Mixed Signal Assertion-Based Verification

Prabal Bhattacharya
Cadence Design Systems, Inc
2655 Seely Ave, B10
San Jose, CA
+1 408 9431234
prabal@cadence.com

Don O'Riordan
Cadence Design Systems, Inc
2655 Seely Ave, B10
San Jose, CA
+1 408 9431234
riordan@cadence.com

Walter Hartong
Cadence Design Systems, Inc
Mozartstrasse 2
D - 85622 Feldkirchen bei München
+ 49.0.89.4563.7770
hartong@cadence.com

ABSTRACT

The increase in mixed-signal content - both in size and complexity – of an SoC demands a change in the existing mixed-signal verification techniques. Although some ad-hoc practices exist today for analog or mixed-signal verification, none of these methods scale to the complex circuit conditions that analog and mixed-signal verification tasks encounter. In general, it is perceived that the verification system for a mixed-signal SoC will need to leverage metric-driven verification techniques that are well established in the digital verification domain for quite some time now and extend them to address the requirements of verifying the analog and mixed-signal content of the SoC.

As part of this thought, this paper introduces the concept of how PSL and SVA - the two most widely used IEEE standards for specifying assertion - can be extended to model an analog/mixed-signal assertion in a natural way without compromising the verification goals. This approach has several long term benefits:

- By extending PSL and SVA to work with mixed-signal languages and semantics, we will be able to leverage the existing rich features that are standard in Assertion Based Verification, such as coverage, in a natural way.
- There is a growing need for analog and mixed-signal verification to be visible to the overall system verification environment. As of now, there is a lack of visibility into the verification tasks specified and covered in the analog or mixed-signal sub-system. By using a consistent language and use model from the pure digital to mixed-signal representation, such visibility and information sharing would be natural.

In discussing the PSL and SVA extensions required to support, we discuss the specifics of how the Boolean layer of the PSL language can incorporate useful aspects of Verilog-AMS to model analog and mixed-signal circuit behavior. We also discuss a smart value access methodology that allows System-Verilog testbenches to be able to use analog quantities such as voltage, current, dissipated power or operating point parameters in System-Verilog Assertions. Finally, we explain these concepts with the aid of a mixed-signal model.

Keywords

Assertions, Verification, ABV, PSL, SVA, VHDL, Verilog, Verilog-A, Verilog-AMS, VHDL-AMS, Mixed-Signal

1. INTRODUCTION

Assertions, by definition, capture the intended behavior of a design. Assertion Based Verification (ABV) is a powerful verification approach which has proven to help digital IC architects, designers, and verification engineers improve design quality and reduce time to market. Assertions are written both during development of the design and the verification environment. Both designers and verification engineers can be involved in identifying requirements and capturing them as assertions.

A designer for a given block enables assertion-based verification of the block by:

- Locating or writing assertions that reflect the properties of the interface between that block and the rest of the design (e.g. the red colored assertions in Figure 1)
- Documenting as assertions any additional assumptions made about the interface as the block is implemented
- Writing assertions about important interactions that are expected to occur among subcomponents of the block
- Writing assertions that prohibit predictable nominal functionality-related, boundary condition-related, startup behavior-related, and other predictable errors
- Creating coverage points to ensure that known corner cases and complex areas of the design are verified

Designers can also verify their blocks using the assertions they have written about its behavior. In particular, designers can use formal analysis to verify that the block behaves correctly. They can also simulate, to test whether the block works correctly in common scenarios.

A verification engineer or design integrator defines assertions and coverage points derived from the functional specification for the device only. For example, a verification engineer might define assertions to ensure that:

- The design is always in a valid configuration
- The design and the environment communicate correctly
- The design responds correctly to its inputs (e.g. the blue colored assertions in Figure 1)

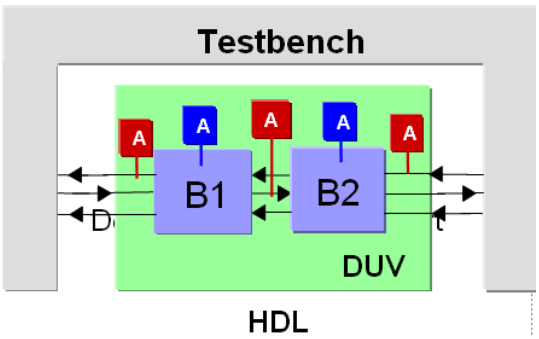


Figure 1 Device under Verification (DUV) with embedded Assertions

A verification engineer will also be concerned about measuring functional coverage, to ensure that the design is thoroughly verified. To that end, the verification engineer will define functional coverage points to check that

- The design has been verified in every valid configuration
- All possible variations in the communication protocol between design and environment have been verified
- All, or at least representative, valid combinations of inputs have been used in the verification
- All, or at least representative, valid combinations of outputs have been observed in the verification

Standard assertion languages such as PSL[2] and SVA[1] have evolved to meet the needs of logic designers and verification engineers in the digital space, and are used both with dynamic (simulation-based and accelerator-based) testing and Formal Verification methods. Such assertion languages provide a formal framework for posing and verifying questions about the design such as those listed in Figure 2.

