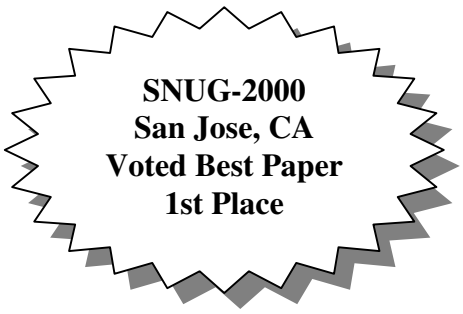

World Class Verilog & SystemVerilog Training

Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!

Clifford E. Cummings
Sunburst Design, Inc.
cliffc@sunburst-design.com
www.sunburst-design.com

ABSTRACT

One of the most misunderstood constructs in the Verilog language is the nonblocking assignment. Even very experienced Verilog designers do not fully understand how nonblocking assignments are scheduled in an IEEE compliant Verilog simulator and do not understand when and why nonblocking assignments should be used. This paper details how Verilog blocking and nonblocking assignments are scheduled, gives important coding guidelines to infer correct synthesizable logic and details coding styles to avoid Verilog simulation race conditions.

1.0 Introduction

Two well known Verilog coding guidelines for modeling logic are:

- Guideline: Use blocking assignments in always blocks that are written to generate combinational logic [1].
- Guideline: Use nonblocking assignments in always blocks that are written to generate sequential logic [1].

But why? In general, the answer is simulation related. Ignoring the above guidelines can still infer the correct synthesized logic, but the pre-synthesis simulation might not match the behavior of the synthesized circuit.

To understand the reasons behind the above guidelines, one needs to have a full understanding of the functionality and scheduling of Verilog blocking and nonblocking assignments. This paper will detail the functionality and scheduling of blocking and nonblocking assignments.

Throughout this paper, the following abbreviations will be used:
RHS - the expression or variable on the right-hand-side of an equation will be abbreviated as RHS equation, RHS expression or RHS variable.
LHS - the expression or variable on the left-hand-side of an equation will be abbreviated as LHS equation, LHS expression or LHS variable.

2.0 Verilog race conditions

The IEEE Verilog Standard [2] defines: which statements have a guaranteed order of execution ("Determinism", section 5.4.1), and which statements do not have a guaranteed order of execution ("Nondeterminism", section 5.4.2 & "Race conditions", section 5.5).

A Verilog race condition occurs when two or more statements that are scheduled to execute in the same simulation time-step, would give different results when the order of statement execution is changed, as permitted by the IEEE Verilog Standard.

To avoid race conditions, it is important to understand the scheduling of Verilog blocking and nonblocking assignments.

3.0 Blocking assignments

The blocking assignment operator is an equal sign ("="). A blocking assignment gets its name because a blocking assignment must evaluate the RHS arguments and complete the assignment without interruption from any other Verilog statement. The assignment is said to "block" other assignments until the current assignment has completed. The one exception is a blocking assignment with timing delays on the RHS of the blocking operator, which is considered to be a poor coding style [3].

Execution of blocking assignments can be viewed as a one-step process:

1. Evaluate the RHS (right-hand side equation) and update the LHS (left-hand side expression) of the blocking assignment without interruption from any other Verilog statement.

A blocking assignment "blocks" trailing assignments in the same always block from occurring until after the current assignment has been completed

A problem with blocking assignments occurs when the RHS variable of one assignment in one procedural block is also the LHS variable of another assignment in another procedural block and both equations are scheduled to execute in the same simulation time step, such as on the same clock edge. If blocking assignments are not properly ordered, a race condition can occur. When blocking assignments are scheduled to execute in the same time step, the order execution is unknown.

To illustrate this point, look at the Verilog code in Example 1.

```
module fbosc1 (y1, y2, clk, rst);
  output y1, y2;
  input  clk, rst;
  reg    y1, y2;

  always @(posedge clk or posedge rst)
    if (rst) y1 = 0;  // reset
    else     y1 = y2;

  always @(posedge clk or posedge rst)
    if (rst) y2 = 1;  // preset
    else     y2 = y1;
endmodule
```

Example 1 - Feedback oscillator with blocking assignments

According to the IEEE Verilog Standard, the two always blocks can be scheduled in any order. If the first always block executes first after a reset, both y1 and y2 will take on the value of 1. If the second always block executes first after a reset, both y1 and y2 will take on the value 0. This clearly represents a Verilog race condition.

