Catching the Elusive Voltage Spike with Analog/Mixed-Signal SVA/PSL Assertions

Charles Dančak [1]
Betasoft Consulting, Inc.
charles@betasoft.org

***Abstract*-A promising start has been made across the industry to incorporate analog/mixed-signal (AMS) assertions into UVM-style testbenches. One gap in this effort still remains: Assertion code written in SystemVerilog (SVA) or in clocked Property Specification Language (PSL) syntax can check signal values only at discrete times defined by a clock. Analog signals, however, are routinely subject to glitches over a time continuum, due to noise, crosstalk, etc. This paper seeks to bridge the gap by presenting assertion code and testbench instrumentation able to detect narrow voltage spikes continuously in time as they are injected onto a wire. The particular design example used is an op-amp bias line.**

I. Introduction

The efficacy of SVA or PSL assertions for AMS verification has been demonstrated by practical, in-depth articles on such topics as: sigma-delta modulation [1]; DDR2 memory [2]; PMUs [3]; and automotive sensor interfaces [4]. In these investigations, voltages and currents in the continuous-time domain are typically handled by means of:

- language extensions, including the Verilog-AMS **`wreal`** type, **`analog`** construct, or **`cross()`** function.
- tool-specific tasks like **`$cds_get_analog_value()`** to fetch analog values for use in SVA or PSL code.
- academic or company-specific enhancements—for instance, STL—extending the capabilities of PSL or SVA.

Much of this work has relied on the well-known paradigm of sampling analog values in the discrete-time domain, at the edges of a digital assertion clock. In an effort to catch spikes that occur *in between* edges, Santonja [5] coded an adaptive clock with an irregular period. Using a Verilog-AMS **`analog`** block, he generated sampling clock edges every analog time step. This method caught a glitch on the output of a voltage regulator when its bandgap reference was swapped out. But even with this extra coding effort, there is no guarantee of detecting *every* spike. Due to the unpredictable timing of glitches, the problem appears intractable.

In this paper, we present a SystemVerilog testbench based on an analytic paradigm that enables *continuous-time* detection of glitches and spikes—with no need of adaptive clocking or co-simulation. As our example, we verify a 0.7-V bias line **`VBN`** leading to the *n*MOS bias pin of a CMOS op-amp. Fig. 1 shows a simplified model. The bias line is part of a larger design-under-test (DUT), a low-droput (LDO) voltage regulator we verified in a UVM testbench [6]. The lengthy on-chip bias line **`VBN`**, with parasitic resistance R_p and capacitance C_p, is subject to voltage spikes that can arise from external noise bursts, excessive crosstalk, or the swapping out of the LDO's bandgap reference:

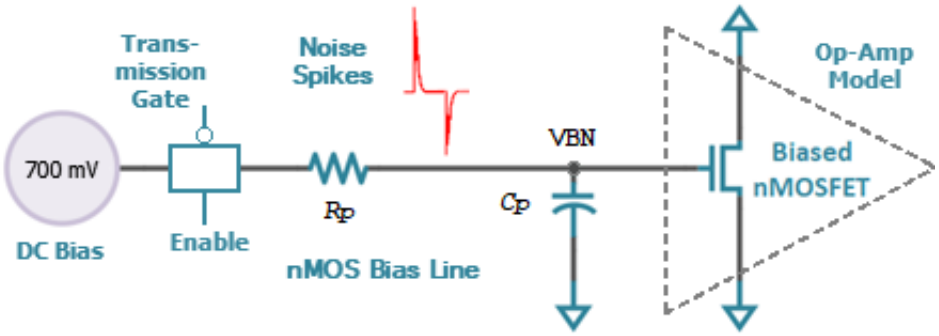

Figure 1. Simplified DUT Model with Bias Line Subject to Spikes

1 C. Dančak is also a SystemVerilog instructor with UC San Diego, Dept. of Extended Studies, La Jolla CA 92093, USA.

The key to detecting spikes *continuously in time* is to measure the local peaks and valleys of analog signal **VBN** using testbench instrumentation that can monitor analog voltages or currents independently of simulator timestep or sampling clock rate. We rely on Scientific Analog's XMODEL simulation package. XMODEL is a plug-in extension to industry-standard logic simulators. It represents analog waveforms in analytic form, with a time resolution limited only by the accuracy of the host computer's arithmetic unit. It has a large library of stimulus and response elements.

Section II is a glimpse into how XMODEL represents an analog signal using data type **xreal**. Predating industry support for SystemVerilog's versatile **nettype**, type **xreal** is a **struct** variable describing a voltage or current [7].

In Section III, we develop SVA code to detect bias-line spikes, including a Boolean condition, a subsequence, and a concurrent property—supported by the XMODEL measurement primitives **meas_max** and **meas_min**. We describe how these elements accurately compute local signal peaks and valleys, over an arbitrary user-specified time interval.

