System Verilog Assertion Linting: Closing Potentially Critical Verification Holes

Erik Seligman, erik.seligman@intel.com
Laurence Bisht, laurence.s.bisht@intel.com
Dmitry Korchemny, dmitry.korchemny@intel.com

Intel Corporation
Agenda

- Motivation
- Logically Wrong Assertions
- Potentially Ignored Assertions
- Performance Hazards
- Conclusions
Agenda

- Motivation
- Logically Wrong Assertions
- Potentially Ignored Assertions
- Performance Hazards
- Conclusions
Why SVA?

- Powerful language for assertions
  - Combinational and temporal logic
  - Triggered logical implication
    - Antecedent |-> Consequent
  - Usable in procedures, functions, modules
    - Concurrent and procedural code
- With library, easy for engineers to use
- Supported by almost all EDA tools
  - Simulation
  - Emulation
  - Formal Verification
But watch out…

- Projects discovered many wrong SVAs
  - Legal, but didn’t match user intention
  - Compiled correctly
  - Affected simulation and formal verification
- Why didn’t library solve?
  - Even with a library, flexibility in arguments
  - Interaction with user RTL code
- Corners of language hard to understand
  - Many ways to express same idea ➔ ways to express it wrong!

➔ Need to combine SVA usage with good methodology & safety checks
What are Lint Rules?

- Sanity checks on RTL
  - Syntactic code scan for common/likely mistakes
  - Flag code that is legal, but risky
  - NOT fancy formal engine checks
    - Though some modern lint tools offer these

- Three main types of rules we developed
  - Logically wrong assertions
  - Potentially ignored assertions
  - Performance hazards

- Presentation shows some examples
  - Many more in paper
Full Rule Set From Paper

- **Wrong Functionality**
  1. Assertion active at both clock edges
  2. Sequence used as clocking event
  3. Complex Boolean expression used for clock
  4. Wrong argument type or size
  5. $\text{stable(sig[index])}$ with variable index
  6. Non-sampled value in action message
  7. Property using negated implication

- **Possibly ignored assertions**
  1. Short-circuitable function has assertion
  2. Action block with no system function
  3. Unbounded assertion always true due to weakness
  4. Implication (|$->$,|$=>$) in cover property
  5. Bad comparison to unknown
  6. Assertion with constant clock

- **Performance Hazards**
  1. Many instances of single assertion
  2. Assertion in loop not using index
  3. Large or distant time windows
  4. Unbounded time/repetition operator in antecedent
  5. Using cover sequence rather than cover property
  6. Applying $\text{past}$ to multiple terms of expression
  7. Antecedents with empty match
Agenda

- Motivation
- Logically Wrong Assertions
- Potentially Ignored Assertions
- Performance Hazards
- Conclusions
Clock Edge Hazards

- Does this assertion find the bug in the waveform?

P1: assert property (@clk p|->q[*4]);
Clock Edge Hazards

- Does this assertion find the bug in the waveform?

\[ P_1: \text{ assert property } (@\text{clk } p \rightarrow q[*4]); \]

\[ P_{\text{correct}}: \text{ assert property } (@(\text{posedge clk})(p \rightarrow q[*4])); \]

**P1 misses the bug:** 4 phases $\Rightarrow$ 2 cycles $\Rightarrow$ 50% weaker check than intended!

**Lint Rule:** Flag any assertion clock without an edge
**Sampling A Variable Index**

```vhls
assign index = f_active_agent();
P1: assert property ($rose(req[index])|->!err);
```

- Should the property **pass** or **fail** here?
**Sampling A Variable Index**

```plaintext
assign index = f_active_agent();

P1: assert property ($rose(req[index])|->!err);
```

- Should the property **pass** or **fail** here?

It **fails**— *index* sampled just like other variables!
- On index rise, $rose compares **current** req[1] to **previous** req[0]

**Lint Rule:** Flag sampled value functions using a sampled variable as an index
Agenda

• Motivation
• Logically Wrong Assertions
• Potentially Ignored Assertions
• Performance Hazards
• Conclusions
function bit f_myfunc(...)
    mySVA: assert #0 (!bad_addr);  ... 
endfunction

assign A = UopV && f_myfunc(UopV);

Will it flag `bad_addr` in this trace?
Short Circuiting Hazard

function bit f_myfunc(...)  
    mySVA: assert #0 (!bad_addr);    ...
endfunction

...  
assign A = UopV && f_myfunc(UopV);

Will it flag bad_addr in this trace?

- No! SystemVerilog short-circuits boolean expressions

Lint rule: Flag functions with assertions in short-circuital positions
Implication In Cover Property

C1: cover property \((a \implies b)\);

Is C1 covered by this waveform?
Implication In Cover Property

C1: cover property \((a \implies b)\);

Is C1 covered by this waveform?

- **Yes!** C1 covers any cycle when \((a\Rightarrow b)\) doesn’t fail
  - Including cases when \(a\) is false

C2 is more useful:

\[
C2: \text{cover property } (a \#\#1 b);
\]

**Lint rule:** Flag any cases of implication in a cover property
Agenda

• Motivation
• Logically Wrong Assertions
• Potentially Ignored Assertions
• Performance Hazards
• Conclusions
Many Instances of Assertion

```verbatim
class always_comb:
    for(int i=0; i<1024; i++) begin
        P1: assert #0 ((~c[i] & ~(d[i] | e[i]))& f);
    end
```

// Logically same, but maybe 1024x efficient
```verbatim
class always_comb:
    P1: assert #0 (~(c & ~(d | e)))& f);
```

**Lint Rule:** Flag any assertion with more than <n> instances
Unbounded Repetition In Antecedent

A1: assert property (a[*1:$] |=> b);

• Potentially many evaluation threads in simulation
  • Think about case where a==1 for a long time
• Will alternate version match user intent?

A2: assert property ($fell(a) |-> b);

Lint Rule: Flag any use of unbounded repetition at the beginning or end of a left-hand-side of |->, |=> .
Agenda

• Motivation
• Logically Wrong Assertions
• Potentially Ignored Assertions
• Performance Hazards
• Conclusions
Conclusions

- Linting == important enabler for SVA
  - Lint well-established in other areas (C/C++, etc)
    - Important to advance in SVA as well
  - Rules in presentation were a sample—see more in paper
- SVA is powerful—and even more so with good lint
  - Intel has observed solid return on investment
    - 20% of reported logic bugs on recent project found thru SVA
    - Not including early local finds by RTLers
  - But great power ➔ ability to misuse
    - Misuses rare but important to catch
  - With new lint rules, expect even better ROI in future
Backup Slides
Poor Failure Reporting

P1: assert property (req && !stall |-> gnt) else 
$error(“P1 failed, stall = %d”,stall);$

P2: assert property (req && !stall |-> gnt) else 
$error(“P2 failed, stall = %d”,$sampled(stall));

- Signal values are sampled
- Action block uses current values

- Messages without $sampled report wrong values

Lint Rule: Require $sampled in action block display statements where appropriate
Unbounded Assertion Always True

P1: assert property (a |-> ##[1:] b);

P2: assert property (a |-> strong(##[1:] b));

What is the difference?
• P1 is a tautology: assertions weak by default
• P2 can be disproven by infinite trace with !b loop

Lint Rule: Flag unbounded assertions always true due to weakness
Cover Sequence vs Cover Property

C1: cover sequence ($fell(rst) ##[*] a);

C2: cover property ($fell(rst) ##[*] a);

- C1 is running continuously
- C2 is done after first report

Lint Rule: Flag any use of cover sequence