Question	Property Type
Are there signals that have a set behavior that must occur independent of time?	**Invariants**
Are there signals that have a set of behavior that must occur within a certain time frame?	**Bounded Invariants**
Does the design contain boundary conditions that must trigger a set behavior?	**Boundary Cases**
Are there ways to specify values or sequences that would describe an error condition?	**Bug Identification**
Is the behavior of certain signals critical to the functionality of the design	**Signal Values**

Figure 2 Basic Questions and Property Types.

2. ASSERTION IN ANALOG AND MIXED-SIGNAL SPACE

2.1 New Frontiers in Mixed-Signal Verification

As design complexity is increasing, the verification tasks around an analog or mixed-signal system are becoming more and more difficult to plan and execute. Some of these fundamental difficulties are:

- There is no consistent language and methodology across the complete spectrum of discrete event driven systems, mixed-signal systems and continuous time varying systems to express the verification intent in form of assertions?
- As no such standard methodology exists, how does information, expressed by one group of design or verification engineers – presumably in the analog/mixed-signal domain – flow from one group to another, or from one level of design abstraction to another?
- In absence of a standard language and methodology to apply verification to a mixed-signal system in its entirety, how can a verification plan include testing analog and mixed-signal blocks - that were tested in isolation – in context of the complete system? This challenge includes verifying aspects like power sequences, current leakages, noise figures etc from the analog or mixed-signal blocks, in a full system context.

The availability of formal property specification languages with their well defined set of semantics has benefited the digital design and verification communities for some time, and in view of the challenges mentioned above it is natural to attempt to apply the same or similar concepts to the Analog and Mixed Signal design and verification domains. Indeed, various efforts are now proceeding in that direction, such as those of the Verilog-AMS language committee [6].

2.2 Mixed-Signal Assertions Applications

Applications of Analog/Mixed signal assertions could potentially cover quite a gamut of properties and possibilities, including but certainly not limited to the following:

- Functional properties. Does the circuit/design meet its basic functionality requirements? The ability to specify properties such as this would be needed should formal verification of analog/mixed signal circuits ever become a reality.
- Mixed-Signal properties. Any property where an analog value on one side of the mixed-signal interface should match to a digital code on the other side fits into this category. Examples include an analog to digital converter (ADC) used for measurement purposes, or a digitally calibrated current-DAC (digital to analog converter). These design styles and associated properties arise due to the increasing amount of analog variability on smaller geometry processes leading to the need for digitally calibrated analog circuitry. These can often be subdivided into digital-centric properties and analog centric properties. Examples of digital centric properties include pure existing digital properties, but where the associated clocking/sampling events reference analog quantities (e.g. “clk” is an analog node). Examples of of analog centric properties include those in which the boolean layers reference real-valued variables or signals.
- Digital properties. Even purely digital properties used in typical verification testbenches need to be re-useable when the

design is reconfigured such that portions of it are represented in the continuous domain. In particular, many of these design configurations require swapping out a digital block or sub-block and replacing it with a transistor level counterpart, and/or re-simulating the blocks in the presence of parasitic devices. During these regression tests, it is desirable to reuse the same testbenches in an "as is" manner across both representations, even though some of the signals referenced in the Boolean Layer of Properties are no longer purely digital, but instead now are real valued voltages/currents solved for by the analog kernel in a mixed signal simulator such as Cadence Virtuoso AMS Designer Simulator.

3. EXISTING APPROACHES CONSIDERED FOR ASSERTION IN ANALOG AND MIXED-SIGNAL

In the analog verification domain, the idea of specification, which drives the need for defining assertion, is not a common notion. Nevertheless, analog designers and verification engineers do set custom characterization checks to specify the safe operating conditions for the devices that comprise a circuit. In the Cadence Virtuoso Spectre circuit simulator [3] this is done by adding a special assert device to the circuit and associating a checklimit analysis to verify if the device level conditions specified by the user has indeed been satisfied during the course of a simulation which the checklimit analysis corresponds to.

3.1 A brief tour of Spectre Assert Statement and Checklimit Analysis

With the assert statement, a user can set custom characterization checks to specify the safe operating conditions for the circuit. Spectre then issues messages telling the user when parameters move outside the safe operating area and, conversely, when the parameters return to the safe area, peak value and duration of violations. When a variable changes from an above-max value directly to a below-min value in one simulation step (that is, no stay within bounds), the Spectre simulator uses a middle bound solution (min+max)/2 to report the peak value and the duration of violations.

The four types of checks that are supported in the device checking flow in the Cadence Virtuoso Spectre circuit simulator are described below.

