December 6, 2010

Using Assertions in AMS Verification

Scott Little (scott.little@freescale.com)
AMS Verification Engineer
Why assertions?

► I shouldn’t have to evangelize to this audience, but…

► Characteristics of assertions:
  • Compact
  • Mathematically precise semantics
  • Reusable across verification methodologies and
    ▪ Simulation/formal
    ▪ Module/SoC level
  • Built-in coverage collection and metrics

► Quickly identify failing condition

► Standard error reporting mechanism
AMS verification

► Still a largely ad hoc process
  • Automated block-level regression suites rarely exist
  • General verification rigor we have come to expect with digital verification is often absent
    ▪ Immature verification plans
    ▪ Little or no random stimulus
    ▪ Coverage is not collected

► Disconnect between system-level and block-level
  • Minimal sharing of checkers/monitors
    ▪ If automatic checkers/monitors exist at all
  • Testing focuses on local behaviors not interaction with the system
Assertion-based AMS verification

► Assertions can:
  • Collect coverage
  • Be checked all levels of the hierarchy
  • Check interface assumptions

► Digital assertions have limitations
  • Real values cannot be referenced (according to the LRM)
    ▪ This works in practice for most simulators
  • Time is aligned to sampling events (clocks)
  • Time is discrete and can’t be referenced directly
  • Continuous quantities (voltage, current) can’t be accessed directly

► What do we do?
  • Approximate with current languages
  • Enhance current modeling/assertion languages
Our AMS verification methodology: A brief glimpse

► Analog blocks are designed and verified [designer]
► Abstract HDL model is created for the block [designer/modeler]
  • Written for use in SoC verification
    ▪ Performance is key
    ▪ Captures high-level functionality
► Abstract HDL model is verified vs. spec [modeler/verifier]
  • Assertions added and checked here
► Abstract HDL model is verified vs. schematic [modeler/verifier]
  • Pin names and directions match
  • High-level behavior matches
  • Timing may differ
► Abstract HDL model is used in SoC-level verification [verifier]
What do we check with assertions?

► Properties that are difficult to model but easy to check

► Interface assumptions
  
  • Illegal input combinations, sequences, or configurations
    ▪ Experimental feature is never enabled
    ▪ Analog inputs are isolated during digital test modes

► Power mode transitions
  
  • Check for conditions that result in floating nodes/leakage paths
    ▪ Block is never enabled without all power supplies present
    ▪ Level shifter is properly isolated when one supply goes down

► Timing relationships
  
  • Check “setup” times for analog blocks
    ▪ Circuit must be functional within the specified time after start-up
    ▪ Output is stable within the specified time after being enabled
Where do we write AMS assertions?

► SystemVerilog
  • Powerful SystemVerilog Assertions (SVAs) are available
  • Can’t access continuous quantities
  • Tend to use carefully timed clocks and multiclocked properties

► Verilog-AMS
  • Can’t write actual assertions
  • Have full access to continuous quantities
  • Use modeling code to approximate assertions

► SystemVerilog/Verilog-AMS
  • Digitize continuous signals using Verilog-AMS monitors
  • Pass digitized signals into the SystemVerilog module
  • Write assertions using the digitized signals, digital signals, and carefully timed clocks
**Property:** To avoid floating nodes ensure that when vdd1 is powered down either isolate is high or vdd2 is powered down. For the purpose of determining if a supply is powered up/down we will ignore “droops” of less than 25 ns.

```
//inertial delays squash short droops
//Note: initialization is ignored
assign #25 vdd1_down = !vdd1_gt_5;
assign #25 vdd2_down = !vdd2_gt_5;
always @(vdd1_down, isolate, vdd2_down)
  if(vdd1_down & !(isolate|vdd2_down))
    $error("ERROR: Floating node!");
```

```
Verilog-AMS checker

Verilog-AMS monitors

@cross(V(vdd1) - 5.0, +1)
  vdd1_gt_5 <= 1'b1;
@cross(V(vdd1) - 5.0, -1)
  vdd1_gt_5 <= 1'b0;
@cross(V(vdd2) - 5.0, +1)
  vdd2_gt_5 <= 1'b1;
@cross(V(vdd2) - 5.0, -1)
  vdd2_gt_5 <= 1'b0;
```
Verilog-AMS monitors example