In Section IV we assert the SVA property and display SimVision results. Section V details a practical AC-coupled subcircuit we used to test our instrumentation by injecting picosecond-wide spikes onto the **VBN** line. Section VI is an equivalent PSL property, demonstrating that our continuous-time approach is independent of assertion language.

II. Analog Signal Representation in XMODEL

Fig. 2 shows a typical analog waveform, comparing its representation in SPICE versus XMODEL. Fig. 2(a) models the waveform as a series of time-value pairs. Simulation accuracy requires a fine time step. Fig. 2(b) is the same signal modeled in mathematical (piecewise-functional) form. Its accuracy is independent of the time step. From t_0 to t_1, the waveform is a rising exponential, with coefficient τ_1. After t_1, it becomes a decaying exponential with new coefficients. In XMODEL a simulation event (red or green dot) is generated only when the functional expression changes, and its coefficients updated. All the necessary signal information is accessed via a variable of **xreal** type:

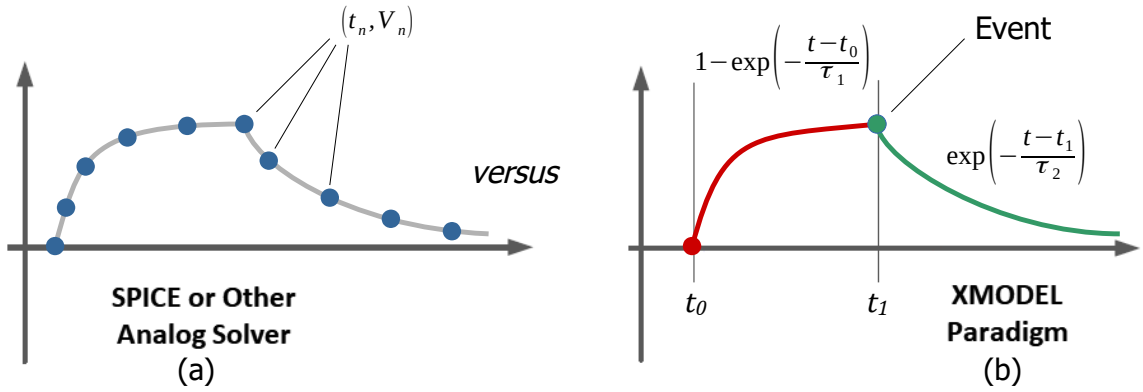

Figure 2. Comparison: SPICE versus XMODEL Signal

In general, XMODEL represents complex analog waveforms of **xreal** type in piecewise-functional form utilizing a weighted sum of exponential terms. Equation (1) shows this internal XMODEL representation in the time domain:

$$x(t) = \sum_i c_i t^{m_i} e^{-a_i t} \tag{1}$$

Transforming the summation in (1) to the Laplace **s**-domain, we obtain (2). By working in the Laplace domain, XMODEL eliminates the need for solving differential equations iteratively. To determine the DUT output, XMODEL multiplies the DUT transfer function by the input transform. Taking the inverse transform then yields the output.

$$X(s) = \sum_i \frac{b_i}{(s + a_i)^{m_i}} \tag{2}$$

The response of a linear system can thus be computed algebraically without solving integro-differential equations. An XMODEL user merely has to declare an analog signal like **VBN** to be of **xreal** type.[2] This ensures that the analog signal will be represented as in Fig. 2(b). A commercial logic simulator such as Xcelium, QuestaSim, or VCS is thus able to simulate the **xreal** signal in its testbench environment—at close to logic-simulation speeds.

2 **VBN** is readily converted for display purposes to a **real** variable, such as **VBN_real**, by using an **xreal_to_real** primitive.

In the following sections, we apply these concepts by developing SystemVerilog code for a property and assertion able to catch narrow spikes injected onto the **VBN** bias line—even when they occur between edges of the clock.

III. SVA Condition, Subsequence, and Property

In our test scenario, the LDO regulator starts up in **PWR_DN** state. When the transmission gate in Fig. 1 closes, the DUT enters **RESUME** state, and power is restored. Once the bias line **VBN** has risen exponentially to its valid voltage range, we begin injecting narrow spikes onto the line at random times. Checking for these spikes starts only when the restored bias **VBN** has reached its valid range. It continues until the LDO powers down again. These requirements are listed below. In turn, they translate into a Boolean condition, subsequence, and a property written in SVA syntax:

- Condition **VBN_VALID**: Bias voltage **VBN** is valid as long as it remains within the range 700 mV ± 50.
- Checking should start after a power-down interval, once the LDO enters the state **RESUME**, the bias is reapplied, and condition **VBN_VALID** goes true.
- Checking should continue for one or more cycles until the LDO exits **RESUME** state and powers down again.

