RTL Coding Styles That Yield
Simulation and Synthesis
Mismatches

Don Mills
LCDM Engineering

Clifford E. Cummings
Sunburst Design, Inc.

mills@lcdm-eng.com
cliffc@sunburst-design.com

ABSTRACT

This paper details, with examples, Verilog coding styles that will cause a mismatch between pre-
and post-synthesis simulations. Frequently, these mismatches are not discovered until after
silicon has been generated, and thus require the design to be re-submitted for a second spin.
Each coding style is accompanied by an example that shows the problem and an example of a
style that will match pre/post synthesis simulations. NOTE: Most of these coding styles also
apply to RTL models written in VHDL. In addition, a summary of Formality applied to each
style is included.
1.0 Introduction

The engineering task of converting a thought, an idea--or for the lucky ones, a specification--into a physical design is what ASIC and FPGA design is all about. The methodology of top down design requires transforming ideas from the abstract into a physical form that can be implemented and built. Developing concise, accurate designs entails learning how RTL coding styles synthesize and which styles can cause problems. This paper will discuss a number of HDL coding styles that cause mismatches between RTL and gate-level simulations. The basic premise is that any coding style that gives the HDL simulator information about the design that cannot be passed on to the synthesis tool is a bad coding style. Additionally, any synthesis switch that provides information to the synthesis tool that is not available to the simulator is bad.

If these guidelines are violated, the pre-synthesis RTL simulations will not match the post-synthesis gate level simulations. These mismatches can be very hard to detect if all possible logic combinations are not fully tested, and if not caught, are generally fatal to production ASICs. In addition, complete testing becomes impractical as the size of a design reaches into the millions of gates. The solution is to understand what coding styles or synthesis switches can cause RTL to gate level simulation mismatches and avoid these constructs.

1.1 Using Formality

Formality can be used to help find and isolate RTL to gate level mismatches. By default, Formality will interpret RTL code like a simulator would. Formality checks for simulation / synthesis mismatches when it reads in the RTL code. If Formality encounters any one of the mismatches listed below, it produces an error message and stops. The user has the ability to override the error and proceed with the verification. If the user does this, Formality will interpret the RTL code like synthesis does, resulting in a situation where Formality would evaluate two designs as equivalent, even though the RTL code simulates different from the gate design. Formality has (as do all formal verification tools) some limitations when working with very large designs.

2.0 SENSITIVITY LISTs

Synthesis tools infer combinational or latching logic from an always block with a sensitivity list that does not contain the Verilog keywords posedge or negedge. For a combinational always block, the logic inferred is derived from the equations in the block and has nothing to do with the sensitivity list. The synthesis tool will read the sensitivity list and compare it against the equations in the always block, only to report coding omissions that might cause a mismatch between pre- and post-synthesis simulations.

The presence of signals in a sensitivity list that are not used in the always block will not make any functional difference to either pre- or post-synthesis simulations. The only effect of
extraneous signals is that the pre-synthesis simulations will run more slowly. This is due to the fact that the \texttt{always} block will be entered and evaluated more often than is necessary.

2.1 Incomplete sensitivity list

The synthesized logic described by the equations in an \texttt{always} block will always be implemented as if the sensitivity list were complete. However, the pre-synthesis simulation functionality of this same \texttt{always} block will be quite different. In module \texttt{code1a}, the sensitivity list is complete; therefore, the pre- and post-synthesis simulations will both simulate a 2-input \texttt{and} gate. In module \texttt{code1b}, the sensitivity list only contains the variable \texttt{a}. The post-synthesis simulations will simulate a 2-input \texttt{and} gate. However, for pre-synthesis simulation, the \texttt{always} block will only be executed when there are changes on variable \texttt{a}. Any changes on variable \texttt{b} that do not coincide with changes on \texttt{a} will not be observed on the output. This functionality will not match that of the 2-input \texttt{and} gate of the post-synthesis model.

