SystemVerilog's always_comb: Problems and Solutions

Matt Cohen

Oracle, Inc.

Burlington, MA, USA

www.oracle.com

ABSTRACT

Since the release of the IEEE 1800-2005 standard, SystemVerilog has been widely adopted by ASIC and FPGA design teams. SystemVerilog provides many useful features that greatly improve design and debug productivity – but it isn't perfect. This paper presents what the author believes is a fundamental flaw with the definition of the new *always_comb* procedural block. In addition, the paper illustrates how Synopsys tools, in particular VC-lint and Formality, can spot the problem with the language before it becomes a problem in your design.

Table of Contents

Table of Figures

1. Introduction

The history of the Verilog language, like all languages, is filled with attempts to fix common sources of user error. One particular construct is now on its third iteration. Initially the *always* block required an explicit sensitivity list. This was first replaced by the implicit sensitivity list (*always* @*), and then in SystemVerilog by *always_comb*. When it comes to eliminating synthesis/simulation mismatches, each is supposed to be better than the last, with *always_comb* supposedly solving the problem once and for all.

Unfortunately, *always_comb* introduces a new class of mismatches that were not present with *always* @*, a particularly insidious one as the simulated behavior often shows up as one extra inverter. If the Verilog source missed an inverter, the simulated behavior could be functionally correct, even though the synthesized gates are not functionally correct.

Synopsys tools such as VC-lint and Formality can be used to catch these mismatches. Formality will find it at the post-synthesis stage, while VC-lint can find the problem based on a static analysis of the RTL.

2. History of Procedural Control Statements

Before the flaws in SystemVerilog's *always_comb* can be discussed, it is necessary to have an understanding of the history of the problem that it is trying to solve, and the previous attempts at solutions.

2.1 Verilog-95 and *always*

The first Verilog standard (IEEE Std 1364-1995, usually called Verilog-95, [1]), defines one control construct (*always*) that can be used to create procedural statements for synthesizable logic (typically called an "*always* block"). To guarantee forward progress in a simulation, an infinite loop such as an *always* block requires a way to regulate its execution. The Verilog-95 spec shows delay controls (`#n`, where n is a number; typically called a "pound-delay"), but pound-delays are not synthesizable:

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

Instead of a delay control, an event control could be used. An event control is of the form `@(event_expression)`, and in the context of *always* statements has become known as a "sensitivity list". The simulator executes the *always* block each time a signal listed in the sensitivity list changes its value:

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

This is a correct, synthesizable way to produce an AND gate in Verilog-95. Synthesis tools such as Design Compiler will correctly produce an AND gate.

One of the big dangers with Verilog-95, though, was that it was very easy for an engineer to omit a signal from the sensitivity list. Code such as:

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

would not produce a functional AND gate in simulation. Figure 1 below shows the results of simulating the preceding always block. Changes in `b` have no effect on the output `c`.

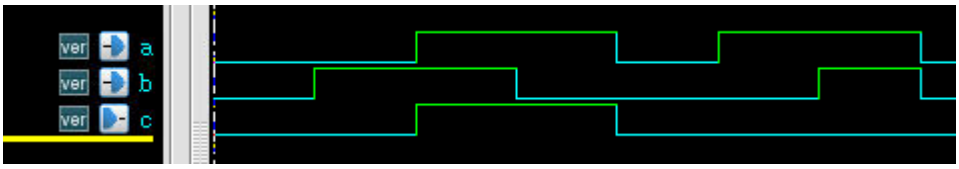

Figure 1. AND gate with one input missing from sensitivity list

Traditionally, synthesis tools ignore the sensitivity list. In the preceding example, Design Compiler would produce an AND gate. This behavior became standardized as part of the IEEE 1364.1 standard for Verilog RTL Synthesis [3], which says "An incomplete sensitivity list on a combinational always block will typically cause a mismatch between pre-synthesis and post-synthesis simulations." Since post-synthesis simulations are typically much less exhaustive than pre-synthesis ones, this kind of mismatch is often not caught until after tapeout.

A key strategy for dealing with sensitivity lists was the use of linters to guarantee that all relevant signals were included in the list. Using combinational *always* blocks in Verilog-95, without a linter, is extremely dangerous.

2.2 V2k and *always @**

The next major release of the Verilog standard, IEEE Std 1364-2001 (typically called V2k, [2]) sought to address the simulation/synthesis mismatch problem by introducing the "implicit event_expression", @*. This effectively acts a shortcut to help produce complete sensitivity lists. For example, code such as:

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

is treated the same as:

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

Simulators and synthesis tools agree on this interpretation and produce the same result. Figure 2 shows the simulation results, which clearly match a simple AND gate.