Check	Description
Initial setup check	Includes checks on constant parameters only, such as constant global, model or instance parameters that are independent of the operating points. This check is done only once before any analysis (including checklimit) is run. This check is also repeated once if any constant parameter is altered. **Note:** The initial setup check cannot be disabled and the error level cannot be changed by the checklimit statement.
Operating point check	Includes checks on MDL (Measurement Description Language) expressions and instance operating point parameters. This check is done for each analysis.
Time domain check	Check done during transient analysis.
Frequency Domain Check	Check done during AC analysis

Assert statements, which are specified in the netlist, are supported in Spectre for transient, AC, DC and DC sweep analyses.

Users can enable or disable an assert or a group of asserts with the checklimit statement. One or more checklimit statements can be enabled in the netlist, each enabling or disabling individual asserts. The statement is applied to subsequent transient, DC, and DC sweep analyses until the next checklimit statement appears.

3.2 Using Mixed-Signal Behavioral Languages to Express Assertion Intent

The power rendered by standard mixed-signal languages such as Verilog-AMS or VHDL-AMS are also used today to define checks of various kinds. These checks employ standard language techniques such as macros, disciplines etc to define a set of common expected circuit behavior and then can set or unset error flags when in the course of simulation a model goes out of or comes back within the specified operating range for that model. A snippet of such a set of macros is as follows

```
// Generic Verilog-AMS assertion macros:
//  `ACHECK(Val,Max,Min,Desc,Name,Flg,En,Td,Vtol)
analog value check
//  `DCHECK(Val,Max,Min,Desc,Name,Flg,En,Td)
discrete value check
//  `XCHECK(Val,Name,Flg,En,Td)
check bus for X
//  `COMP_CHECK(DP,DN,Name,Out,En,Td)       Out=DP if
complementary, else X
// Process specific assertion macros:
//   macro        arguments              nominal range
//  ------------  ----------------------    -------
-----
//  `DVDD_CHECK(node,"name",ef,en)    1      +-10%
//  `AVDD_CHECK(node,"name",ef,en)    1      +-10%
//  `AVDDH_CHECK(node,"name",ef,en)  2.5     +-10%
//  `GND_CHECK(node,"name",ef,en)     0      +-0.1
//  `IBIAS_CHECK(expr,"name",nom,ef,en) nom     +-10%
//  `VBIAS_CHECK(expr,"name",nom,ef,en)  nom     +-10%
//  `RANGE_CHECK(expr,"name",Max,Min,ef,en)  check for
// in range
//  `DX_CHECK(expr,"name",errflg,en)        checks
// digital bus for any X
// Useful macro functions:
//  `FCUROUT(N1,N2,Iref,Kgain,Tr,dV,Ioff);    Current
// mirror output driver
```

3.3 Review of Limitations of the Approaches Discussed so far

While the application of the assert device along with checklimit analyses is useful to verify the device level characteristics, there is currently no way to set up and verify more complex circuit conditions that analog and mixed-signal verification tasks encounter. Some of these challenging operations include the ability to predicate assertions on some time varying circuit characteristics, or the ability to check temporal properties of a circuit at intervals determined by complex clocking conditions. Moreover, the current use model of using assert and checklimit creates an isolated solution in that the methodology only applies to the pure analog or mixed-signal applications and it neither leverages nor makes itself visible to the much broader digital verification methodologies that currently exist.

On the other hand, a mixed-signal HDL based approach for checking model behavior is more abstract and can cover a range of behaviors from device characteristic level to more complex temporal behavior. However, the limitation with this approach is that it does not leverage the set features that come with standard assertion languages. Measurement of coverage for the assertions that are set for a particular design block, ability to keep the design and verification aspects of a block separate from each other and having visual rendering tools dedicated to browse and debug assertions are just a few examples.

In contrast, the digital verification system has a well established use model for assertion based verification. This use model is based on standard assertion languages such as PSL and SVA and methodologies that have evolved over time to satisfy the verification needs in the discrete domain. In defining the language and methodology for assertion based verification that covers the entire spectrum of digital, mixed-signal and analog systems, we attempt to fully leverage this power and address the limitations that are listed above with the existing approaches.

4. USING PSL WITH VERILOG-AMS

4.1 PSL Assertions involving Analog Expressions

For the purpose of this discussion, analog expressions refer to the combination of legal Verilog-AMS operators and operands as defined by the Cadence Verilog-AMS Language Reference.

Analog expressions can appear in PSL assertions in Boolean expressions, clocking expressions and as actual arguments in property and sequence instances when there is a single top-level clock either defined explicitly or defined via a default clock.

```
electrical sig;
reg a, b, clk;
// top-level clock specified in the assertion
// psl assert always (V(sig) -> next(b)) @(posedge
clk);

// the default clock is inferred as the top-level
clock
// psl default clock = (posedge clk);
// psl property P1 = always {a;V(sig1)} |=>
{V(sig2);b};
// psl assert P1;
```

4.2 Analog Events for Assertion Clocking

Verilog-AMS analog event functions cross and above are supported as clocking events in PSL assertion.

```
electrical sig1, sig2, sig3;
reg a, b;
// psl assert always ({V(sig1);a} |=> {V(sig2);b})
@(cross(V(sig3)));
```

4.3 Support for wreal in PSL

The `wreal` net type represents a real-valued physical connection between structural entities in the Verilog-AMS language. For more information on wreal net type and how it supports more than one driver, refer to the section on Real Nets in Cadence Verilog-AMS Language Reference.