Finally, module \texttt{code1c} does not contain any sensitivity list. During pre-synthesis simulations, this \texttt{always} block will lock up the simulator into an infinite loop. Yet, the post-synthesis model will again be a 2-input \texttt{and} gate.

```verilog
module code1a (o, a, b);
    output o;
    input a, b;
    reg o;

    always @(a or b)
        o = a & b;
endmodule

module code1b (o, a, b);
    output o;
    input a, b;
    reg o;

    always @(a)
        o = a & b;
endmodule

module code1c (o, a, b);
    output o;
    input a, b;
    reg o;

    always
        o = a & b;
endmodule
```
2.2 Complete sensitivity list with mis-ordered assignments

Pre-synthesis assignments within an always block are executed sequentially. This becomes an issue when local temp variables are used in the always block. The temp variable could be used in the conditional part of an if statement, the case statement expression, or on the right hand side of an assignment statement. If a temp variable is used prior to being assigned, a mis-ordered assignment will result. Until the temp variable assignment statement is executed, temp will contain the value assigned to it during the previous pass through the always block.

In module code2a below, the object temp is read prior to being assigned. The value assigned to temp during the previous pass through the block will be used to determine the assignment to o. In the next line temp is assigned its new value corresponding to the current pass through the always block. During pre-synthesis simulation, temp will simulate as if it is latched. The value will be held for use during the next pass through the always block. This same code will synthesize as if the assignment order were listed correctly. This results in a mismatch between pre- and post-synthesis simulations. The code in module code2b shows the correct ordering which will result in pre- and post-synthesis simulations matching.

```verilog
module code2a (o, a, b, c, d);
  output o;
  input a, b, c, d;
  reg  o, temp;

  always @(a or b or c or d) begin
    o = a & b | temp;
  end
endmodule
```
```verilog
module code2b (o, a, b, c, d);
    output o;
    input  a, b, c, d;
    reg    o, temp;

    always @(a or b or c or d) begin
        temp = c & d;
        o = a & b | temp;
    end
endmodule
```

// Warning: Variable 'temp' is being read
// in routine code2a line 6 in file 'code2.v',
// but does not occur in the timing control of the
// block which begins there.

// Both designs infer an **and-or** gate (two 2-input
// **and** gates driving one 2-input **or** gate

---

### 3.0 Functions

Functions always synthesize to combinational logic. For this reason, some engineers choose to code all combinational logic using functions. As long as the coded function simulates like combinational logic, there is no problem using functions. The problem occurs when engineers make a mistake in the combinational function code and create simulation code that behaves like a latch. Since there are no synthesis tool warnings when function code simulates latch behavior, the practice of using functions to model synthesizable combinational logic is dangerous.

In the following example, module **code3a** shows a typical way to code a latch. When the same **if** statement is used inside a function, as shown in module **code3b**, the outcome is a 3-input **and** gate. If the code in a function is written to infer a latch, the pre-synthesis simulation will simulate the functionality of a latch, while the post-synthesis simulation will simulate combinational logic. Thus, the results from pre- and post-synthesis simulations will not match.

```verilog
module code3a (o, a, nrst, en);
    output o;
    input  a, nrst, en;
    reg    o;
```
always @(a or nrst or en)
  if (!nrst) o = 1'b0;
  else if (en) o = a;
endmodule

// Infers a latch with asynchronous low-true
// nrst and transparent high latch enable "en"

module code3b (o, a, nrst, en);
  output o;
  input a, nrst, en;
  reg o;

  always @(a or nrst or en)
    o = latch(a, nrst, en);

  function latch;
    input a, nrst, en;
    if (!nrst) latch = 1'b0;
    else if (en) latch = a;
  endfunction
endmodule

// Infers a 3-input and gate

Example 3.0 – Latch code in a function

4.0 CASE STATEMENTS

By default, Formality will interpret RTL code with the synthesis tool directives `//synopsys full_case` and `//synopsys parallel_case` like a simulator would. The user has the option to take a synthesis interpretation of the code by setting a couple of variables in the Formality tool. However, if the user does this, there is a potential for the problems described below in sections 4.1 and 4.2. For a more detailed explanation of this, see the notes on the Advanced Formality tutorial by Osman Eralp presented at the SJ SNUG 00.