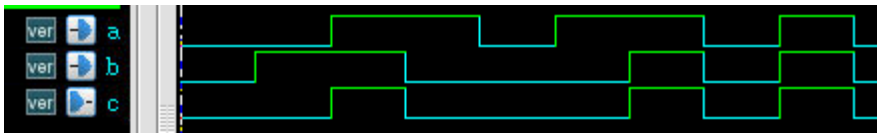

Figure 2. AND gate created by always @*

It is important to note exactly which variables and nets are and are not included in the effective sensitivity list. Included are:

- "All nets and variables which are read by the statement (which can be a statement group) of a `procedural_timing_control_statement` to the `event_expression`"
- "Nets and variables which appear on the right hand side of assignments, in function and task calls, or case and conditional expressions"

Excluded items are:

- "Identifiers which only appear in wait or event expressions"
- "Identifiers which only appear as a hierarchical_reg_identifier in the `reg_lvalue` of the left hand side of assignments"

The exclusions are not relevant for synthesizable RTL, which does not use wait or event expressions or hierarchical identifiers.

always @* is not perfect, however. If a Verilog function uses a global variable by name directly, instead of being passed as an input, that global variable is omitted from the effective sensitivity list. The following module:

```
module star_func (input a, b, output logic c);

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


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

endmodule // star_func
```

simulates identically to Figure 1.

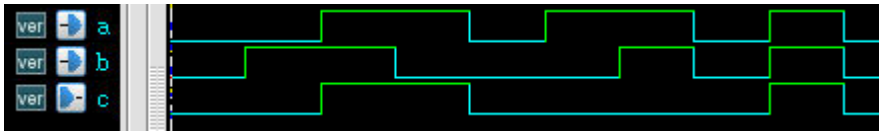

Figure 3. AND gate created with always @* and functions, input missing from effective sensitivity list

Once again, linters come to the rescue. Good design practice would have a linter rule enabled to

catch exactly this use of a function. Functions should in general be required to have all signals that are used be passed as actual inputs, rather than as module-global variables.

3. always_comb

To address the problems with *always* @*, as well as add some new enhancements, SystemVerilog introduced the *always_comb* block. *always_comb* provides new protections, ensuring that any variable written inside an *always_comb* is not written inside any other blocks, and it is guaranteed to execute at time 0, regardless of whether or not the sensitivity list is triggered. The problem with functions, global variables, and *always* @* is also successfully addressed by *always_comb*. *always_comb* further tells all tools that the user is writing combinational logic, and if anything is not combinational, an error or warning may be produced.

Like *always* @*, *always_comb* uses an implicit sensitivity list. The System Verilog Standard [4] reads:

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.

The 2nd item is what causes potential mismatches. Consider the very simple assignment:

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

`b`, having been written within the block, is omitted from the sensitivity list. The block is simulated as if it were:

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

In this simple example, that is ok. An update of `a` triggers the *always* block to execute, causing `b` to update first, followed by `b` updating `c`. The result in both cases is a wire that fans out from `a` to both `b` and `c` (see Figure 4).

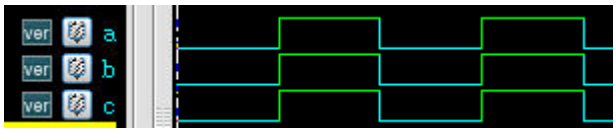

Figure 4. always_comb simulating in the expected way

When the order of assignments is reversed, *always* @* remains safe:

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

The effective sensitivity list is `(a or b)`. When `a` changes, the simulator will kick off the *always* block, leaving `c` unchanged but updating `b`. Once `b` updates, the *always* block is triggered again, and `c` updates, taking the new value of `b` (which is also the new value of `a`). The simulated behavior is once again a wire that fans out from `a` to both `b` and `c` (see Figure 5).

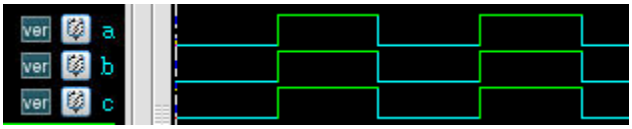

Figure 5. always @* simulating in the expected way

With *always_comb*, however, an interesting (and dangerous) thing happens:

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

The waves produced are shown in Figure 6.

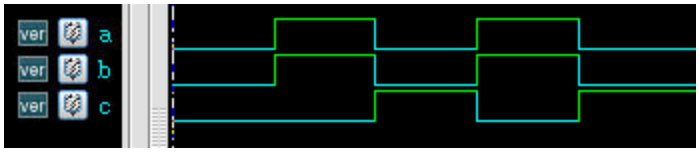

Figure 6. always_comb behaves unexpectedly!

