dubbed adcX†, is shown below:

Figure 6 – Verilog Model for adcX

```
module adcX(adcout,aafout,ibg5u,adcinp,adcclk,rstn);

parameter real Inom = (-5e-6);  // nominal input reference current
parameter real Vref = 1.0;      // nominal reference V
parameter real Voff = 0.8;      // input offset voltage
parameter real Raaf = 1e3;      // anti-alias filter resistance
parameter real Caaf = 1e-12;    // anti-alias filter capacitance

output [11:0] adcout;           // adc output value
output aafout;                  // anti-alias filter output (sim. only)
input ibg5u;                    // bandgap current reference
input adcinp;                   // adc input voltage
input adcclk;                   // adc clock
```

* There are ways to surreptitiously transfer additional real values between modules if necessary

† See www.analogcircuitworks.com for the latest adcX model with testbench and include files

```
input rstn;                     // asynchronous reset (active low)

wire [11:0] adcout;
real aafout;
wire real ibg5u;
wire real adcinp;
wire adcclk;
wire rstn;

real refv;                      // adc reference voltage
real refv0;                     // adc initial reference voltage
real vadc;                      // sampled, modified input voltage
real ires;                      // anti-alias resistor current
real lsttim;                    // last time anti-alias filter eval'd
integer adcout_int[11:0];       // internal ADC output register
real aafout_v;                  // anti-alias filter voltage

genvar i1;                      // index for generate statements
integer i;
integer rstn_int;

                                // body of model
  ////////////////////// initialize local variables ////////////////////
  initial begin
    aafout_v = 0.0;
    lsttim = 0.0;
  end

  ////////////////////////// anti-alias RC filter /////////////////////////
  initial #1e-10 forever begin     // evaluate ~10 times highest freq
    ires = (adcinp - aafout_v) / Raaf;      // resistor current
                // capacitor difference equation C = V + I * deltaT / C
    aafout_v = aafout_v + ires * ($realtime-lsttim) / Caaf;
    lsttim = $realtime;                          // save time
  #1e-10; end

  ////////////////////// ADC example core model //////////////////////
  always @(posedge adcclk or negedge rstn) begin
    if(!(rstn)) begin
      for(i=11; i>0; i=i-1) int[i] = 0;
      $display("adcX reset at %10.4g\n",$realtime);
    end else begin
      refv0 = 0.5 * ibg5u/5e-6;
      refv = refv0;                 // start with full reference voltage
      vadc = adcinp-Voff+refv0;     // sample input, subtract Voff-refv0
      for(i=11; i>0; i=i-1) begin
        if(vadc > refv) begin
          adcout_int[i] = 1;        // output logic high if vadc > refv
          vadc = vadc - refv;       // and subtract refv from vadc
        end else begin
          adcout_int[i] = 0;        // else output logic low
        end
        refv = refv / 2.0;          // halve refv for next bit
      end
    end
  end

  ////////////////////////// assign outputs ////////////////////////////
  assign adcout[0] = adcout_int[0];
  assign adcout[1] = adcout_int[1];
  assign adcout[2] = adcout_int[2];
  assign adcout[3] = adcout_int[3];
  assign adcout[4] = adcout_int[4];
```

```
  assign adcout[5] = adcout_int[5];
  assign adcout[6] = adcout_int[6];
  assign adcout[7] = adcout_int[7];
  assign adcout[8] = adcout_int[8];
  assign adcout[9] = adcout_int[9];
  assign adcout[10] = adcout_int[10];
  assign adcout[11] = adcout_int[11];
  always @( aafout_v) aafout =  aafout_v;

endmodule
```

3. Verilog-A Behavioral Models

The Verilog-A language that is supported by most analog simulators shares some syntax with digital Verilog. And both languages have methods for accomplishing the three basic tasks needed to model mixed-signal circuitry, albeit with frustratingly different syntax. They have mechanisms for coupling analog and digital values in and out of modules, for specifying when to update values, and for specifying equations to updates the values. Before considering how to create a portable model, let's look at the Verilog-A code that is equivalent to the above Verilog model for adcX:

Figure 7 – Verilog-A Model for adcX

```
module adcX(adcout,aafout,ibg5u,adcinp,adcclk,rstn);

parameter real Inom = (-5e-6);  // nominal input reference current
parameter real Vref = 1.0;      // nominal reference V
parameter real Voff = 0.8;      // input offset voltage
parameter real Raaf = 1e3;      // anti-alias filter resistance
parameter real Caaf = 1e-12;    // anti-alias filter capacitance

output [11:0] adcout;           // adc output value
output aafout;                  // anti-alias filter output (sim. only)
input ibg5u;                    // bandgap current reference
input adcinp;                   // adc input voltage
input adcclk;                   // adc clock
input rstn;                     // asynchronous reset (active low)

electrical [11:0] adcout;
electrical aafout;
electrical ibg5u;
electrical adcinp;
electrical adcclk;
electrical rstn;

real refv;                      // adc reference voltage
real refv0;                     // adc initial reference voltage
real vadc;                      // sampled, modified input voltage
real ires;                      // anti-alias resistor current
real lsttim;                    // last time anti-alias filter eval'd
integer adcout_int[11:0];       // internal ADC output register
real aafout_v;                  // anti-alias filter voltage

integer i1;                     // index for generate statements
integer i;
integer rstn_int;

analog begin                    // body of model
  /////////////////////// initialize local variables ///////////////////////
  @(initial_step) begin
    aafout_v = 0.0;
```

```
    lsttim = 0.0;
  end

  ////////////////////// anti-alias RC filter ////////////////////////
  @(timer(1e-10,1e-10)) begin        // evaluate ~10 times highest freq
    ires = (V(adcinp) - aafout_v) / Raaf;      // resistor current
               // capacitor difference equation C = V + I * deltaT / C
    aafout_v = aafout_v + ires * ($realtime-lsttim) / Caaf;
    lsttim = $realtime;                          // save time
  end

  ////////////////////// ADC example core model ////////////////////////
   @(cross(V(adcclk)-(vdd_v+vss_v)/2.0, 1)
       or cross(V(rstn)-(vdd_v+vss_v)/2.0, -1)) begin
    if(!(V(rstn)>(vdd_v+vss_v)/2.0) ? 1 : 0)) begin
      generate i(11,0) adcout_int[i] = 0;
      $display("adcX reset at %10.4g\n",$realtime);
    end else begin
      refv0 = 0.5 * I(ibg5u)/5e-6;
      refv = refv0;                // start with full reference voltage
      vadc = V(adcinp)-Voff+refv0; // sample input, subtract Voff-refv0
      generate i(11,0) begin
        if(vadc > refv) begin
          adcout_int[i] = 1;       // output logic high if vadc > refv
          vadc = vadc - refv;      // and subtract refv from vadc
        end else begin
          adcout_int[i] = 0;       // else output logic low
        end
        refv = refv / 2.0;         // halve refv for next bit
      end
    end
  end

  ////////////////////////// assign outputs ////////////////////////////
  adcout[0] <+ transition((adcout_int[0] ? vdd_v : vss_v), 100e-12, 100e-12);
  adcout[1] <+ transition((adcout_int[1] ? vdd_v : vss_v), 100e-12, 100e-12);
  adcout[2] <+ transition((adcout_int[2] ? vdd_v : vss_v), 100e-12, 100e-12);
  adcout[3] <+ transition((adcout_int[3] ? vdd_v : vss_v), 100e-12, 100e-12);
  adcout[4] <+ transition((adcout_int[4] ? vdd_v : vss_v), 100e-12, 100e-12);
  adcout[5] <+ transition((adcout_int[5] ? vdd_v : vss_v), 100e-12, 100e-12);
  adcout[6] <+ transition((adcout_int[6] ? vdd_v : vss_v), 100e-12, 100e-12);
  adcout[7] <+ transition((adcout_int[7] ? vdd_v : vss_v), 100e-12, 100e-12);
  adcout[8] <+ transition((adcout_int[8] ? vdd_v : vss_v), 100e-12, 100e-12);
  adcout[9] <+ transition((adcout_int[9] ? vdd_v : vss_v), 100e-12, 100e-12);
  adcout[10] <+ transition((adcout_int[10] ? vdd_v : vss_v), 100e-12, 100e-12);
  adcout[11] <+ transition((adcout_int[11] ? vdd_v : vss_v), 100e-12, 100e-12);
  V(aafout) <+  aafout_v;
  V(ibg5u) <+ 1.1;              // voltage termination at current input
end

endmodule
```

Note how the bulk of the code is almost identical, with differences in the specification of when to update values, and the coupling of input and output ports.