Expressions involving wreal type objects that are explicitly declared can appear in PSL assertions in boolean expressions, clocking expressions and as actual arguments in property and sequence instances.

```
wreal mywreal1, mywreal2;
reg clk;
// psl assert always ({mywreal1 > 4.4; mywreal2 <
6.6}) @(posedge clk);
```

4.4 Module bound Verification Units

vunits can be used for analog psl assertions. This is a very useful feature if the source text of the design block should not or cannot be modified. A vunit is a side file that is linked to the design file for simulation. Thus, the design file remains untouched and the assertion code is provided reside in the vunit file.

```
vunit myvunit(test) {
//psl assert (V(sig1) > 1.4) @(cross(V(sig3)));
}
```

Note that vunits are mainly used to store the assertion code; however, they are not limited to assertions only. If additional behavioral code is needed for the assertions, like storing some values in variable/registers, it can be added to the vunit as well. Consider to use this feature as it makes the coding of assertions much easier in some cases.

Verification units can be used to add assertions to Verilog/Verilog-ams/System-Verilog or VHDL instances.

```
vunit myvunit(test) {
// psl property P1 = ({V(sig1)} -> next (V(sig2))
@(cross(V(sig3)));
//psl assert P1;
}
```

4.5 Support for PSL built-in Functions

Analog expressions are allowed within the `prev` PSL built-in sampled value function. It is an error to have analog expressions as arguments to built-in sampled value functions other than `prev`.

```
// psl assert always ({V(sigout) > 0.5} |=>
{prev(V(sigout)) > 0.4}) @(cross(V(sigout)));
```

4.6 Coverage Analysis for PSL Assertion

Assertions are an important part of the coverage driven verification environment. Coverage points indicate whether the stimulus was able to create the conditions necessary to test the design's behavior. This information is critical to ensuring that the design has been sufficiently tested. This is typically achieved by defining signals and expressions that are being asserted on as coverage points and also defining the assertions themselves as coverage monitors.

Similar applications of coverage are expected to occur when using assertion in a mixed-signal context. In particular, an assert directive, when comprising of mixed-signal expressions in the boolean layer will take part in coverage report and therefore will provide valuable insight into whether the assertions are adequately checking all the analog conditions the design is passing through. Similarly, creating coverage points for the actual expressions that make up an assertion provide valuable insight into whether the assertion itself needs to be refined to verify the design properly.

5. USING SYSTEM-VERILOG ASSERTION IN A MIXED-SIGNAL DESIGN

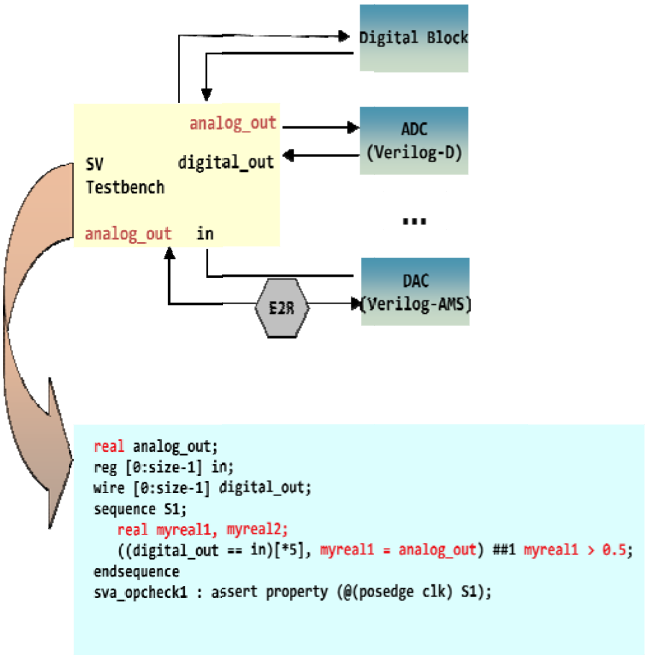
SystemVerilog Assertion (SVA) is a legal subset of the SystemVerilog P1800-2009 standard. This version of the SystemVerilog standard does not allow the presence of a continuous domain object. Therefore creating analog expressions is not possible.