4.0 Nonblocking assignments

The nonblocking assignment operator is the same as the less-than-or-equal-to operator ("<="). A nonblocking assignment gets its name because the assignment evaluates the RHS expression of a nonblocking statement at the beginning of a time step and schedules the LHS update to take place at the end of the time step. Between evaluation of the RHS expression and update of the LHS expression, other Verilog statements can be evaluated and updated and the RHS expression of other Verilog nonblocking assignments can also be evaluated and LHS updates scheduled. The nonblocking assignment does not block other Verilog statements from being evaluated.

Execution of nonblocking assignments can be viewed as a two-step process:
1. Evaluate the RHS of nonblocking statements at the beginning of the time step.
2. Update the LHS of nonblocking statements at the end of the time step.

Nonblocking assignments are only made to register data types and are therefore only permitted inside of procedural blocks, such as initial blocks and always blocks. Nonblocking assignments are not permitted in continuous assignments.

To illustrate this point, look at the Verilog code in Example 2.

```
module fbosc2 (y1, y2, clk, rst);
  output y1, y2;
  input  clk, rst;
  reg    y1, y2;

  always @(posedge clk or posedge rst)
    if (rst) y1 <= 0;  // reset
    else     y1 <= y2;

  always @(posedge clk or posedge rst)
    if (rst) y2 <= 1;  // preset
    else     y2 <= y1;
endmodule
```

Example 2 - Feedback oscillator with nonblocking assignments

Again, according to the IEEE Verilog Standard, the two always blocks can be scheduled in any order. No matter which always block starts first after a reset, both nonblocking RHS expressions will be evaluated at the beginning of the time step and then both nonblocking LHS variables will be updated at the end of the same time step. From a users perspective, the execution of these two nonblocking statements happen in parallel.

5.0 Verilog coding guidelines

Before giving further explanation and examples of both blocking and nonblocking assignments, it would be useful to outline eight guidelines that help to accurately simulate hardware, modeled using Verilog. Adherence to these guidelines will also remove 90-100% of the Verilog race conditions encountered by most Verilog designers.

Guideline #1: When modeling sequential logic, use nonblocking assignments.

Guideline #2: When modeling latches, use nonblocking assignments.

Guideline #3: When modeling combinational logic with an always block, use blocking assignments.

Guideline #4: When modeling both sequential and combinational logic within the same always block, use nonblocking assignments.

Guideline #5: Do not mix blocking and nonblocking assignments in the same always block.

Guideline #6: Do not make assignments to the same variable from more than one always block.

Guideline #7: Use $strobe to display values that have been assigned using nonblocking assignments.

Guideline #8: Do not make assignments using #0 delays.

Reasons for these guidelines are given throughout the rest of this paper. Designers new to Verilog are encouraged to memorize and use these guidelines until their underlying functionality is fully understood. Following these guidelines will help to avoid "death by Verilog!"

6.0 The Verilog "stratified event queue"

An examination of the Verilog "stratified event queue" (see Figure 1) helps to explain how Verilog blocking and nonblocking assignments function. The "stratified event queue" is a fancy name for the different Verilog event queues that are used to schedule simulation events.

The "stratified event queue" as described in the IEEE Verilog Standard is a conceptual model. Exactly how each vendor implements the event queues is proprietary, helps to determine the efficiency of each vendor's simulator and is not detailed in this paper.

As defined in section 5.3 of the IEEE 1364-1995 Verilog Standard, the "stratified event queue" is logically partitioned into four distinct queues for the current simulation time and additional queues for future simulation times.