4. Portable Behavioral Models

While analog-digital simulators continue to be developed, they remain expensive, slow and finicky. Rather than wrestle with a hybrid simulator, consider a model that uses the common Ver-

ilog and Verilog-A features for manipulating real and integer variables. A mixed-signal model that is portable between digital and analog simulators allows a digital model to be verified with an analog simulator, with results compared to the transistor-level design. The model can then confidently be used in digital simulations at break-neck speeds.

While not all Verilog code corresponds to valid Verilog-A code, the examples above include all the essential features of mixed signal models. And by taking advantage of the powerful `define construct, a portable mixed-signal model can be written to compile as either Verilog or Verilog-A code. Ideally, we will work toward standards and simulators that use more common syntax. But in the meantime, we can benefit from the existing similarities and reap the benefits of portable models. Since SystemVerilog is a superset of Verilog, and Verilog-AMS is a superset of both Verilog and Verilog-A, the portability techniques in this paper are compatible with those languages as well. The following include file is one method for producing portable Verilog/Verilog-A/Verilog-AMS/SystemVerilog models, and was used to generate the above examples that have been verified on VCS, Spectre, Icarus and NCVlog simulators:

Figure 8 – Include File for Portable Models/Testbenches

```
// verDef.h - defines to allow Verilog variant independent code
//
// Definitions to support portable models/testbenches in System/Verilog/A/AMS
                                        // default simulator: Spectre (Verilog-A)
// `define __VCS__                      // simulator: VCS (System Verilog)
// `define __NCVLOG__                   // simulator: ncvlog
// `define __ICARUS__                   // simulator: Icarus Verilog

`ifdef __VCS__                          // if VCS (System Verilog)
  `define __VlogD__
  `define AnalogInput real
  `define AnalogOutput real
`endif // __VCS__
`ifdef __NCVLOG__
  `define __VlogD__
  `define AnalogInput wreal
  `define AnalogOutput wreal
`endif // __NCVLOG__
`ifdef __ICARUS__
  `define __VlogD__
  `define AnalogInput wire real
  `define AnalogOutput real
`endif // __ICARUS__
`ifdef __VlogD__                // common defines for System/Verilog/AMS
  `timescale 1s/1fs             // timescale 1s to match default in Verilog-A
  `define AnalogBegin
  `define LogicInput wire
  `define LogicOutput wire
  `define Always always
  `define Initial initial
  `define AnalogAssign(sig,val) always @(val) sig = val
  `define LogicAssign(sig,val) assign sig = val
  `define Posedge(sig) posedge sig
  `define Negedge(sig) negedge sig
  `define True(sig) (sig)       // logic signal merely passed as wire in Verilog
  `define AtTimerBegin(dly,period) initial #dly forever begin
  `define TimerEnd(period) #period; end
  `define V(sig) sig            // V(sig) merely passed as real value in Verilog
  `define I(sig) sig            // I(sig) merely passed as real value in Verilog
  `define AnalogTerm(sig,val)   // termination (NOP in Verilog)
```

```
  `define For(i,i1,i2) for(i=i1; (i1<i2?i<=i2:i>=i2); i=(i1<i2?i+1:i-1))
  `define AnalogEnd             // nothing needed in digital Verilog
`else   // ifndef __VlogD__: must be Verilog-A
  `define _Trise 100e-12        // logic output rise time
  `define _Tfall 100e-12        // logic output fall time
  `define _Tdelay 100e-12       // logic output delay
  `define AnalogBegin analog begin
  `define AnalogInput electrical
  `define AnalogOutput electrical
  `define LogicInput electrical
  `define LogicOutput electrical
  `define Always                // nothing required for Verilog-A
  `define Initial @(initial_step)
  `define AnalogAssign(sig,val) sig <+ val
  `define AnalogTerm(sig,val) sig <+ val        // termination (NOP in Verilog)
    // note: must declare and assign values to real vdd_v, vss_v for logic `defines
  `define LogicAssign(sig,val) sig <+ transition((val ? vdd_v : vss_v), `_Tdelay,
`_Trise, `_Tfall)
  `define Posedge(sig) cross(V(sig)-(vdd_v+vss_v)/2.0, 1)
  `define Negedge(sig) cross(V(sig)-(vdd_v+vss_v)/2.0, -1)
  `define True(sig) (V(sig)>(vdd_v+vss_v)/2.0) ? 1 : 0)
  `define AtTimerBegin(dly,period) @(timer(dly,period)) begin
  `define TimerEnd(period) end
  `define V(sig) V(sig)
  `define I(sig) I(sig)
  `define For(i,i1,i2) generate i(i1,i2)
  `define AnalogEnd end
`endif  // ifndef __VlogD__
```