4.1 Full Case

Using the synthesis tool directive `//synopsys full_case` gives more information about the design to the synthesis tool than is provided to the simulation tool. This particular directive is used to inform the synthesis tool that the `case` statement is fully defined, and that the output assignments for all unused cases are “don’t cares”. The functionality between pre- and post-synthesized designs may or may not remain the same when using this directive. Additionally,
although this directive is telling the synthesis tool to use the unused states as “don’t cares”, this
directive will sometimes make designs larger and slower than designs that omit the full_case
directive.

In module code4a, a case statement is coded without using any synthesis directives. The
resultant design is a decoder built from 3-input and gates and inverters. The pre- and post-
synthesis simulations will match. Module code4b uses a case statement with the synthesis
directive full_case. Because of the synthesis directive, the en input is optimized away
during synthesis and left as a dangling input. The pre-synthesis simulator results of modules
code4a and code4b will match the post-synthesis simulation results of module code4a, but
will not match the post-synthesis simulation results of module code4b.

```verilog
// no full_case
// Decoder built from four 3-input and gates
// and two inverters
module code4a (y, a, en);
  output [3:0] y;
  input [1:0] a;
  input en;
  reg [3:0] y;
  
  always @(a or en) begin
    y = 4'h0;
    case ({en,a})
      3'b1_00: y[a] = 1'b1;
      3'b1_01: y[a] = 1'b1;
      3'b1_10: y[a] = 1'b1;
      3'b1_11: y[a] = 1'b1;
    endcase
  end
endmodule

// full_case example
// Decoder built from four 2-input nor gates
// and two inverters
// The enable input is dangling (has been optimized away)
module code4b (y, a, en);
  output [3:0] y;
  input [1:0] a;
  input en;
  reg [3:0] y;
  
  always @(a or en) begin
    y = 4'h0;
    case ({en,a}) // synopsys full_case
      3'b1_00: y[a] = 1'b1;
  ```
4.2 Parallel Case

Using the synthesis tool directive 
\texttt{//synopsys parallel_case} gives more information about the design to the synthesis tool than is provided to the simulation tool. This particular directive is used to inform the synthesis tool that all cases should be tested in parallel, even if there are overlapping cases which would normally cause a priority encoder to be inferred. When a design does have overlapping cases, the functionality between pre- and post-synthesis designs will be different. In some cases, using this switch can also make designs larger and slower.

One consultant related the experience of adding \texttt{parallel_case} to an RTL design to improve optimized area and speed. The RTL model (behaving like a priority encoder) passed the test bench, but testing missed the flaw while simulating the gate-level model, which was implemented as non-priority parallel logic. Result: the design was wrong, the flaw was not discovered until ASIC prototypes were delivered, and the ASIC had to be redesigned at significant cost in both dollars and schedule.

The pre-synthesis simulations for modules \texttt{code5a} and \texttt{code5b} below, as well as the post-synthesis structure of module \texttt{code5a} will infer priority encoder functionality. However, the post-synthesis structure for module \texttt{code5b} will be two \texttt{and} gates. The use of the synthesis tool directive \texttt{//synopsys parallel_case} will cause priority encoder case statements to be implemented as parallel logic, causing pre- and post-synthesis simulation mismatches.

```verilog
// no parallel_case
// Priority encoder - 2-input \texttt{nand} gate driving an
// inverter (z-output) and also driving a
// 3-input \texttt{and} gate (y-output)
module code5a (y, z, a, b, c, d);
  output y, z;
  input a, b, c, d;
  reg y, z;

  always @(a or b or c or d) begin
    {y, z} = 2'b0;
    casez ({a, b, c, d})
      4'b11??: z = 1;
      4'b??11: y = 1;
  endcase
end
endmodule
```

Example 4.1 – Full Case

```verilog
3'b1_01: y[a] = 1'b1;
3'b1_10: y[a] = 1'b1;
3'b1_11: y[a] = 1'b1;
endcase
end
endmodule
```
endcase
end
endmodule