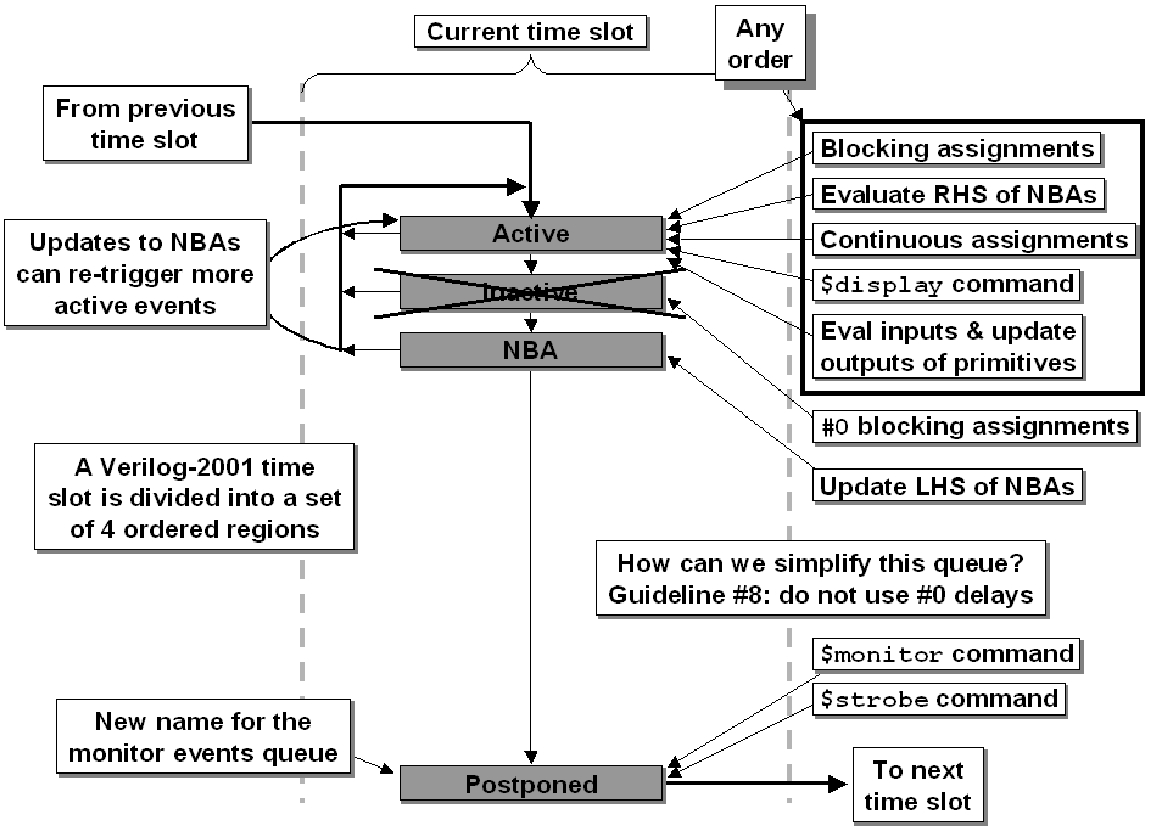

Figure 1 - Verilog "stratified event queue"

The *active events* queue is where most Verilog events are scheduled, including blocking assignments, continuous assignments, $display commands, evaluation of instance and primitive inputs followed by updates of primitive and instance outputs, and the evaluation of nonblocking RHS expressions. The LHS of nonblocking assignments are not updated in the *active events* queue.

Events are added to any of the event queues (within restrictions imposed by the IEEE Standard) but are only removed from the *active events* queue. Events that are scheduled on the other event queues will eventually become "activated," or promoted into the active events queue. Section 5.4 of the IEEE 1364-1995 Verilog Standard lists an algorithm that describes when the other event queues are "activated."

Two other commonly used event queues in the current simulation time are the *nonblocking assign updates* event queue and the *monitor* events queue, which are described below.

The *nonblocking assign updates* event queue is where updates to the LHS expression of nonblocking assignments are scheduled. The RHS expression is evaluated in random order at the beginning of a simulation time step along with the other *active events* described above.