These specifications are illustrated in the SimVision waveforms of Fig. 3. The enumerated **STATE** signal changes from **PWR_DN** to **RESUME**. Bias level **VBN** (red trace) gradually returns to its valid range (between the horizontal blue lines). Later, the testbench asserts the signal **INJ**, enabling the random injection of sharp spikes (red) onto the bias line. These random spikes are tall enough to cause **VBN** to rise above, or fall below, its valid range:

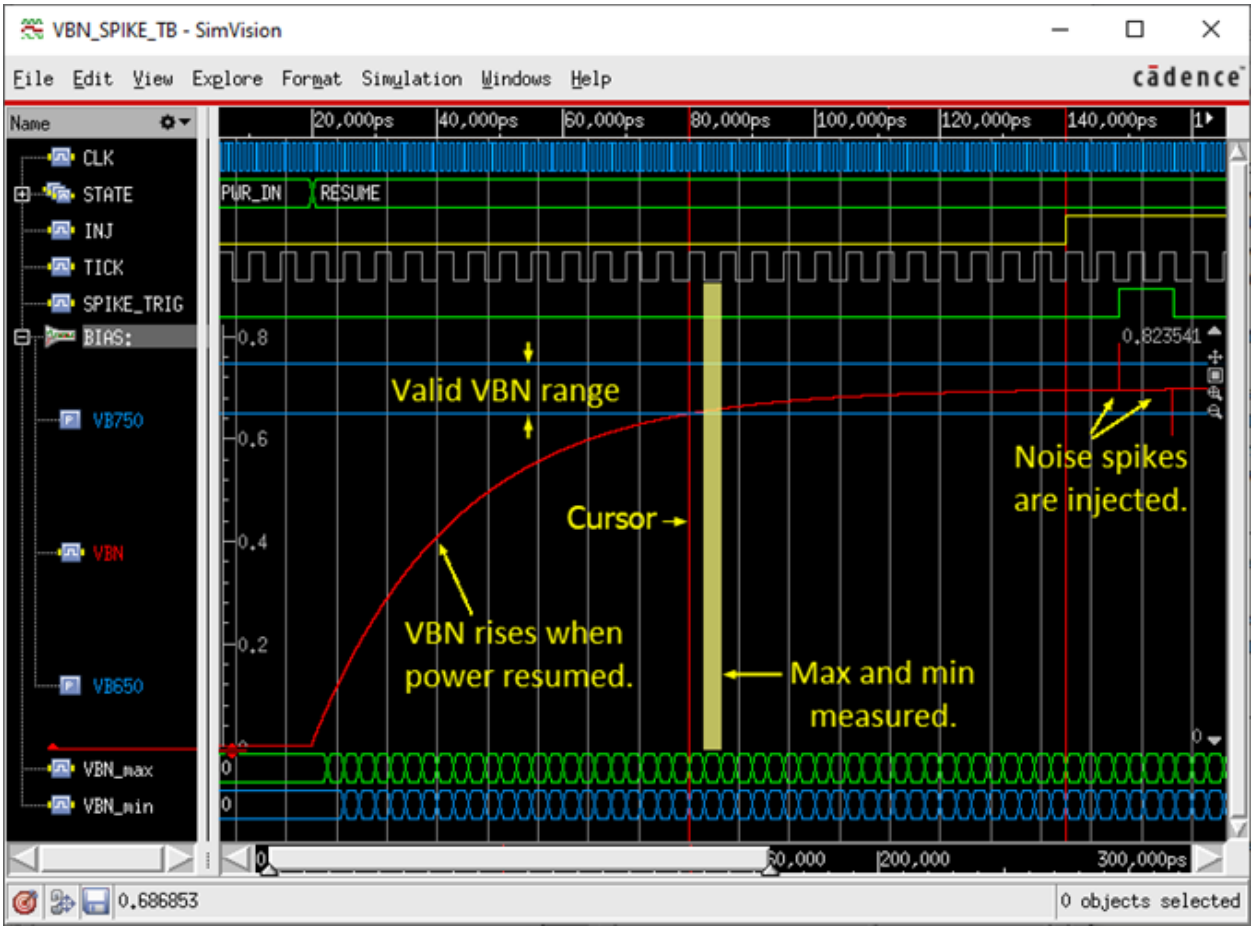

Figure 3. Bias Level Restored After Power-Down

The testbench code in Listing 1 declares **VBN** as a variable of **xreal** type. This is all an XMODEL user has to do to ensure representation of an analog signal in the piecewise-functional form described in Section II. Listing 1 declares signals **VBN_max** and **VBN_min**, the outputs of the measurement primitives **meas_max** and **meas_min**. As ordinary SystemVerilog **real** variables, they can be referenced directly in SVA code. The testbench signal **CLK** has a period of 1 ns. Measurement trigger signal **TICK** (gray trace) can be slower; it was arbitrarily given 2.5× the clock period.

```
//Testbench code for nMOS bias voltage (type xreal):
  xreal VBN;  //Converted to VBN_real for display.
  real  VBN_max, VBN_min, VBN_real;
//Upper and lower limits of VBN bias range (V):
  parameter real VB750 = 0.750, VB650 = 0.650;
```