However, the Cadence IUS[7] implementation of SystemVerilog assertions allows real data types in the Boolean Layer. In the context of analog design, this can be used if a real valued SV port is connect to nets belonging to the electrical domain. The following example illustrates this use model:

```
module top;
  var real r, xr, wr;
  assign xr = 3.14;
  ams_electrical_src e_s1(r);
  // causes insertion of Electrical2Real
  // connection module
  ams_electrical_dst e_d1(xr);
  // causes insertion of
  // Real2Electrical connection module
  ams_wreal_src      w_s1(wr);
  // Coercion of SystemVerilog real variable
  // to wreal
endmodule

module ams_electrical_dst(e);
  input e; electrical e;
  initial #10 $display("%M: %f", V(e));
endmodule

module ams_electrical_src(e);
  output e; electrical e;
  analog V(e) <+ 5.0;
endmodule

module ams_wreal_src(w);
  output w; wreal w;
  assign w = 2.5;
endmodule
```

Once a SystemVerilog `real` variable imports the AMS functionality as shown above, it can appear in any SystemVerilog assertion statement that permits the use of `real` variables. The simple example below shows how this is being done

```
input  var real outp, outm; simple_SVA_example:
  assert property (@(posedge(hold))
    ((outp-outm) < 10));
```



For the sake of illustration, we present another schematic above that explains how the `Electrical2Real` connection can help bring a real value into a SystemVerilog testbench to be used in an SVA block.

6. USING VALUE FETCH TO APPLY ASSERTION ON PURE ANALOG CHARACTERISTICS

Fetching values of continuous-domain objects into discrete domain is a common practice in verification that testbench methodologies use to account for mixed-signal effects. Such value fetch requires ability to transcend language boundaries across arbitrary levels of hierarchy through discrete and continuous domain language layers. Another requirement of such fetch operation is to be able to work from pure digital languages that do not understand continuous-domain syntax or semantics. In a more advanced application, verification engineers may require to have ability to choose between sloppy fetch (with interpolation between the last accepted analog solution point and the speculative solution) or accurate fetch (with an analog solution created at the time point when the fetch request is made). Such fetch operations usually require a broad spectrum of quantities that may need to be queried such as potential, flow, power, operating point parameter values etc. Unfortunately, current mixed-signal languages do not offer such features.

To overcome this limitation, a new system function called `$cds_get_analog_value` has been introduced in the Cadence AMS Designer Simulator. The function is used like this:

```
real x;
always @(…)
   x = $cds_get_analog_value(“top.sub.end.foo”,
“potential”);
```

The user provides the hierarchical string to the object to be accessed and defines a quantity specifier (in this case potential to the voltage) and the return value is a real number that represents the analog voltage at the moment where the function is triggered.

The syntax of the `cds_get_analog_value` function is the following:

```
real $cds_get_analog_value(hierarchical_name [,
optional index[, optional quantity qualifier]])
```

where:

- The `index` can be `variable`, `reg`, or `parameters` so long as their value evaluates to an integer constant.
- The `quantity` qualifier can be `potential`, `flow`, `pwr`, or `param`. If none is specified, `potential` is assumed.

The object referred to by hierarchical_name must exist and must be owned by the analog solver. It must be a scalar or a vector, and if the later, the index must be specified, such that the result resolves to a scalar. The hierarchical_name can be a relative or absolute path.

Note: It is possible to check whether the object referred to by hierarchical_name meets these conditions by using the helper functions `cds_analog_is_valid`, `cds_analog_exists`, and `cds_analog_get_width`. These helper functions enable the user to create reusable testbenches where the representation of the model containing the object that the value fetch routine points to can change from digital to analog or vice versa.

The value fetch routine can be called from within:

- Verilog, SystemVerilog or Verilog-AMS scope

The following calling scopes are not considered for the current version:

- VHDL, VHDL-AMS, SystemC, Specman/e, Verilog-A (if not compiled as Verilog-ams code)

The fetch routine needs to reference analog object. It can reference into any analog language:

- Verilog-AMS, VHDL-AMS, Spectre, Spice Verilog-A (compiled as Verilog-ams or included by ahdl_include)

7. RELATED ACTIVITIES IN THE STANDARDS COMMITTEES

Two standards groups are actively working towards standardizing analog/mixed signal assertions:

- ASVA. The Analog System Verilog Assertions committee is focusing solely on analog/mixed signal extensions to the SVA subset of the System Verilog language.
- SV-AMS. This group are defining AMS extensions to the entire System Verilog language, a work which parallels what Verilog-AMS has previously done for Verilog. The outputs generated by the ASVA group are expected to feed into this longer term SV-AMS effort.

There is ongoing discussion regarding bringing aspects of ASVA into the Verilog-AMS language. There also is discussion happening to explore how analog simulation cycle can be fit into the overall SV simulation cycle and where would analog assertions fit in that cycle.

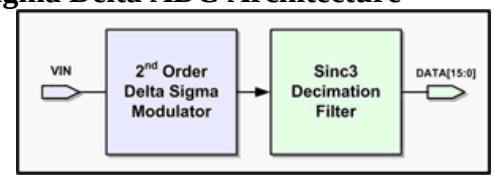
The Cadence SVA implementation discussed herein depends only on the support of `real` valued data-types in SVA, which is expected to be a fundamental change that will be incorporated by both groups.

