SystemVerilog's *always_comb*

Problems and Solutions

Matt Cohen
Oracle

April 27, 2017
Boston

Agenda

History lesson: always through the years

Deep dive: The problem with always_comb

Solutions: How to make always_comb work the way it should

History

"Those who cannot remember the past are condemned to repeat it."
– George Santayana

"History, as long as it continues to happen, is *always* another chance."
– R. Jackson Wilson

Verilog-95

“Making a list, checking it twice”

- Initial language standard defines a control construct (always)
- always can be controlled by a #delay:

```
always
  #50 a = ~a;
```

- For synthesizable logic, always requires an event expression (aka sensitivity list) instead:

```
always @(a or b)
  c = a & b;
```

- What’s the danger here?

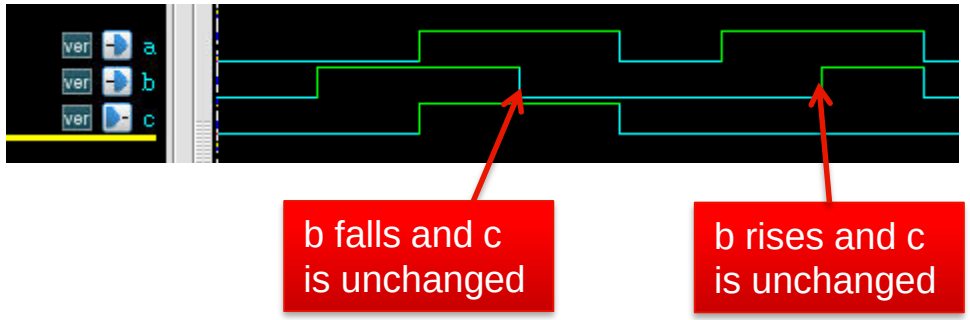
Verilog-95

"Making a list, checking it twice"

- Simulation respects the sensitivity list:

```
always @(a)
    c = a & b;
```

  - **c** does not update when **b** changes
  - Does not simulate as an AND gate



- Synthesis ignores the sensitivity list
  - Produces an AND gate
- "An incomplete sensitivity list on a combinational always block will typically cause a mismatch between pre-synthesis and post-synthesis simulations." – IEEE STD 1364.1
- Very dangerous – not caught until gate sims or silicon

Verilog-95

“Making a list, checking it twice”

- How did we deal with this?
- Many ad hoc solutions:
  - Linters can look for incomplete sensitivity list
  - Synthesis tools can provide warnings
  - Equivalence tools can give errors
  - Editor tricks like emacs AUTOSENSE
- Better solution: Move to v2k!

Verilog-2001

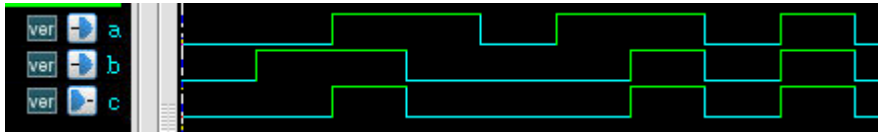
“Born to be wild(carded)”

- V2k introduces “implicit event_expression”, @*
- Shortcut to producing complete sensitivity list:

```
always @(*)                    always @(a or b)
  c = a & b;                     c = a & b;
```

Identical!

- Simulators and synthesis tools agree:



- Is the problem solved for good?