Listing 1. SystemVerilog Testbench Declarations

Listing 2 instantiates **meas_max** and **meas_min** into the testbench, using any SystemVerilog instantiation syntax. We then reference their real-valued outputs to define the Boolean condition **VBN_VALID**:

```
//Measure max, min between successive TICK edges (type xbit):
  meas_max XP_MAX(.in(VBN), .out(VBN_max), .from(TICK_xb), .to(TICK_xb));
  meas_min XP_MIN(.in(VBN), .out(VBN_min), .from(TICK_xb), .to(TICK_xb));
//Boolean condition for bias in range:
  let VBN_VALID = ((VBN_min >= VB650) && (VBN_max <= VB750));
```

Listing 2. SystemVerilog Testbench Code for Condition VBN_VALID

These XMODEL library elements measure voltage maxima and minima over a well-defined time interval. They require a trigger signal like **TICK** to delimit the exact interval. (For precise edge timing, XMODEL actually employs **struct** type **xbit** for the trigger **TICK_xb** in Listing 2. This trigger is readily converted to **bit** type, however, for waveform display.) Notice a shortcut: **TICK_xb** drives both the **.from** and **.to** pins, avoiding the need for an extra trigger. With each successive edge of **TICK**, these elements will output the latest **VBN_max** and **VBN_min** values.

An element like **meas_max** finds the highest *peak* of its input signal during the time interval marked by successive **TICK** transitions. Similarly, **meas_min** finds the lowest *valley.* These values appear in numeric (or *Digital)* format at the bottom of Fig. 3. With this capability, an SVA property can check condition **VBN_VALID** continuously in time.

How does an element like **meas_max** find its input's highest peak over a given interval? It examines the signal's mathematical time-domain expression, as in Fig. 2(b). It computes the first derivative, then finds all of its falling zero-crossings. From this list of peaks, it picks the highest. This computation is carried out on an analytical expression—not a series of time-value pairs—and is thus *independent* of simulator time step or sampling clock rate.

Listing 3 declares an SVA subsequence, used to delimit the specified time window for checking **VBN_VALID**. This condition should hold true from the time **VBN** reaches its valid range in Fig. 3, then persist for one or more cycles up until the LDO's exit from state **RESUME**. We will use **WINDOW_seq** as a building block for the property to be asserted:

```
//Time window to check VBN_VALID:
  sequence WINDOW_seq;
    $rose(VBN_VALID) ##1 VBN_VALID[+] ##1 $fell(STATE == RESUME);
  endsequence: WINDOW_seq
```

Listing 3. SVA Sequence Defining a Time Window for VBN_VALID

Listing 4 declares property **VBN_STABLE_pro**. It requires that **VBN_VALID** hold true throughout **WINDOW_seq**. SVA operator **throughout** takes a Boolean condition as its left-hand side (LHS) operand and a sequence as its RHS:

```
//VBN shall remain valid during window:
  property VBN_STABLE_pro;
  //Activate when bias enters its range:
    (STATE == RESUME) && $rose(VBN_VALID) |->  //Antecedent clause.
      (VBN_VALID throughout WINDOW_seq);       //Consequent clause.
  endproperty: VBN_STABLE_pro
```

Listing 4. SVA Property for Checking VBN_VALID

A comment in Listing 4 identifies the *antecedent* clause, which triggers evaluation of this property. At that point, its assertion waveform displayed in SimVision becomes *active*. Of course, a property is only declarative code. In the next section, we assert this named property, directing the Xcelium simulator to actually verify its validity over a run.

Listing 5 declares a default clock edge for the checking of properties and sequences. Though the logic simulator checks **VBN_STABLE_pro** only on rising edges of the 1-ns clock, the measurement of peaks and valleys described in Section III still proceeds continuously in time. Variable **FAILURES** is used in printing a cumulative failure count.

```
//Default clock edge for checking:
  clocking CB @(posedge CLK); endclocking
  default clocking CB;
//Assertion failure count:
  shortint FAILURES = 0;
```

Listing 5. SVA Default Clock and Local Variable

IV. Assert Named SVA Property

Named property **`VBN_STABLE_pro`** is asserted by the statement in Listing 6. It tells the simulator to verify the requirement that bias **`VBN`** should remain valid, once it returns to its range until the next power-down cycle. Even a transient departure due to a random spike or glitch should yield a run-time error (such as **`*E,ASRTST`**). Fig. 4 shows a **`VBN_STABLE_chk`** assertion waveform (bottom trace). It indicates the first few failures as red waveform segments.

```
//Assert property VBN_STABLE_pro:
  VBN_STABLE_chk:
  assert property(VBN_STABLE_pro)
    $info("VBN_STABLE passing at: %t.", $realtime);
  else begin
    ++FAILURES;
    $error("VBN_STABLE failing at: %t.", $realtime,
      " Failure count: %2d.", FAILURES);
  end
```