The *monitor* events queue is where $strobe and $monitor display command values are scheduled. $strobe and $monitor show the updated values of all requested variables at the end of a simulation time step, after all other assignments for that simulation time step are complete. The IEEE Std 1800-2005 SystemVerilog Standard [4] renamed this queue to be the Postponed region.

A fourth event queue described in section 5.3 of the Verilog Standard is the *inactive* events queue, where #0-delayed assignments are scheduled. The practice of making #0-delay assignments is generally a flawed practice employed by designers who try to make assignments to the same variable from two separate procedural blocks, attempting to beat Verilog race conditions by scheduling one of the assignments to take place slightly later in the same simulation time step. Adding #0-delay assignments to Verilog models needlessly complicates the analysis of scheduled events. The author knows of no condition that requires making #0-delay assignments that could not be easily replaced with a different and more efficient coding style and hence discourages the practice.

Guideline #8: Do not make assignments using #0 delays.

The "stratified event queue" of Figure 1 will be frequently referenced to explain the behavior of Verilog code examples shown later in this paper. The event queues will also be referenced to justify the eight coding guidelines given in section 5.0.

7.0 Self-triggering always blocks

In general, a Verilog always block cannot trigger itself. Consider the oscillator example in Example 3. This oscillator uses blocking assignments. Blocking assignments evaluate their RHS expression and update their LHS value without interruption. The blocking assignment must complete before the **`@(clk)`** edge-trigger event can be scheduled. By the time the trigger event has been scheduled, the blocking **`clk`** assignment has completed; therefore, there is no trigger event from within the always block to trigger the **`@(clk)`** trigger.

```
module osc1 (clk);
  output clk;
  reg    clk;

  initial #10 clk = 0;

  always @(clk) #10 clk = ~clk;
endmodule
```

Example 3 - Non-self-triggering oscillator using blocking assignments

In contrast, the oscillator in Example 4 uses nonblocking assignments. After the first **`@(clk)`** trigger, the RHS expression of the nonblocking assignment is evaluated and the LHS value scheduled into the *nonblocking assign updates* event queue. Before the *nonblocking assign updates* event queue is "activated," the **`@(clk)`** trigger statement is encountered and the always block again becomes sensitive to changes on the **`clk`** signal. When the nonblocking LHS value is

updated later in the same time step, the `@(clk)` is again triggered. The osc2 example is self-triggering (which is not necessarily a recommended coding style).

```
module osc2 (clk);
  output clk;
  reg    clk;

  initial #10 clk = 0;

  always @(clk) #10 clk <= ~clk;
endmodule
```

Example 4 - Self-triggering oscillator using nonblocking assignments

8.0 Pipeline modeling

Figure 2 shows a block diagram for a simple sequential pipeline register. Example 5 - Example 8 show four different ways that an engineer might choose to model this pipeline using blocking assignments.

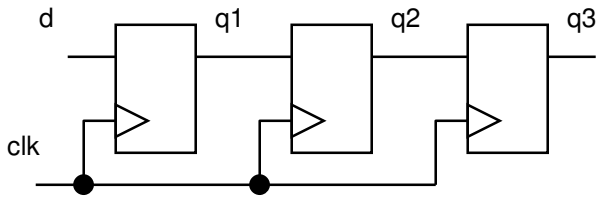

Figure 2 - Sequential pipeline register

In the pipeb1, Example 5 code, the sequentially ordered blocking assignments will cause the input value, **`d`**, to be placed on the output of every register on the next **`posedge clk`**. On every clock edge, the input value is transferred directly to the **`q3`**-output without delay. This clearly does not model a pipeline register and will actually synthesize to a single register! (See Figure 3).

```
module pipeb1 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) begin
    q1 = d;
    q2 = q1;
    q3 = q2;
  end
endmodule
```

Example 5 - Bad blocking-assignment sequential coding style #1

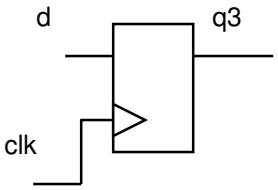

Figure 3 - Actual synthesized result!