Verilog-2001

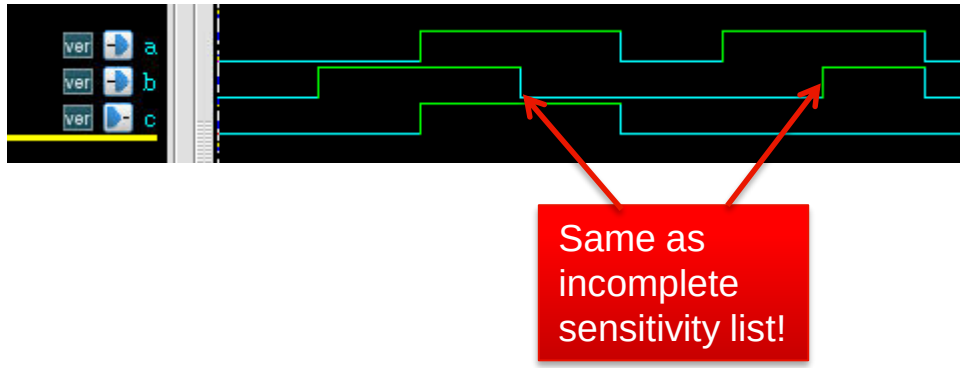
“Born to be wild(carded)”

- Functions can still break things:

```
wire b;

function and_with_b;
  input a_in;
  begin
    and_with_b = a_in & b; // b is global to the module here
  end
endfunction

always @(*) begin // b is left out of the sensitivity list
    c = and_with_b(a);
end
```



- Still rely on linters, etc.
- Better solution: Move to System Verilog!

System Verilog

“Well this could be the last time”

- System Verilog 2005 introduced always_comb
- Sensitivity:
  - Implicit sensitivity list (similar to always @*)
  - Correctly handles functions with global variables (better than always @*)

```
always_comb
  c = a & b;
```

Identical!

```
always @(a or b)
  c = a & b;
```

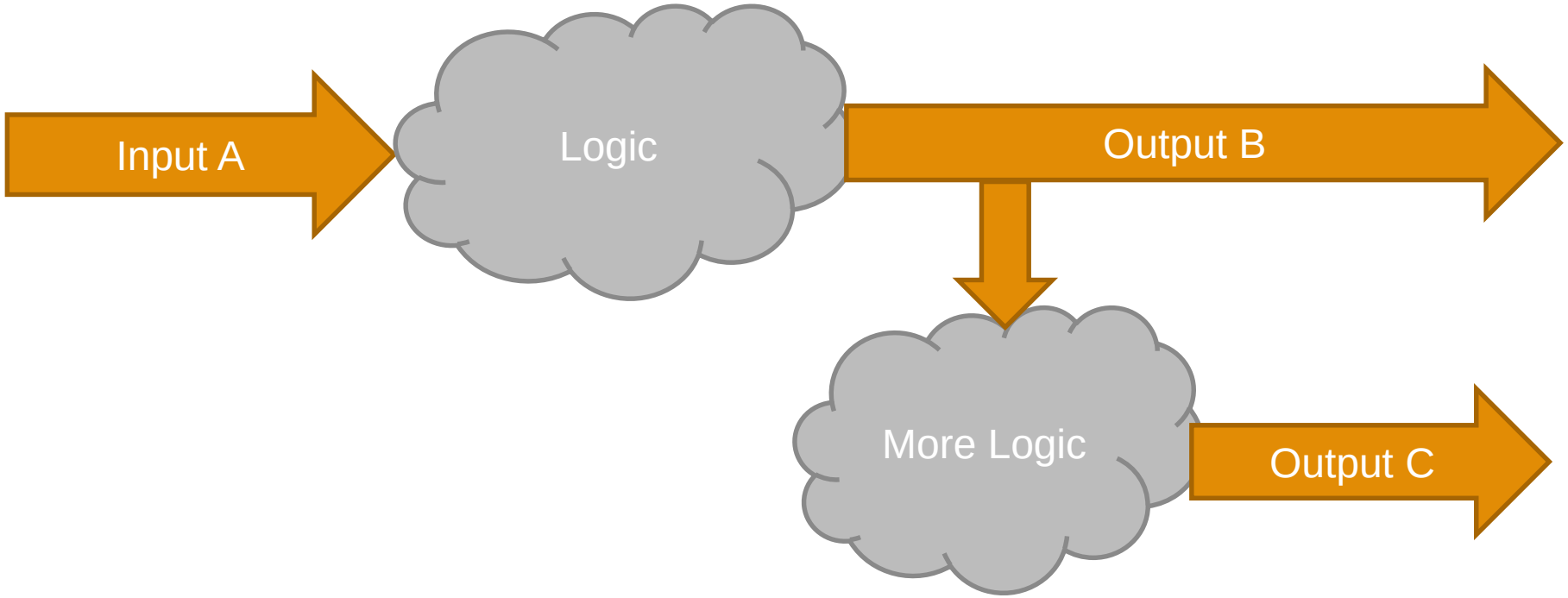
- All tools, including simulators, know that a block is supposed to be combinational, and may issue a warning or error if it’s not
- This is everything we want, right???