// parallel_case
// two parallel 2-input and gates
module code5b (y, z, a, b, c, d);
  output y, z;
  input a, b, c, d;
  reg y, z;

  always @(a or b or c or d) begin
    {y, z} = 2'b0;
    casez ({a, b, c, d}) // synopsys parallel_case
      4'b11?? : z = 1;
      4'b??11 : y = 1;
    endcase
  end
endmodule

Example 4.2 – Parallel Case

4.3 casex

The use of casex statements can cause design problems. A casex treats 'X's as "don't cares" if they are in either the case expression or the case items. The problem with casex occurs when an input tested by a casex expression is initialized to an unknown state. The pre-synthesis simulation will treat the unknown input as a "don't care" when evaluated in the casex statement. The equivalent post-synthesis simulation will propagate 'X's through the gate-level model, if that condition is tested.

One company related an experience they had with the use of casex in a design. The design went into a state where one of the inputs to a casex statement was unknown after the reset was released. Since the pre-synthesis RTL simulation treated the unknown input as a "don't care", the casex statement erroneously initialized the design to a working state. The gate-level simulation was not sophisticated or detailed enough to catch the error and consequently, the first turn of the ASIC came back with a serious flaw.

Module code6 below is a simple address decoder with an enable. Sometimes design errors in an external interface will cause the enable to glitch to an unknown state after initialization, before settling to a valid state. While the enable is in this unknown state, the case selector will erroneously match one of the case conditions, based on the value of addr. In the pre-synthesis design, this might mask a reset initialization problem that would only be visible in post-synthesis simulations. A similar situation could exist if the MSB of the address bus went unknown while
en is asserted. This would cause either memce0 or memcel to be asserted whenever the chip select (cs) signal should have been asserted.

Guideline: Do not use casex for RTL coding. It is too easy to match a stray unknown signal. It is better to use the casez statement as shown in the next section.

```verilog
module code6 (memce0, memcel, cs, en, addr);
    output memce0, memcel, cs;
    input en;
    input [31:30] addr;
    reg memce0, memcel, cs;

    always @(addr or en) begin
        {memce0, memcel, cs} = 3'b0;
        casex ({addr, en})
            3'b101: memce0 = 1'b1;
            3'b111: memcel = 1'b1;
            3'b0?1: cs = 1'b1;
        endcase
    end
endmodule
```

Example 4.3 - Casex Address Decoder

### 4.4 casez

The use of casez statements can cause the same design problems as casex, but these problems are less likely to be missed during verification. With casez, a problem would occur if an input were initialized to a high impedance state. However, the casez statement is a short, concise, and tabular method for coding certain useful structures, such as priority encoders, interrupt handlers, and address decoders. Therefore, the casez statement should not be completely dismissed from a design engineer’s repertoire of useful HDL coding structures.

Module code7 is the same simple address decoder with enable as shown in module code6 above, except that it uses a casez statement instead of the casex statement. The same problem described in Section 4.3 will occur when one of the inputs goes to a high-impedance state rather than an unknown state. Once again, an erroneous case match will occur, depending on the state of the other inputs to the case statement. However, it is less likely that a stray match will occur with a casez statement (floating input or tri-state driven signal) than with a casex statement (signal goes unknown briefly), but a potential problem does exist. Note that casez is useful for modeling address decoders and priority encoders.

Guideline: Use casez sparingly and cautiously for RTL coding since it is possible to match a stray tri-state signal.

```verilog
module code7 (memce0, memcel, cs, en, addr);
```
output     memce0, memcel1, cs;
input      en;
input [31:30] addr;
reg        memce0, memcel1, cs;

always @(addr or en) begin
  {memce0, memcel1, cs} = 3'b0;
  casez ({addr, en})
    3'b101: memce0 = 1'b1;
    3'b111: memcel1 = 1'b1;
    3'b0?1: cs    = 1'b1;
  endcase
end
endmodule

Example 4.3 - Casez Address Decoder

5.0 INITIALIZATION

5.1 Assigning ‘X’

When making assignments in RTL code, sometimes it is tempting to assign the ‘X’ value. The ‘X’ assignment is interpreted as an unknown by the Verilog simulator (with the exception of `casex` as previously discussed), but is interpreted as a “don't care” by synthesis tools. Making ‘X’ assignments can cause mismatches between pre- and post-synthesis simulations; however, the technique of making ‘X’ assignments can also be a useful trick. In FSM designs where there exist unused states, making an ‘X’ assignment to the state variable can help debug bogus state transitions. This is done by defaulting the next state registers to ‘X’ prior to entering the case statement, resulting in ‘X’ for any incorrect state transitions. Keep in mind that synthesis tools interpret unused ‘X’ state transitions as “don’t cares” for better synthesis optimization.