In concert with the above include file, the following portable model code was used with digital Verilog simulators and with analog SPICE-like simulators. Note that the core model manipulates internal real and integer variables, which are then assigned to outputs. This approach limits the use of `define macros to only those statements that communicate directly to the inputs and outputs, or specify when to change variable values.

Figure 9 – Portable Model for adcX

```
module adcX(adcout,aafout,ibg5u,adcinp,adcclk,rstn);

parameter real Inom = (-5e-6);  // nominal input reference current
parameter real Vref = 1.0;      // nominal reference V
parameter real Voff = 0.8;      // input offset voltage
parameter real Raaf = 1e3;      // anti-alias filter resistance
parameter real Caaf = 1e-12;    // anti-alias filter capacitance

output [11:0] adcout;           // adc output value
output aafout;                  // anti-alias filter output (sim. only)
input ibg5u;                    // bandgap current reference
input adcinp;                   // adc input voltage
input adcclk;                   // adc clock
input rstn;                     // asynchronous reset (active low)

`LogicOutput [11:0] adcout;
`AnalogOutput aafout;
`AnalogInput ibg5u;
`AnalogInput adcinp;
`LogicInput adcclk;
`LogicInput rstn;

real refv;                      // adc reference voltage
real refv0;                     // adc initial reference voltage
real vadc;                      // sampled, modified input voltage
```

```
real ires;                      // anti-alias resistor current
real lsttim;                    // last time anti-alias filter eval'd
integer adcout_int[11:0];       // internal ADC output register
real aafout_v;                  // anti-alias filter voltage

`Genvar i1;                     // index for generate statements
integer i;
integer rstn_int;

`AnalogBegin                    // body of model
  ////////////////////////// initialize local variables
//////////////////////////
  `Initial begin
    aafout_v = 0.0;
    lsttim = 0.0;
  end

  ////////////////////// anti-alias RC filter ////////////////////////////////
  `AtTimerBegin(1e-10,1e-10)       // evaluate ~10 times highest freq
    ires = (`V(adcinp) - aafout_v) / Raaf;      // resistor current
                // capacitor difference equation C = V + I * deltaT / C
    aafout_v = aafout_v + ires * ($realtime-lsttim) / Caaf;
    lsttim = $realtime;                         // save time
  `TimerEnd(1e-10)

  ////////////////////// ADC example core model //////////////////////////
  `Always @(`Posedge(adcclk) or `Negedge(rstn)) begin
    if(!`True(rstn)) begin
      `For(i,11,0) adcout_int[i] = 0;
      $display("adcX reset at %10.4g\n",$realtime);
    end else begin
      refv0 = 0.5 * `I(ibg5u)/5e-6;
      refv = refv0;                         // start with full reference voltage
      vadc = `V(adcinp)-Voff+refv0;         // sample input, subtract Voff-refv0
      `For(i,11,0) begin
        if(vadc > refv) begin
          adcout_int[i] = 1;                // output logic high if vadc > refv
          vadc = vadc - refv;               // and subtract refv from vadc
        end else begin
          adcout_int[i] = 0;                // else output logic low
        end
        refv = refv / 2.0;                  // halve refv for next bit
      end
    end
  end

  //////////////////////////// assign outputs ////////////////////////////////
  `LogicAssign(adcout[0],adcout_int[0]);
  `LogicAssign(adcout[1],adcout_int[1]);
  `LogicAssign(adcout[2],adcout_int[2]);
  `LogicAssign(adcout[3],adcout_int[3]);
  `LogicAssign(adcout[4],adcout_int[4]);
  `LogicAssign(adcout[5],adcout_int[5]);
  `LogicAssign(adcout[6],adcout_int[6]);
  `LogicAssign(adcout[7],adcout_int[7]);
  `LogicAssign(adcout[8],adcout_int[8]);
  `LogicAssign(adcout[9],adcout_int[9]);
  `LogicAssign(adcout[10],adcout_int[10]);
  `LogicAssign(adcout[11],adcout_int[11]);
  `AnalogAssign(`V(aafout), aafout_v);
  `AnalogTerm(`V(ibg5u),1.1);               // voltage termination at current
input
`AnalogEnd
```

```
endmodule
```