Listing 6. SVA Assertion to Check Whether Bias Stays in Range

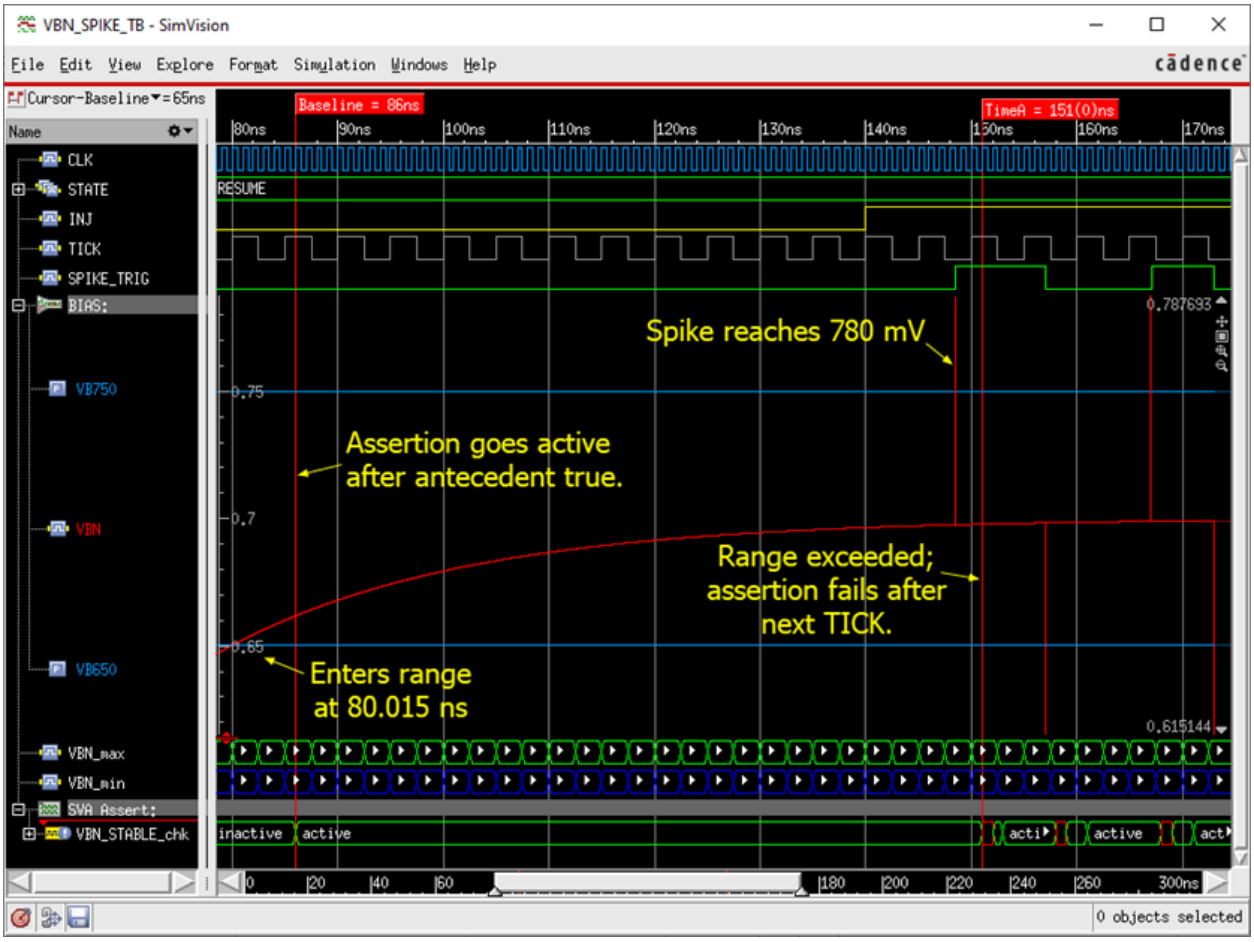

Figure 4. First Few Spikes Flagged as Assertion Failures

Fig. 4 zooms in on the same simulation run as Fig. 3. Most of the spikes, capacitively injected onto the bias line while **INJ** is high, fall between clock edges. These spikes would be missed by an SVA or PSL assertion unaided by XMODEL instrumentation. With **VBN_max** and **VBN_min** continuously monitoring peaks and valleys, however, each spike is caught by assertion **VBN_STABLE_chk**. To prevent a halt at the first failure, we issued the command below:

```
xcelium> set assert_stop_level never
```

Notice each assertion failure in Fig. 4 lags the spike by several cycles. This is expected for a clocked assertion. It takes a one-**TICK** interval of 2.5 cycles to measure the out-of-range **VBN_max** or **VBN_min** value. Only on the next rising edge of **CLK**—a cumulative lag of 3 cycles—does the simulator actually report the assertion state as failed.