In the pipeb2 example, the blocking assignments have been carefully ordered to cause the simulation to correctly behave like a pipeline register. This model also synthesizes to the pipeline register shown in Figure 2.

```
module pipeb2 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) begin
    q3 = q2;
    q2 = q1;
    q1 = d;
  end
endmodule
```

Example 6 - Bad blocking-assignment sequential coding style #2 - but it works!

In the pipeb3 example, the blocking assignments have been split into separate always blocks. Verilog is permitted to simulate the always blocks in any order, which might cause this pipeline simulation to be wrong. This is a Verilog race condition! Executing the always blocks in a different order yields a different result. However, this Verilog code will synthesize to the correct pipeline register. This means that there might be a mismatch between the pre-synthesis and post-synthesis simulations. The pipeb4 example, or any other ordering of the same always block statements will also synthesize to the correct pipeline logic, but might not simulate correctly.

```
module pipeb3 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) q1=d;

  always @(posedge clk) q2=q1;

  always @(posedge clk) q3=q2;
endmodule
```

Example 7 - Bad blocking-assignment sequential coding style #3

```
module pipeb4 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) q2=q1;
  always @(posedge clk) q3=q2;
  always @(posedge clk) q1=d;
endmodule
```

Example 8 - Bad blocking-assignment sequential coding style #4

If each of the four blocking-assignment examples is rewritten with nonblocking assignments, each will simulate correctly and synthesize the desired pipeline logic.

```
module pipen1 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) begin
    q1 <= d;
    q2 <= q1;
    q3 <= q2;
  end
endmodule
```

Example 9 - Good nonblocking-assignment sequential coding style #1

```
module pipen2 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) begin
    q3 <= q2;
    q2 <= q1;
    q1 <= d;
  end
endmodule
```

Example 10 - Good nonblocking-assignment sequential coding style #2

```
module pipen3 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) q1<=d;

  always @(posedge clk) q2<=q1;

  always @(posedge clk) q3<=q2;
endmodule
```

Example 11 - Good nonblocking-assignment sequential coding style #3

```
module pipen4 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) q2<=q1;

  always @(posedge clk) q3<=q2;

  always @(posedge clk) q1<=d;
endmodule
```

Example 12 - Good nonblocking-assignment sequential coding style #4

Upon examination of the pipeline coding styles shown in this section:

- 1 out of 4 blocking assignment coding styles was guaranteed to simulate correctly
- 3 out of 4 blocking assignment coding styles were guaranteed to synthesize correctly
- 4 out of 4 nonblocking assignment coding styles were guaranteed to simulate correctly
- 4 out of 4 nonblocking assignment coding styles were guaranteed to synthesize correctly.

Although, if confined to one always block with carefully sequenced assignments, it was possible to code the pipeline logic using blocking assignments. On the other hand, it was easy to code the same pipeline logic using nonblocking assignments; indeed, the nonblocking assignment coding styles all would work for both synthesis and simulation.

9.0 Blocking assignments & simple examples

There are many Verilog and Verilog synthesis books that show simple sequential examples that are successfully coded using blocking assignments. Example 13 shows a flipflop model that appears in most Verilog text books.

```
module dffb (q, d, clk, rst);
  output q;
  input  d, clk, rst;
  reg    q;

  always @(posedge clk)
    if (rst) q = 1'b0;
    else     q = d;
endmodule
```

Example 13 - Simple flawed blocking-assignment D-flipflop model - but it works!

If an engineer is willing to limit all modules to a single always block, blocking assignments can be used to correctly model, simulate and synthesize the desired logic. Unfortunately this reasoning leads to the habit of placing blocking assignments in other, more complex sequential always blocks that will exhibit the race conditions already detailed in this paper.

```
module dffx (q, d, clk, rst);
  output q;
  input  d, clk, rst;
  reg    q;

  always @(posedge clk)
    if (rst) q <= 1'b0;
    else     q <= d;
endmodule
```

Example 14 - Preferred D-flipflop coding style with nonblocking assignments

It is better to develop the habit of coding all sequential always blocks, even simple single-block modules, using nonblocking assignments as shown in Example 14.

Now consider a more complex piece of sequential logic, a Linear Feedback Shift-Register or LFSR.