8. TYING IT ALL TOGETHER: A MIXED-SIGNAL SIGMA DELTA ADC EXAMPLE

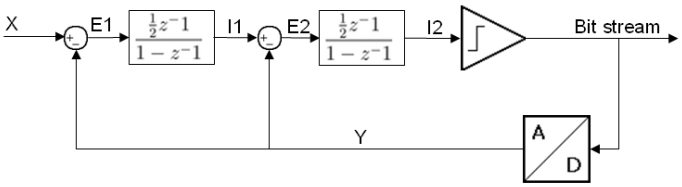
Sampled Data circuits are ideal for assertion based debugging, since the PSL/SVA assertion languages allow and encourage reasoning about sequential circuit behavior, and sampled data circuits (DACs, ADCs, SERDES, Switched Capacitor Filters etc.) by their very nature are sequential circuits.

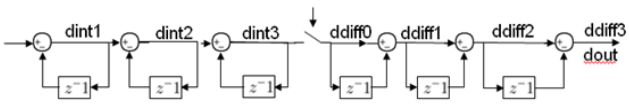
A Sigma Delta ADC (Modulator and Low Pass Digital Filter) is a typical example. Even the (essentially) analog modulator itself exhibits sequential behavior. In this example, we fully exploit the fact that, PSL/SVA Assertions aren't limited to just reasoning about sequential behaviors, but can also be used to reason about *predicated* mixed signal behavior e.g. *if* something happens, then something else must (or must not) happen (possibly at the same time, possibly later).

8.1 Sigma Delta ADC Architecture



The Sigma Delta architecture couples a modulator [4] with a Low pass filter. For ADC designs, the modulator is analog and the filter digital, and vice versa for DAC designs. A key characteristic of the Sigma Delta Modulator design is a feedback loop, with a very low order (often just a single bit) quantizer. Integrator(s) are inserted into the loop in order to shape the quantization noise. The output from the quantizer (comparator/latch combo) is a bit-stream, which is largely modulated by the input signal, and contains quantization noise.





Our modulator architecture features a second order feedback scheme, with discrete (switched capacitor) integrators modeled using *H(z)* transfer functions shown in the above diagram. Also shown is the architecture for the *Sinc3* filter. The node names used will also be referenced throughout this extended example.

In the simulation Verilog-AMS testbench, An ADC (`i1`) module is instantiated, which further contains a modulator instance (mod1), and a filter_decimator (actually *Sinc3* filter) instance `df1`. The modulator and filter are both modeled using Verilog-AMS. Also instantiated are a sine-wave voltage source generator (input), and a clock generator (pulse waveform). The ADC input is a sine-wave, with amplitude 0.65 and a frequency of `Tsig`. The clock for this circuit has a chosen frequency that corresponds to

an oversampling rate of 256 times the Nyquist rate of the input sine-wave.

A manual (laborious) inspection of the simulation waveforms in order to determine if the core loop characteristics of this sequential circuit architecture are continually upheld is a somewhat tall order (even for a short number of input wave periods) due to the high oversampling rate.

We use Assertion Based Verification (ABV) while the simulation is running to complement the more traditional (and laborious) methods of waveform inspection as a post-processing step.

8.2 Assertion Properties

All assertions are evaluated upon the same default clock used to switch the modulator integrator circuits.

```
vunit my_psl_vunit_all(ADC) {
// DEFAULT CLOCK FOR ASSERTIONS
//default clock = (timer(254.5*80e-9, 8*80e-9));
default clock = (cross(V(clk), +1));


// modeling layer. Create some expression placeholders
// (used in pos_integ1 assertion)
integer i1_pos, i1_inputs_pos;
integer vx_le_half_vref;
real abs_vx, abs_vref, abs_vi1, abs_vi2;
integer abs_vx_close_0;


analog begin
  i1_pos = V(I1) > 0.0;
  i1_inputs_pos = (V(X) > 0.0) && (V(I1) > 0.0) &&
(V(Y) <= -V(Vref));
  vx_le_half_vref = abs(V(X)) <= abs(V(Vref))/2.0;
  abs_vx = abs(V(X));
  abs_vref = abs(V(Vref));
  abs_vi1 = abs(V(I1));
  abs_vi2 = abs(V(I2));
  abs_vx_close_0 = abs(V(X)) <= 0.005;
end
```

We use the modeling layer of the vunit (above) to introduce several auxiliary Verilog-AMS variables that are referenced with the assertions detailed below.