Of the dozen spikes generated during the hundred cycles in which **INJ** stayed high, all were successfully caught. Pass/fail statistics for assertion **VBN_STABLE_chk** are summarized in the SimVision Assertion Browser in Fig. 5:

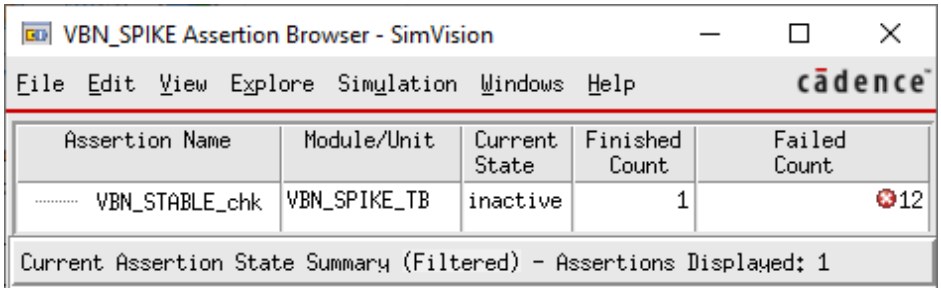

Figure 5. Detecting All 12 Injected Spikes During a Run

Does **VBN_STABLE_chk** ever *pass*? Yes, a pass does occur at the end of this run. It is cited in Fig. 5 under column *Finished Count* (though not included in our waveform views). After the last injected spike, **VBN_STABLE_pro** continues to hold without failing, right up to the time the LDO regulator exits **RESUME** state and powers down again.

V. Spike Injection Subcircuit

The injection subcircuit is built out of procedural SystemVerilog code and various XMODEL library elements. It is intended to produce very narrow spikes, testing our ability to catch them. Listing 7 shows the **initial** block where the injection begins. Its **while** loop creates the binary **SPIKE_TRIG** signal in Figs. 3 and 4. Its edges occur at random times controlled by local variable **DELAY**. Varying its random range will adjust the density of spikes. Each upward or downward edge of **SPIKE_TRIG** yields a corresponding spike, as is evident at the right of Fig. 4.

```
//Generate rising, falling edges:
  bit SPIKE_TRIG = 1'b0;

  initial begin: SPIKING
    realtime DELAY;
  //Avoid loop fall-through:
    wait (INJ);
  //Loop while injecting spikes:
    while (INJ) begin
      DELAY = $urandom_range(10000, 5000);  //Adjust the density of spikes.
      #(DELAY) SPIKE_TRIG <= ~SPIKE_TRIG;   //Delay next edge, in ps units:
    end
  end: SPIKING
```

Listing 7. Procedural SystemVerilog Code to Create Spikes

The schematic in Fig. 6 shows how **SPIKE_TRIG** is converted from **bit** type to timing-accurate **xbit** type, then fed into a **transition** filter which transforms the digital trigger into an analog pulse train. For simplicity, we used fixed-height pulses. These analog pulses are AC-coupled via series resistor R_c and capacitor C_c onto bias line **VBN**.

The resistors and capacitors R_c, C_c, R_p, C_p in this figure were chosen to produce sharp, narrow spikes. In order to emulate the slow exponential rise of **VBN** in Figs. 3 and 4, the DC bias source and transmission gate were replaced by an exponential generator primitive, **exp_gen**, whose output rises gradually to 700 mV. Fig. 6 thus models a lengthy bias line on a chip, subject to crosstalk-generated or other voltage spikes, feeding an LDO regulator DUT.

Listing 8 is the structural testbench code for the spike-injection path in Fig. 6. Ordinary SystemVerilog syntax is used to instantiate all XMODEL primitives. Instance **`B2X`** merely converts **`SPIKE_TRIG`** from **`bit`** to **`xbit`** type. Many XMODEL library primitives are parameterized; for example, coupling capacitor C_c has a value of 0.02 pF:

```
//Transform edges to upward, downward spikes:
  bit_to_xbit B2X(.in(SPIKE_TRIG), .out(SPIKE_TRIG_xb));
  transition #(.value0(0.0), .value1(1.2))  //Controls height.
    XP_TRANS(.in(SPIKE_TRIG_xb), .out(SPIKE_GEN_x));

//Couple the spikes into VBN line:
  resistor #(.R(10.0))
    XP_Rc(.pos(SPIKE_GEN_x), .neg(SPIKE_INJ_x));
  capacitor #(.C(0.02e-12))
    XP_Cc(.neg(SPIKE_INJ_x), .pos(VBN));
```