The problem with always_comb

"The implicit sensitivity list of an always_comb includes the expansions of the longest static prefix of each variable or select expression that is read within the block or within any function called within the block with the following exceptions: 1) any expansion of a variable declared within the block or within any function called within the block. 2) any expression that is also written within the block or within any function called within the block" – System Verilog LRM

"Why'd you have to go and make things so complicated?" – Avril Lavigne

Motivating Example



How do we code this?

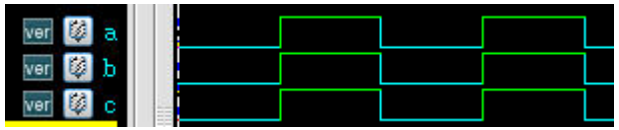
Motivating Example

- Assume both logic clouds are pass-through for simplest case:

```
always_comb begin
   b = a;
   c = b;
end
```

- What's the sensitivity list?
  - LRM says it **excludes** "any expression that is also written within the block"
  - Equivalent code:

```
always @(a) begin
  b = a;
  c = b;
end
```



This is still ok!

The Problem

Sequential always_comb!

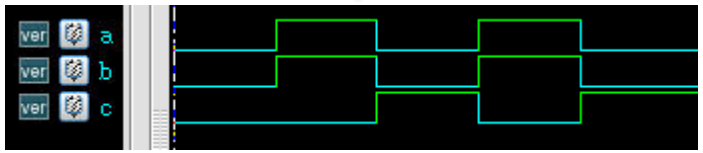
- What about this?

```
always_comb begin
   c = b;
   b = a;
end
```

BAD!!!

- What's the sensitivity list?

```
always @(a) begin
   c = b;
   b = a;
end
```



- Sequential logic in always_comb!!!

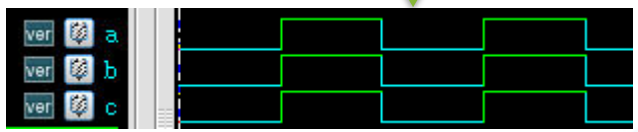
- Compare to v2k:

```
always @* begin
   c = b;
   b = a;
end
```

SAFE!

- What's the sensitivity list?

```
always @(a or b) begin
   c = b;
   b = a;
end
```



- Combinational logic in always @*!

Simulation Summary

- Simulation summary:

```
always @* begin
  b = a;
  c = b;
end
```

Combinational

```
always @* begin
  c = b;
  b = a;
end
```

Combinational

```
always_comb begin
  b = a;
  c = b;
end
```

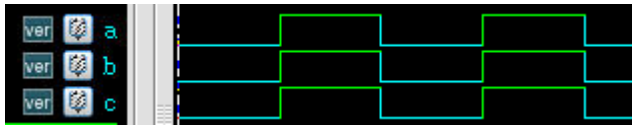
Combinational

```
always_comb begin
  c = b;
  b = a;
end
```

Sequential

What about Synthesis?

- As always, synthesis ignores sensitivity lists
- All 4 examples synthesize to simple wires:



- always_comb, supposed to solve the problem of RTL/gate mismatches, has created a whole new class of them!

Official mug of always_comb

Disclaimer!!!