The first pair of assertions simply test that the first integrator preserves the sign of arithmetic operations, a fundamental property of an integrator circuit. The first assertion tests that in any given cycle where the inputs to the integrator are positive, the output from the integrator in the subsequent cycle must also be positive. The second tests the converse (though is more verbose as the modeling layer is not used)

```
// INTEGRATORS and DIFF JUNCTIONS, basic behavior
// Check that integrators preserve sign of arithmetic
operations
// ie. assert that when V(in) and V(I1) both positive,
and comparator feedback
// is negative,
// then the first integrator output in the next cycle
must be positive.
// Ditto with polarities flipped
pos_integ1:  assert  always  {  i1_inputs_pos  }  |=>
i1_pos;
neg_integ1: assert always { (V(X) < 0.0) && (V(I1) <
0.0) && (V(Y) >= V(Vref)) } |=> V(I1) < 0.0;
// COMPARATOR BASIC FUNCTIONALITY
//  if  the  input  to  the  comparator  (integrator  2
output) is positive,
// ensure the comparator detects that immediately, and
vice versa
comparator_pos: assert always ((V(I2) > 0.001)-> (V(Y)
>= V(Vref)));
comparator_neg:  assert  always  ((V(I2)  <  -0.001)  ->
(V(Y) <= -V(Vref)));


// ensure integrator 2 output above threshold before
comparator output goes high
// if I2 output is positive, then ensure it goes high
before (or at) the time the comparator output is high
integ_to_comp1: assert always V(I2) < 0.0 -> V(I2) >=
0.0 before_ V(Y) >= V(Vref);
// if comparator output is negative, then ensure it
stays negative until integrator 2 output positive
integ_to_comp2: assert always V(Y) <= -V(Vref) -> V(Y)
<= -V(Vref) until V(I2) >= 0.0;
```

The next four assertions check the basic comparator operations. The first two of these check that if the I2 integrator output (the input to the comparator, which assumes a zero detection threshold) is positive/negative respectively, then the value fed back from the comparator via the one bit DAC is greater than `V(Vref)` and less than or equal to `-V(Vref)` respectively. These are examples of *predicated assertions*.

The third (`integ_to_comp1`) ensures that once the `I2` integrator output goes negative, then it has to subsequently go positive again before (or during) the cycle in which the comparator/feedback is positive (i.e. `>= V(Vref)).` This is an example of a *condition predicating a sequence of events*, and the events in that sequence must happen in the specified order. Note the use of the *before_* keyword.

The fourth (`integ_to_comp2`) assertion above is a somewhat mirrored example, testing that once the comparator/feedback output has become negative, then it must remain negative (strictly) until the `I2` integrator output (which is the input to the comparator) has again become positive. Note the presence of the *until* keyword.

A second set of properties (below) test loop stability fundamentals (higher order modulators are notoriously prone to instability).

The first loop stability property tested (`vin_less_vref` in the following screenshot) captures a *key design assumption*, that the input voltage to the modulator/ADC never exceeds half the reference voltage.

This is an example of using a property to formally capture an assumption made in the design IP regarding the environment in which the design is to be subsequently integrated. By capturing assumptions in this way, simulations can check that the input constraint is never violated in an integration context.

```
// INPUT ASSUMPTION PROPERTIES
// assert at every oversampling clock that the input voltage is constrained
// within half of Vref to avoid risking instability
vin_less_vref: assume always abs_vx <= abs_vref/2.0;

// STABILITY PROPERTIES
// assert that integrator outputs are bounded within +/-1.5*vref
// as instabilility tends to force integrator outputs out of bounds
integ1_bounded: assert never abs_vi1 > 1.5*V(Vref);
integ2_bounded: assert never abs_vi2 > 1.5*V(Vref);

// ensure that a LONG (7 in a row) sequence of CONSECUTIVE
// ones or zeros from the comparator doesn't
// happen, as this would also indicate instability
no_long_one_seq: assert never {V(Y) >= V(Vref)[*7]};
no_long_zero_seq: assert never {V(Y) <= -V(Vref)[*7]};

// test for limit cycle sequence of 1100110011001100
limit_cycle_p1: assert never { { {V(Y) >= V(Vref)[*2] ; V(Y) <= -V(Vref)[*2] }[*2] }[*2] };
// test for limit cycle sequence of 0011001100110011
limit_cycle_p2: assert never { { {V(Y) <= -V(Vref)[*2] ; V(Y) >= V(Vref)[*2] }[*2] }[*2] };
```

The next group of properties ensures that the integrator outputs remain properly bounded, in this case within 1.5 times `V(Vref)`. (Note: unstable Sigma Delta Modulators typically exhibit large signal swings in integrator outputs, and indeed more exotic designs have additional circuitry to detect such large swings and reset/nullify the integrators to break the oscillations. See references). For our example, we check the integrator output levels to ensure they remain in bounds.