Listing 8. Structural SystemVerilog Code to Inject Spikes

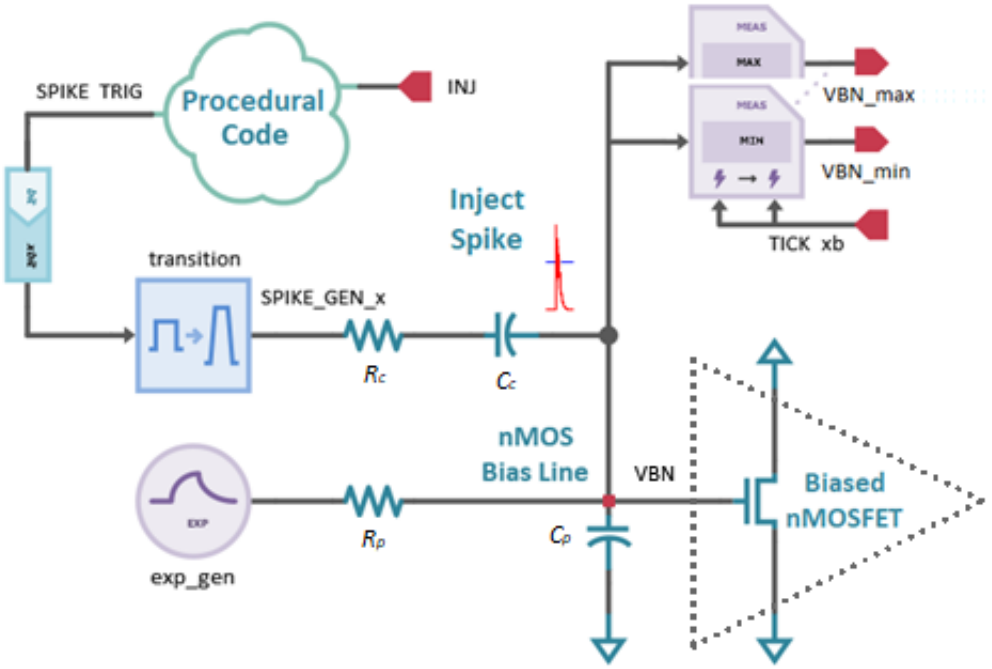

Figure 6. Spike Injection Subcircuit

Since we have relied on standard SystemVerilog syntax throughout this example, other logic simulators—such as VCS or QuestaSim—yield similar results. Due to our use of **`$urandom_range()`**, however, random spike times and counts may vary somewhat from tool to tool unless careful hierarchical seeding techniques are employed [8].

In Fig. 7, Scientific Analog's XWAVE viewer is used to directly display the **`xreal`** signal **`VBN`** (red trace). An inset zooms in on a typical spike, roughly 6 ps wide. Because each waveform segment of an **`xreal`** signal is an analytical expression, even a sharp spike is accurately modeled by a *single* simulation event (small red square). A new event is generated only when the analytical expression changes—for instance, from a rising exponential to a spike transient.

VI. PSL Property Code

While we have focused on SVA syntax in this paper because it is a standard, PSL syntax performed equally well. PSL has both **`sequence`** and **`property`** constructs, as well as a **`within`** operator not unlike SVA's **`throughout`**. All PSL code was embedded inside a **`vunit()`**, bound to our testbench—thus ensuring that all the signals were visible.

```
//VBN shall remain valid during window:
  property VBN_STABLE_pro =
    always ((STATE == RESUME) && rose(VBN_VALID))
      -> {{VBN_VALID} within {WINDOW_seq}};  //Braces required.
```

Listing 9. PSL Property for Checking VBN_VALID

Listing 9 is the PSL equivalent of **`VBN_STABLE_pro`**. Keyword **`always`** here means: check at *every single cycle*. The PSL operator **`within`** is similar to SVA's **`throughout`**, but takes two subsequence operands. In Xcelium, curly braces are required around both the individual subsequences, as well as the entire compound sequence [9].