// inertial delays squash short droops
// Note: initialization is ignored
assign #25 vdd1_down = !vdd1_gt_5;
assign #25 vdd2_down = !vdd2_gt_5;
always @(vdd1_down, isolate, vdd2_down)
  if (vdd1_down & !(isolate | vdd2_down))
    $error("ERROR: Floating node!");

@cross(V(vdd1) - 5.0, +1)
  vdd1_gt_5 <= 1'b1;
@cross(V(vdd1) - 5.0, -1)
  vdd1_gt_5 <= 1'b0;
@cross(V(vdd2) - 5.0, +1)
  vdd2_gt_5 <= 1'b1;
@cross(V(vdd2) - 5.0, -1)
  vdd2_gt_5 <= 1'b0;
Property: To avoid floating nodes ensure that when vdd1 is powered down either isolate is high or vdd2 is powered down.

SystemVerilog Assertion

```verilog
global:
    property noFloatingNodes;
    @(negedge vdd1_gt_5) 1'b1
    ##0
    @(posedge clk_1ns) (!vdd1_gt_5)[*25]
    |->
    !vdd2_gt_5 | isolate throughout !vdd1_gt_5[*0:$] 
;
endproperty
```

Verilog-AMS monitors

```verilog
@cross(V(vdd1) – 5.0, +1)
  vdd1_gt_5 <= 1'b1;
@cross(V(vdd1) – 5.0, -1)
  vdd1_gt_5 <= 1'b0;
@cross(V(vdd2) – 5.0, +1)
  vdd2_gt_5 <= 1'b1;
@cross(V(vdd2) – 5.0, -1)
  vdd2_gt_5 <= 1'b0;
```
property noFloatingNodes;
  @(negedge vdd1_gt_5) 1'b1
  ##0
  @(posedge clk_1ns) (!vdd1_gt_5)[*25]
  |->!vdd2_gt_5 | isolate throughout !vdd1_gt_5[*0:$]
endproperty

@cross(V(vdd1) – 5.0, +1)
vdd1_gt_5 <= 1'b1;
@cross(V(vdd1) – 5.0, -1)
vdd1_gt_5 <= 1'b0;
@cross(V(vdd2) – 5.0, +1)
vdd2_gt_5 <= 1'b1;
@cross(V(vdd2) – 5.0, -1)
vdd2_gt_5 <= 1'b0;
What bugs have we found?

► Incorrect enable sequencing
  • The system was enabling an analog block before it was stable. This early enable “crashes” the block.

► Floating analog blocks
  • Designer misunderstood the system configuration in a key low power mode. The isolation logic was incorrect and resulted in a large chunk of floating analog circuitry

► Incorrect initialization
  • For correct behavior leaving reset, specific outputs must be pulled low/high. This system-level requirement was not understood or implemented by the block designer.

► Inconsistent specifications
  • The minimum operating voltage was specified in an incompatible way between key blocks.
What is the future of AMS assertions?

- SVA is being added to Verilog-AMS
- Analog SVA (ASVA) effort is underway in Accellera and IEEE
- ASVA is an extension of SVA
  - Real values in boolean expressions
  - Realtime (i.e., continuous time) semantics
  - New realtime operators in sequences and properties
- Draws on Verilog-AMS
  - Eventually, ASVA will be a part of a unified SystemVerilog-AMS language
SVAs in Verilog-AMS modules example

Property: To avoid floating nodes ensure that when vdd1 is powered down either isolate is high or vdd2 is powered down.