Modules `code8a` and `code8b` are simple Verilog models that implement 3-to-1 multiplexers. The coding style used in module `code8a` will give a simulation mismatch if the select lines ever take on the value of `2'b11`. The coding style used in module `code8b` will have no such mismatch. This mismatch can be valuable if the select line combination of `2'b11` is never expected. If this select line combination does occur, it will become obvious during simulation as the `y` output will be driven to an unexpected ‘X’ condition (which might facilitate debugging). However, if the design routinely and harmlessly transitions through the `2'b11` select-state, the first coding style will cause annoying simulation mismatches.

    // Note: the second example synthesizes to a smaller
    // and faster implementation than the first example.
module code8a (y, a, b, c, s);
  output y;
  input a, b, c;
  input [1:0] s;
  reg y;

  always @(a or b or c or s) begin
    y = 1'bx;
    case (s)
      2'b00: y = a;
      2'b01: y = b;
      2'b10: y = c;
    endcase
  end
endmodule

module code8b (y, a, b, c, s);
  output y;
  input a, b, c;
  input [1:0] s;
  reg y;

  always @(a or b or c or s)
    case (s)
      2'b00: y = a;
      2'b01: y = b;
      2'b10, 2'b11: y = c;
    endcase
endmodule

Example 5.1 – Initializing with ‘X’

5.2 Model initialization using translate_off / translate_on

This one seems so obvious that it should not require mention. However, an engineer related the following experience. He was using a state-machine tool that generated FSM code with variables initialized in an initial block, which was hidden from synthesis with the translate_on and translate_off synthesis directives. The pre-synthesis simulations ran fine, but the first production ASIC did not initialize properly, which required the ASIC to be redesigned. Module code9 shows the incorrect use of the directives translate_off/translate_on to initialize parts of the design. This will most likely cause the pre- and post-synthesis simulations to mismatch.
module code9 (y1, go, clk, nrst);
  output y1;
  input  go, clk, nrst;
  reg   y1;

  parameter IDLE = 1'd0,
             BUSY = 1'd1;

  reg [0:0] state, next;

  // Hiding the initialization of variables from the
  // synthesis tool is a very dangerous practice!!

  // synopsys translate_off
  initial y1 = 1'b1;
  // synopsys translate_on

  always @(posedge clk or negedge nrst)
    if (!nrst) state <= IDLE;
    else        state <= next;

  always @(state or go) begin
    next = 1'bx;
    y1 = 1'b0;
    case (state)
      IDLE: if (go)  next = BUSY;
      BUSY: begin
        if (!go) next = IDLE;
        y1 = 1'b1;
      end
    endcase
  end
endmodule

Example 5.2 - initialization using translate_off / translate_on

Formality does understand translate_off / translate_on compiler directives. By default, Formality sees the code as the synthesis tool does and not as the simulator does. The user has the option to have Formality take a simulation interpretation of the code by setting a couple of variables in the Formality tool.

6.0 General use of translate_off / translate_on

In general, the translate_off/translate_on synthesis directives should be used with caution. They are great when used to display information about a design, but they are dangerous when used to model functionality. One exception is the D flip-flop with both asynchronous reset
and set, where the typical coding style synthesizes and simulates the correct logic 100% of the
time; however during pre-synthesis simulation, it simulates the correct functionality 99% of the
time. This exception requires the use of non-synthesizable constructs to provide the correct pre-
synthesis model that accurately models and matches the post-synthesis model. This exception
condition is created as follows: assert reset, assert set, remove reset, leaving set still asserted. In
this case, the D flip-flop model needs a little assistance to correctly model the set condition
during pre-synthesis simulation. This is due to the always block only being entered on the
active edge of the set/reset. With both these inputs being asynchronous, the set should be active
once the reset is removed, but that will not be the case since there is no way to trigger the
always block. The fix to this problem is to model the flip-flop using the
translate_off/translate_on directive and force the output to the correct value for this
one condition. The best recommendation here is to avoid, as much as possible, the condition that
requires a flip-flop that uses an asynchronous set/reset.