5. Testbenches and Debugging

With portable models, verification is simplified. A Verilog-A testbench, along with simulator-specific settings and scripts, can verify that a portable model matches a transistor-level design. This allows digital simulation with confidence in mixed-signal models. In addition, the portable model can help analog simulation by allowing accelerated simulation for transistor-level integration. In fact, the testbench can even be made portable, although since the testbench will not likely be used for digital verification without modification, the restricted coding style required for portability may outweigh the benefits.

Debugging portable Verilog can be challenging, as simulators do not always give clear indications of syntax errors, particularly when `define's are used. The `define's in this paper are as straightforward as possible and avoid generating multiple statements to simplify debugging. If error messages are confusing on a `define, a useful technique is to use a run-time option on the simulator to generate the preprocessed Verilog or Verilog-A code, and then compile that code to get more specific error messages.

6. Conclusions

A standards-based methodology for modeling and simulation of mixed-signal circuitry in SoCs is presented that is portable, accurate and efficient. The portable behavioral models represent analog voltages and currents with discrete-time real variables which can be connected through the design hierarchy. Verilog-A testbenches are used to verify transistor-level circuits and ensure that the behavioral models produce the same output and are thus equivalent.

The behavioral models execute accurately and rapidly, allowing comprehensive modelling of mixed-signal circuits in even the largest integrated circuits. Verification of models and transistor-level designs with the same analog testbenches ensures correspondence (comprehensive testbenches are crucial to this), which is critical to succesful SoC design.

The high-level code in the models and testbenches improves productivity and portability between semiconductor processes and tool suites. This portability allows mixed-signal circuit designers to expand their served market, and allows SoC designers to integrate designs from a wide set of circuit providers. Similarly, EDA tool users benefit from a wider choice of simulators, while EDA tool providers can reach a more diverse set of customers. Thanks to the vision and efforts of many, standards-based mixed-signal methodologies are now achievable, and will enable increasing levels of integration and efficiency throughout our industry.

7. References

[1] W. Harstong and S. Cranston, "Real Valued Modeling for Mixed Signal Simulation", Jan. 2009.

[2] K. Kundert, "Verifying All of an SoC - Analog Circuitry Included", Fall 2009 IEEE Solid State Circuits Magazine.

[3] IEEE Standard for Verilog® Hardware Description Language, IEEE Std 1364™-2005, April 7, 2006.

[4] IEEE Standard for SystemVerilog, Unified Hardware Design, Specification and Verification Language, IEEE Std 1800-2009, Dec. 11, 2009.

[5] Verilog-A, Language Reference Manual, Analog Extensions to Verilog HDL, Open Verilog International, Aug 1, 1996.

[6] M. Silverman, “Using Scripting and Verilog AMS to Improve Design and Testbench Reuse”, Ohio State University Masters Thesis, 2006.