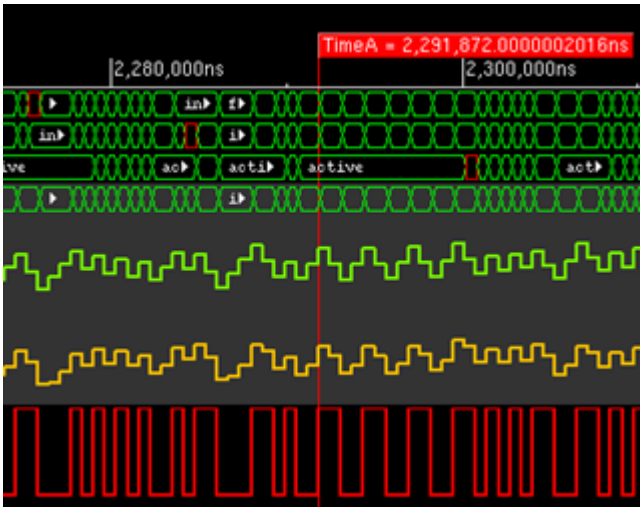
A second and related characteristic of unstable modulators is the presence of particular bit patterns. In the next two properties, we assert that a series of consecutive comparator high value outputs (logic 1) or low value outputs (logic 0) is never supposed to happen for this particular design. Note the presence of the `[*7]` term in the sequence, which acts as a sequence multiplier using the PSL language.

An often undesirable property of modulators (even stable ones) is the presence of 'idle tones'[5] i.e. additional repeating bit patterns which can lead to audible tones/clicks when the modulator is used in audio applications. Correspondingly we have two limit cycle checks, coded to check for undesirable bit-stream sequences of `1100110011001100` and it's inverse.

8.3 Simulation Results

A large number of assertion failures were noted after performing a simulation, per the table below. Assertion waveforms allow debugging such failures (see red highlighted cycles within green assertion status waveforms) in the presence of other circuit simulation waveforms. In our example, we noted the high number of assertion failures were due to an error in the magnitude of a random dither signal being added to the modulator comparator in a not entirely successful attempt to minimize the likelihood of limit cycles.

Assertion Name	Finished Count	Failed Count
comp_not_stuck	15	3
comparator_neg	2176	138
comparator_pos	2170	139
integ_to_comp1	2053	264
integ_to_comp2	2053	125
limit_cycle_p2	0	2
no_long_one_seq	14	2



9. CONCLUSIONS

In this paper, our objective was to take a broad look at the existing challenges in analog and mixed-signal verification and then evaluate how an assertion based verification concept addresses some of these challenges and also bring forth many new possibilities that are either too complex and expensive to develop and maintain in today's design and verification methodologies, or are not possible at all. We then reviewed a set of extensions in the standard PSL and SystemVerilog assertion languages that enable users to develop complex assertions on their analog and mixed-signal models.

Finally, we applied the language extensions that we described on a mixed-signal Sigma Delta ADC design. We verified several different types of properties related to sampled data/sequential analog circuit behavior:

- predicated events (condition A occurring implies condition B must/must not occur)
- predicated sequences (condition or sequence A occurring implies sequence B must (or not) occur in specified order
- checks for desired and undesired repetitive sequences
- extended checks over multiple clock cycles
- constrained sub-sequences that must happen within a larger sequence, in a given timeframe (number of clocks) etc
- predicated/triggered conditions that had to happen before other conditions happened
- predicated/triggered conditions that had to hold until some other condition happened

These checks were performed during a running simulation, not as a post-processing step.

Waveform inspection alone is a laborious debug method for analog and/or mixed signal circuits. Within this paper, we have seen how complementing the waveform approach with an Assertion Based Verification (ABV) approach during AMS verification leads to much quicker and more rigorous identification of bugs/issues i.e. improved throughput. The various properties captured as assertions and assumptions have varied from the very simple to the reasonably complex, and some of the violations wouldn't be immediately apparent from a casual waveform inspection alone. Inspection of assertion status waveforms superimposed on circuit node waveforms makes it easier to identify and debug issues in the correct context.

10. REFERENCES

[1] 1800-2009 IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language. (2009). Retrieved from IEEE Xplore: http://ieeexplore.ieee.org/servlet/opac?punumber=5354133

[2] 1850-2010 IEEE Standard for Property Specification Language (PSL). (2010 , April 6). Retrieved from IEEE Xplore: http://ieeexplore.ieee.org/servlet/opac?punumber=5445949

[3] Cadence Virtuoso Spectre Circuit Simulator. (n.d.). Retrieved from www.cadence.com: http://www.cadence.com/products/rf/spectre_circuit/pages/default.aspx

[4] Delta-sigma Modulation. (n.d.). Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Sigma_delta

[5] Perez Gonzalez, E., & Reiss, J. D. (n.d.). Idle Tone Behavior in Sigma Delta Modulation. Retrieved from AES E-Library (Audio Engineering Society): http://www.aes.org/e-lib/browse.cfm?elib=14093

[6] Verilog-Analog Mixed Signal Technical Subcommittee. (n.d.). Retrieved from Accelera: http://www.accellera.org/activities/verilog-ams/

[7] Incisive Enterprise Simulator (n.d.). Retrieved from www.cadence.com: http://www.cadence.com/products/fv/enterprise_simulator/pages/default.aspx