Module code10a will simulate correctly 99% of the time (pre-synthesis). It has the flaw
described above. If the negedge is removed from the rstn and setn in the sensitivity list as
shown in module code10b, the design will not simulate correctly, either pre- or post-synthesis,
nor will it synthesize correctly. Finally, the code in module code10c shows the fix that will
simulate correctly 100% of the time, and will match pre- and post-synthesis simulations. This
code uses the translate_off/translate_on directives to force the correct output for the
exception condition described above.

```verbatim
// Generally good DFF with asynchronous set and reset
module code10a (q, d, clk, rstn, setn);  
output q;  
input  d, clk, rstn, setn;  
reg   q;  

always @(posedge clk or negedge rstn or negedge setn)  
  if (!rstn) q <= 0; // asynchronous reset  
  else if (!setn) q <= 1; // asynchronous set  
  else q <= d;  
endmodule

// synopsys translate_off
// Bad DFF with asynchronous set and reset. This design
// will not compile from Synopsys, and the design will
// not simulate correctly.
module code10b (q, d, clk, rstn, setn);  
output q;  
input  d, clk, rstn, setn;  
reg   q;  

always @(posedge clk or rstn or setn)  
  if (!rstn) q <= 0; // asynchronous reset  
  else if (!setn) q <= 1; // asynchronous set
```

//
else q <= d;
endmodule

// Good DFF with asynchronous set and reset and self-correcting
// set-reset assignment
module code10c (q, d, clk, rstn, setn);
output q;
input d, clk, rstn, setn;
reg q;

always @(posedge clk or negedge rstn or negedge setn)
if (!rstn) q <= 0; // asynchronous reset
else if (!setn) q <= 1; // asynchronous set
else q <= d;

// synopsys translate_off
always @(rstn or setn)
if (rstn && !setn) forc eq=1 ;
else release q;
// synopsys translate_on
endmodule

Example 7.0 - translate_off / translate_on

7.0 Timing Delays

An always block that does not schedule events in zero time could miss RTL- or behavioral-model triggered events. Adding timing delays to the left side of an assignment, as shown in module code11, will cause pre-synthesis simulations to differ from post-synthesis simulations. First, once the always block is entered due to a change on the sensitivity list variable in, subsequent changes on in will not cause re-entry until the always block is exited 65 time units later. Second, after a delay of 25 time units, the current value of in is read, inverted, and assigned to out1. After an additional 40 time units, in will again be read, inverted, and assigned to out2. During the timing delays, all other events on in will be ignored. The outputs will not be updated on every input change if changes happen more frequently than every 65 time units. The post-synthesis gate-level model will simulate two inverters while the pre-synthesis RTL code will miss multiple input transitions. Placing delays on the left side of always block assignments does not accurately model either RTL or behavioral models.

module code11 (out1, out2, in);
output out1, out2;
input in;
reg out1, out2;

always @(in) begin
  #25 out1 = ~in;
  #40 out2 = ~in;
end
endmodule

Example 7.0 - Timing Delays

Formality is not used to check timing. It cannot verify circuits such as an AND gate with a signal to one input and the signal through an inverter to the other input to produce a pulse. To Formality, this looks like a constant 0. Also, Formality cannot read the “#10 ...” delay syntax.

8.0 Conclusion

Knowing which coding styles can cause mismatches between pre- and post-synthesis simulations, understanding why mismatches might occur, and avoiding error-prone coding styles will greatly reduce RTL design flaws and the debug time necessary to fix the mismatches. As ASICs continue to increase in size, the ability to run 100% coverage regression tests after each synthesis turn is becoming more impractical. Designers must use every means available to reduce risk early in the design cycle.