- This presentation is not intended to encourage code like this:

```
always_comb begin
  c = b;
  b = a;
end
```

- Combinational always blocks have a definite order of execution, which makes things like this one-hot decoder possible:

```
always_comb begin
   foo = '0;
   foo[bar] = 1'b1;
End
```

- But always_comb should save us from these mistakes!

Disclaimer #2!!!

- This example is artificial and is reduced to the simplest possible problem case:

```
always_comb begin
  c = b;
  b = a;
end
```

- Real examples of this issue have been seen (and caught because the RTL behaved incorrectly) in much more complicated code involving case statements and multiple signals interacting
- Would have been much worse had RTL behaved correctly (designer missed inverter, for example) giving gates that behaved wrong!
- It is a real thing!

Solutions

"Stop talking about your problems and start thinking about solutions."
– Random inspirational quote found on the Internet

"Beer. Now there's a temporary solution."
– Homer Simpson

Solution 1 – Linters

Synopsys VC-Lint

- Linting has always been necessary to make always blocks safe
  - VC-lint rule SYN_9_2: Incomplete sensitivity list for v95-style
  - VC-lint rule VER_2_1_2_3: Function uses global variable for v2k-style
- Does VC-lint detect the always_comb issue?
  - VC-lint rule FM_2_36: Signal is read before being assigned
- After our first discovery of this problem, used VC-lint to check our other code
  - Found one instance in a design that had taped out
  - Luckily it used always @* so it was a non-issue
  - If we had been using always_comb without FM_2_36 we might have had a bug in silicon

Solution 1 – Linters

Other ways to Lint

- What about Spyglass?
  - No personal experience with Spyglass
  - I am told it can do this!
- Another approach: Ban always_comb altogether
  - My preferred method
  - Years of success with always @*, no need to introduce a dangerous replacement
  - Linters can do this also!

Solution 2 – Equivalence Checking

Synopsys Formality

- Formality catches this mismatch, but it can be overlooked!
- Test case, comparing these two modules:

```
// Golden
module always_test (
   input a,
   output logic c );

logic b;

  always_comb begin
     c = b;
     b = a;
  end

endmodule
```

```
// Revised
module always_test (
   input a,
   output logic c );

logic b;

  always @* begin
     c = b;
     b = a;
  end

endmodule
```

Only difference

Solution 2 – Equivalence Checking

Synopsys Formality

- Initial run through Formality gives this result:

```
Warning: variable 'b' read before write may cause simulation and synthesis
mismatch. (Signal: b Block: /always_test File: ./comb.v Line: 10)  (FMR_ELAB-117)
Error: Unsuppressed RTL interpretation message(s) :
       FMR_ELAB-117
       were produced during link. (FM-262)
Error: Failed to set top design to 'ref:/WORK/always_test' (FM-156)
```

- Great news! Formality recognized the problem and errored out
- What happens if this message is waived or downgraded to a warning for some reason?
  - Perhaps errors are temporarily downgraded to push design through flow, but never removed?

Solution 2 – Equivalence Checking

Synopsys Formality

- Actual Formality log for the preceding case:

```
1 Passing compare points
----------------------------------------------------------------------------
Matched Compare Points     BBPin    Loop    BBNet      Cut     Port      DFF      LAT    TOTAL
----------------------------------------------------------------------------
Passing (equivalent)           0       0        0        0        1        0        0        1
Failing (not equivalent)       0       0        0        0        0        0        0        0
****************************************************************************
Verification Successful
```

- Is Formality wrong?
- Formality gives you the option of shooting yourself in the foot. Don't waive those mismatch errors!

Summary

"Those who cannot remember the past are condemned to repeat it."
– George Santayana

Summary

- Different always blocks have always been susceptible to different synthesis/simulation mismatches
- always_comb is no different
- Signals on LHS and RHS in same always_comb block are the problem
- Linters and Formality can catch the problem
- The language committee's work is not done yet!

ORACLE

snug
Synopsys Users Group
Boston

Thank You