Getting started with SystemVerilog Assertions

Tudor Timisescu

VerificationGentleman.com

April 15, 2020

Introduction

SystemVerilog assertions (SVAs) express temporal behavior

- Compact
- Similar to regular expressions

Introduction

SystemVerilog assertions (SVAs) express temporal behavior

- Compact
- Similar to regular expressions

This talk won't be a comprehensive SVA tutorial

Top 5 essential SVA features

- Implication operator
- Delay operation
- Repetition operator
- Throughout operator
- Named properties and sequences

Implication

Overlapping implication:

```
antecedent |-> consequent
```

- If antecedent is 1, then consequent must be 1 as well
- If antecedent is 0, then consequent can have any value

Implication

Overlapping implication:

```
antecedent |-> consequent
```

- If antecedent is 1, then consequent must be 1 as well
- If antecedent is 0, then consequent can have any value

Don't grant any requests while busy:

```
assert property (busy |-> !grant);
```

Implication

Overlapping implication:

```
antecedent |-> consequent
```

- If antecedent is 1, then consequent must be 1 as well
- If antecedent is 0, then consequent can have any value

Don't grant any requests while busy:

```
assert property (busy |-> !grant);
```

New request while not busy is granted immediately:

```
assert property (req && !busy |-> grant);
```

Implication

Overlapping implication using immediate assert:

```
always @(posedge clk)
  if (antecedent)
    assert (consequent);
```

Implication

Non-overlapping implication:

```
antecedent |=> consequent
```

- If antecedent is 1, then in the next cycle consequent must be 1 as well
- If antecedent is 0, then consequent can have any value

Implication

Non-overlapping implication:

```
antecedent |=> consequent
```

- If antecedent is 1, then in the next cycle consequent must be 1 as well
- If antecedent is 0, then consequent can have any value

Start working on granted requests:

```
assert property (req && grant |=> busy);
```

Implication

Non-overlapping using immediate assert:

```
always @(posedge clk)
  if ($past(antecedent))
    assert (consequent);
```

Implication

Non-overlapping using immediate assert:

```
always @(posedge clk)
  if ($past(antecedent))
    assert (consequent);
```

Gotcha

$past(...) can be either 0 or 1 in the very first cycle $\rightarrow$ false negative

Implication

Non-overlapping using immediate assert (corrected):

```
always @(posedge clk)
  if (past_valid && $past(antecedent))
    assert (consequent);

bit past_valid = 0;
always @(posedge clk)
  past_valid <= 1;
```

Delay

One cycle delay:

```
a ##1 b
```

- a is 1 in the 1st cycle and b is also 1 in the 2nd cycle

Delay

One cycle delay:

```
a ##1 b
```

- a is 1 in the 1st cycle and b is also 1 in the 2nd cycle

Delay of multiple cycles:

```
a ##3 b
```

- a is 1 in the 1st cycle and b is also 1 in the 4th cycle

Delay

Delay in antecedent:

```
assert property (a ##1 b |=> c);
```

- If a is 1 in the 1st cycle and b is 1 in the 2nd cycle, then c must be 1 in the 3rd cycle

Delay

Delay in antecedent:

```
assert property (a ##1 b |=> c);
```

- If a is 1 in the 1st cycle and b is 1 in the 2nd cycle, then c must be 1 in the 3rd cycle

Delay in consequent:

```
assert property (c |=> a ##1 b);
```

- If c is 1 in the 1st cycle, then a must be 1 in the 2nd cycle and a must be 1 in the 3rd cycle

Delay

One cycle delay using RTL modeling:

```
enum { IDLE, A_SEEN, END } state, next_state;
bit match;

always_comb begin
  next_state = state;
  case (state)
    IDLE: if (a) next_state = A_SEEN;
    A_SEEN: next_state = END;
  endcase
end

assign match = (state == A_SEEN) && b;
```

Delay

One cycle delay using RTL modeling:

```
enum { IDLE, A_SEEN, END } state, next_state;
bit match;

always_comb begin
  next_state = state;
  case (state)
    IDLE: if (a) next_state = A_SEEN;
    A_SEEN: next_state = END;
  endcase
end

assign match = (state == A_SEEN) && b;
```

Gotcha

Not equivalent to SVA. Doesn't start a new attempt on each cycle.