How is this possible? Once again `b` is left out of the sensitivity list. As a result, the *always_comb* block executes whenever `a` is updated. However, the first assignment, from `b` to `c`, does nothing, since `b` has not yet changed. At the second line, `a` updates `b`, and then the *always_comb* block has completed execution. `c` is not updated until the next time the *always_comb* block is entered, which will happen on the next update of `a`. The resulting behavior is a wire from `a` to `b`, and a sequential element driving `c` with the previous value of `a`. If these are all single wires, `c` is essentially inverted with respect to `a` (after the first transition, before which they are equal).

Synthesis tools, as before, ignore the sensitivity list. All four of these examples synthesize into a simple wire from `a` to both `b` and `c`, even though one case behaves quite differently in simulation.

Although this example is trivial, the problem is not; I have seen its occurrence in many subtle ways, in much more complicated multi-level *if/else* and *case* statements. The example shown is merely the simplest possible instance of the problem. The following example is based on a real design at Oracle (signal names have been changed for this example):

```
always @( * ) begin
    case( config )
      `CONFIG_0: begin
          reset = reset0;
          req = req0;
          cmd = cmd0;
          updt = updt0;
          active = vld & active0;
          ctl = ctl0;
          vld = vld0;
    ...
end
```

Note that `active` is defined in terms of `vld`, which is assigned later on in the same *always* block, exactly the kind of problem introduced by *always_comb*. This particular example predated SystemVerilog and was therefore safe.

The issue is, I believe, a problem with the SystemVerilog Standard. Section 9.2.2.2.1 must be modified to remove the exclusion of "any expression that is also written within the block or within any function called within the block".

4. Linters to the Rescue

In the past, the shortcomings of both explicit sensitivity lists and *always* @* were solved by using a linter. A linter can easily tell if a signal is missing from an explicit sensitivity list, or if a function is vulnerable to the problem that affects *always* @* sensitivity lists. If a designer is linting all releases, these blocks are useable. However, *always_comb* provides additional benefits that the other options don't. Luckily, linters can also be used to solve the always_comb problem.

Synopsys linters include VC-lint and Spyglass. VC-lint includes a rule, SYN_9_2, that detects incomplete explicit sensitivity lists, and another rule, VER_2_1_2_3, that finds functions that use global variables. For the *always_comb* problem, VC-lint includes a rule, FM_2_36, that flags "Signal is read before being assigned". The default severity is a warning, but VC-lint allows all rules to have their severity levels modified. I strongly recommend enabling this rule, and elevating its severity to a level that would cause a release failure.

Our initial discovery of the *always_comb* behavior came in the work of a junior engineer who had been out of school for less than a year. One senior engineer suggested that the problem was due entirely to inexperience, and that more experienced engineers would not code this way. VC-lint in fact detected the same dangerous structure in that senior engineer's code. Because it was in an *always* @* block, the mismatch behavior did not occur in that situation, but VC-lint proved both its own usefulness and the widespread nature of the problem. Note that as with Verilog-95 and V2k, SystemVerilog users are still dependent on the quality of their linter to guarantee good RTL. The risks may have decreased, but they are still present.

5. Formal Equivalence

What happens when we run Synospys Formality to compare two designs that produce different simulated behavior, where one uses *always_comb* and the other is *always* @*?

The following two files are compared:

```
// begin comb.v
module always_test (
   input a,
   output logic c
   );

  logic   b;

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

// begin star.v
module always_test (
   input a,
   output logic c
   );

  logic   b;

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

The output, `c` is the only point to be compared by Formality. A careless user may push these files through Formality and see the following report:

```
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
```

Formality reports the two designs are equivalent. Is Formality wrong?

Well, kind of. The most basic simulation proves that. But to get to this point, the user has to suppress a warning message from Formality, one that causes an error in elaboration. Normally, Formality will produce this message:

```
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)
```

The warning is caused by hitting the *always_comb* mismatch issue, which causes an elaboration error. This message must be intentionally set to be ignored in order for verification to continue. Still, an inexperienced user, or one unaware of the mismatch problem, might quiet this warning message and proceed with the comparison, which can give invalid results. The engineer who knows about the seriousness of this problem can use Formality to catch it before tapeout.

6. Conclusions

always_comb is a useful construct, but one with a serious flaw. This flaw is not fatal, however, thanks to the availability of linters and formal equivalence checkers that are capable of alerting the user about dangerous code. As always, rigorous linting and investigation of all warning and error messages prove invaluable, allowing *always_comb* to be used safely.

In addition, I cannot stress strongly enough that I think the SystemVerilog Standard should be modified to prevent this behavior. Statement #2 in section 9.2.2.2.1 should be removed.

7. References

[1] IEEE Standard 1364-1995

[2] IEEE Standard 1364-2001

[3] IEEE Standard 1364.1-2002

[4] IEEE Standard 1800-2012