SystemVerilog Assertion in a Verilog-AMS module

```verilog
property noFloatingNodes;
  @ (cross(V(vdd1) - 5.0, -1) 1'b1
      ##0
      @(posedge clk_1ns) (V(vdd1) < 5.0)[*25]
      |->
      V(vdd2)<5.0|isolate throughout (V(vdd1)<5.0)[*0:$]
    ;
endproperty
```

Verilog-AMS monitor

```verilog
@cross(V(vdd2) - 5.0, 0) ;
```
property noFloatingNodes;
    @(cross(V(vdd1) - 5.0, -1) 1'b1 ##0 @(posedge clk_1ns) (V(vdd1) < 5.0)[*25]
    |-> V(vdd2)<5.0|isolate throughout (V(vdd1)<5.0)[*0:$]
    ;
endproperty

@cross(V(vdd2) - 5.0, 0) ;
ASVA extension requirements

► ASVA committee has voted on extension requirements
  • ASVA should include all existing SVA
  • The meaning of existing constructs should not change
  • New constructs should provide realtime capabilities useful for AMS verification

► Existing SVA has a discrete semantic framework
  • Based on linear temporal logic and regular expressions

► Problem: Find a “good” way to extend SVA to realtime
  • Allow free intermingling of old and new operators, not just a union of old and new forms
  • Non-trivial
Extending SVA to realtime

► Some extensions are straightforward
  • Logical connectives (and, or, not) have the same meaning
  • Non-temporal implications (|->, if-else) have the same meaning
  • Clocked booleans (ignoring sampling questions)

► Some extensions have been studied in external research
  • Realtime linear temporal logic operators
    ▪ \( p \text{ until}[0:1.5\text{m}] q \) requires \( q \) to occur within 1.5 ms of the start of the property

► What about realtime sequences?
We invented a realtime semantic framework for sequences based on continuous intervals

- Proved equivalence between the new realtime semantics and the existing SVA semantics for the SVA sequence forms

Introduced three new primitive realtime sequence forms:

- $b$: realtime (i.e., unclocked) boolean
- $r$ without @(c): sequence without an event
- $b[^{a1:a2}]$: boolean “smear”, i.e., boolean holds continuously for a specified time range

Introduced several new derived realtime sequence forms:

- $r \#[a:b] s$: realtime concatenation
- $r \#0 s$: realtime fusion
- $b[\sim>1]$: realtime goto
Realtime sequence examples

- **a** is true and **b** is false continuously for 10.5 ns
  - \((a && !b) [*10.5n]\)

- **a** is true and 9.7 ms later **b** is true
  - \(a \#9.7m b\)

- From the beginning of the interval, advance to the first time where **a** is high, then find **b** and **c** high 1.6 ns later, and also ensure that **b** subsequently stays high continuously for 5.1 ms
  - \(a[-\rightarrow1] \#1.6n (b && c) \#0 b[*5.1m]\)
**Property:** From the time we see a valid bias voltage until the circuit gets enabled must be >= 5 µs.

```verilog
property noFloatingNodes;
  @(cross(V(vdd1) - 5.0, -1) 1'b1
  #0
  (V(vdd1) < 5.0)[*25.0n]
  ->
  V(vdd2) < 5.0 | isolate throughout (V(vdd1) < 5.0)[*0.0:$]

endproperty
```
property noFloatingNodes;
  @(cross(V(vdd1) - 5.0, -1) 1'b1 #0
  (V(vdd1) < 5.0)[*25.0n]
  |->
  V(vdd2) < 5.0 | isolate throughout (V(vdd1) < 5.0)[*0.0:$]
  ;
endproperty
Conclusions

► AMS assertions are a powerful tool in the verification toolbox
  • They are very useful when checking for illegal interface conditions, power down conditions, and timing sequences
  • They are in the process of maturing
  • Even while maturing they are very usable
  • They help find real bugs

► More information on the standards effort:
  • http://www.vhdl.org/twiki/bin/view.cgi/VerilogAMS/AmsAssertions
I would like to thank the following people for their collaboration:

- John Havlicek
- Himyanshu Anand
- Chris Stoll
- Ben Ehlers
- Neil Spake