Modeling SVA sequences using RTL

It's more difficult than it appears at first glance. A great resource on the topic is this post:
`https://tomverbeure.github.io/rtl/2019/01/04/Under-the-Hood-of-Formal-Verification.html`

Delay

Delay of three cycles using naive RTL modeling:

```
enum { IDLE, A_SEEN, WAIT1, WAIT2, END } state, ...;

always_comb begin
  next_state = state;
  case (state)
    IDLE: if (a) next_state = A_SEEN;
    A_SEEN: next_state = WAIT1;
    WAIT1: next_state = WAIT2;
    WAIT2: next_state = END;
  endcase
end

assign match = (state == WAIT2) && b;
```

Repetition

Consecutive repetition:

`a [*5]`

- a is 1 in the 1st, 2nd, 3rd, 4th and 5th cycles

Repetition

Consecutive repetition:

`a [*5]`

- a is 1 in the 1st, 2nd, 3rd, 4th and 5th cycles

After starting, an operation takes 5 cycles:

```
assert property (start |=> busy [*5]);
```

Repetition

Consecutive repetition (5 times) using immediate cover:

```
int unsigned counter;

always_ff @(posedge clk)
  if (a && counter < 5-1) counter++;
  else counter = 0;

assign match = (counter == 5-1 && a);

always @(posedge clk)
  cover (match);
```

Repetition

Goto repetition:

```
a ##1 b [->1]
```

- a is 1 in the 1st and b is 1 in some future cycle, starting from the 2nd cycle

Note

a ##1 is not part of the goto repetition definition, but it makes the explanation easier.

Repetition

Goto repetition:

```
a ##1 b [->1]
```

- a is 1 in the 1st and b is 1 in some future cycle, starting from the 2nd cycle

Note

a ##1 is not part of the goto repetition definition, but it makes the explanation easier.

The device starts operation after eventually granting a request:

```
assert property (req ##1 grant [->1] |=> busy);
```

Repetition

Goto repetition using immediate cover:

- haven't tried it out
- some kind of counter with b as an enable?

Repetition

Goto repetition using immediate cover:

- haven't tried it out
- some kind of counter with b as an enable?

Modeling using RTL becomes more and more complicated

- could build library of RTL-based SVAs
- worth it?

Throughout

```
a throughout b [->3]
```

- a is 1 in every cycle during a sequence in which b pulses 3 times

Throughout

While waiting for a grant for an issued request, no new requests shall be issued:

```
assert property (req |=> !req throughout grant [->1]);
```

Named properties and sequences

Might want to check the same property for different sets of signals:

```
property some_property(bit a, bit b, bit c, bit d);
  a ##1 b [*2] ##1 c [*3] |-> d;
endproperty

assert property (some_property(a0, b0, c0, d0));
assert property (some_property(a1, b1, c1, d1));
```

Using named properties avoids duplication.

Named properties and sequences

Might have the same sequence of signal behavior that is interesting for multiple properties:

```
sequence antecedent(bit a, bit b, bit c);
  a ##1 b [*2] ##1 c [*3];
endsequence

assert property (antecedent(a2, b2, c2) |-> d2);
assert property (antecedent(a2, b2, c2) |-> !e2);
assert property (antecedent(a2, b2, c2) |-> f2 ##1 g2);
```

Using named sequences avoids duplication.

Tips and tricks

Split long sequences, even if you don't plan on reusing the subsequences

- Read more like natural language
- Sequence names reveal intent
- Functions calling functions $\rightarrow$ sequences composed of sequences

Tips and tricks

Multiple smaller assertions instead of one big assertion

- Easier to write/read
- Easier to debug
- Might run faster
- More difficult to reason about (all requirements?)

Tips and tricks

Check that properties work as expected, especially that they fail when they should

- SVUnit
  - needs commercial simulator
- Dream: test using SymbiYosys
  - given a set of constraints, expect that the proof fails

Tips and tricks

Think about cause, not just effect

Tips and tricks

Think about cause, not just effect

- Causes lead to effect:

```
assert property (cause0 |-> effect);
assert property (cause1 |-> effect);
```

Tips and tricks

Think about cause, not just effect

- Causes lead to effect:

```
assert property (cause0 |-> effect);
assert property (cause1 |-> effect);
```

- Effect only due to causes:

```
assert property (effect |-> cause0 || cause1);
```

Resources

https://www.doulos.com/knowhow/sysverilog/tutorial/assertions/

- Covers clocking, reset, bind, more operators

https://zipcpu.com/formal/formal.html

- Formal verification, immediate assertions

https://tomverbeure.github.io/rtl/2019/01/04/Under-the-Hood-of-Formal-Verification.html

- FSM modeling of SVAs

https://standards.ieee.org/content/ieee-standards/en/standard/1800-2017.html

- Definitive resource for SVA