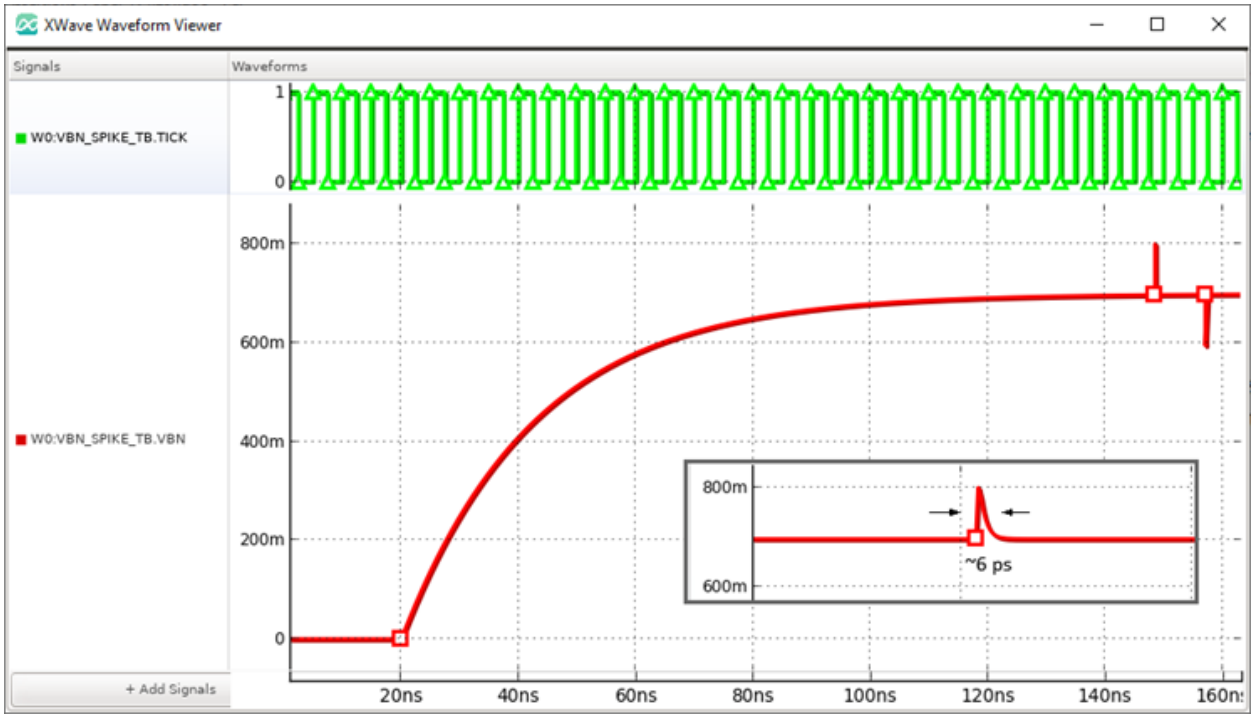

Figure 7. Bias Voltage Spike Detail in XWAVE Viewer

VII. Conclusions

This bias-line example has demonstrated how SVA or PSL assertions, supported by XMODEL instrumentation, can realize a key goal in analog/mixed-signal verification: detecting anomalous signal behavior in continuous time. We showed how to catch injected noise spikes, several picoseconds wide, falling in between assertion clock edges.

We outlined the key principles that enable this continuous-time checking. XMODEL represents any analog signal of **`xreal`** type as a mathematical (piecewise-functional) expression, as diagrammed in Fig. 2(b)—not as a series of time-value pairs, as in Fig. 2(a). Thus it can model arbitrarily-fast continuous-time signals, without a huge number of simulation events. In addition, various library elements like **`meas_max`** or **`trig_cross`** can monitor signal peaks or threshold crossings analytically, without the need to sample the signal repeatedly [11].

Our testbench is thus able to simulate an **`xreal`** bias voltage like **`VBN`** on an industry-standard logic simulator with little or no speed penalty, even at the circuit level shown in Fig. 6. There is no need to sample **`VBN`** more frequently to catch narrow spikes. Traditional Verilog methods, in contrast, detect spikes—or other signals with high-frequency components—only by sampling more often. This demands a shorter time step, directly impacting simulation speed.

All of our SystemVerilog code can readily be incorporated into a standard UVM framework—such as a testbench to apply directed line- and load-transient tests to an on-chip LDO regulator [6]. Using SVA syntax—in contrast to writing traditional Verilog property checkers—provides access to rich verification features like coverage metrics [1], simulator commands (e.g. **`assert_stop_level`**, **`$assertoff`**, etc.), and assertion browser windows as in Fig. 5.

Full-chip verification of analog blocks and digital interface logic is growing more crucial, and bias-line issues can be one prevalent source of bugs [10]. This paper highlights an effective technique that may help meet the challenge.

VIII. Acknowledgments

The author wishes to thank his colleagues Jaeha Kim for the spike-injection subcircuit, Rafael Betancourt for the full LDO design, and both of them for many insightful and encouraging discussions.

REFERENCES

[1] Bhattacharya, O'Riordan, Hartong (Cadence), "Mixed Signal Assertion-Based Verification," §5, 2011.
[2] Jones, Konrad, Ničković, "Analog property checkers: a DDR2 case study" in: Formal Methods of System Design (2010) 36: 114–130.
[3] Mukhopadhyay, Panda, Dasgupta, Gough (National), "Instrumenting AMS Assertion Verification on Commercial Platforms," ACM Trans. Design Autom. Electr. Syst., 14(2), 1–47 (2009).
[4] Nguyen (Infineon) & Ničković, "Assertion-based monitoring in practice: Checking correctness of an automotive sensor interface," 2015.
[5] Santonja (Freescale), "Reusable Continuous-Time Analog SVA Assertions," 2013. [Online]. See: //www.slideshare.net/slideshow/re-usable-continuoustime-analog-sva-assertions/21520118.
[6] Dančak, "A UVM SystemVerilog Testbench for Directed and Random Testing of an AMS Low-Dropout Voltage Regulator," DVCon 2024.
[7] Lim, Mao, Horowitz et al., "Digital Analog Design: Enabling Mixed-Signal System Validation," IEEE Design & Test, 2014.
[8] Smith (Doulos), "Random Stability in SystemVerilog," SNUG Austin 2013. §3.4.
[9] Doulos, "PSL Golden Reference Guide," Version 2.0 (2005): SEREs, p. 92.
[10] Barua, Farshad, Chang, "Advanced UVM-Based Chip Verification Methodologies with Full Analog Functionality," DVCon 2024.
[11] Stanley, Wang, Horowitz et al., "Fast Validation of Mixed-Signal SoCs," IEEE Solid-State Circuits Society, 2021.