10.0 Sequential feedback modeling

A Linear Feedback Shift-Register (LFSR) is a piece of sequential logic with a feedback loop. The feedback loop poses a problem for engineers attempting to code this piece of sequential logic with correctly ordered blocking assignments as shown in Example 15.

```
module lfsrb1 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;
  wire   n1;

  assign n1 = q1 ^ q3;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) begin
      q3 = 1'b1;
      q2 = 1'b1;
      q1 = 1'b1;
    end
    else begin
      q3 = q2;
      q2 = n1;
      q1 = q3;
    end
endmodule
```

Example 15 - Non-functional LFSR with blocking assignments

There is no way to order the assignments in Example 15 to model the feedback loop unless a temporary variable is used.

One could group all of the assignments into one-line equations to avoid using a temporary variable, as shown in Example 16, but the code is now more cryptic. For larger examples, one-line equations might become very difficult to code and debug. One-line equation coding styles are not necessarily encouraged.

```
module lfsrb2 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) {q3,q2,q1} = 3'b111;
    else        {q3,q2,q1} = {q2,(q1^q3),q3};
endmodule
```

Example 16 - Functional but cryptic LFSR with blocking assignments

If the blocking assignments in Example 15 and Example 16 are replaced with nonblocking assignments as shown in Example 17 and Example 18, all simulations function as would be expected from an LFSR.

```
module lfsrn1 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;
  wire   n1;

  assign n1 = q1 ^ q3;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) begin
      q3 <= 1'b1;
      q2 <= 1'b1;
      q1 <= 1'b1;
    end
    else begin
      q3 <= q2;
      q2 <= n1;
      q1 <= q3;
    end
endmodule
```

Example 17 - Functional LFSR with nonblocking assignments

```
module lfsrn2 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) {q3,q2,q1} <= 3'b111;
    else        {q3,q2,q1} <= {q2,(q1^q3),q3};
endmodule
```

Example 18 - Functional but cryptic LFSR with nonblocking assignments

Based on the pipeline examples of section 8.0 and the LFSR examples of section 10.0, it is recommended to model all sequential logic using nonblocking assignments. A similar analysis would show that it is also safest to use nonblocking assignments to model latches

Guideline #1: When modeling sequential logic, use nonblocking assignments.

Guideline #2: When modeling latches, use nonblocking assignments.

11.0 Combinational logic - use blocking assignments

There are many ways to code combinational logic using Verilog, but when the combinational logic is coded using an always block, blocking assignments should be used.

If only a single assignment is made in the always block, using either blocking or nonblocking assignments will work; but in the interest of developing good coding habits one should always using blocking assignments to code combinational logic.

It has been suggested by some Verilog designers that nonblocking assignments should not only be used for coding sequential logic, but also for coding combinational logic. For coding simple combinational always blocks this would work, but if multiple assignments are included in the always block, such as the and-or code shown in Example 19, using nonblocking assignments with no delays will either simulate incorrectly, or require additional sensitivity list entries and multiple passes through the always block to simulate correctly. The latter would be inefficient from a simulation time perspective.

The code shown in Example 19 builds the **`y`**-output from three sequentially executed statements. Since nonblocking assignments evaluate the RHS expressions before updating the LHS variables, the values of **`tmp1`** and **`tmp2`** were the original values of these two variables upon entry to this always block and not the values that will be updated at the end of the simulation time step. The **`y`**-output will reflect the old values of **`tmp1`** and **`tmp2`**, not the values calculated in the current pass of the always block.

```
module ao4 (y, a, b, c, d);
  output y;
  input  a, b, c, d;
  reg    y, tmp1, tmp2;

  always @(a or b or c or d) begin
    tmp1 <= a & b;
    tmp2 <= c & d;
    y    <= tmp1 | tmp2;
  end
endmodule
```

Example 19 - Bad combinational logic coding style using nonblocking assignments

The code shown in Example 20 is identical to the code shown in Example 19, except that **tmp1** and **tmp2** have been added to the sensitivity list. As describe in section 7.0, when the nonblocking assignments update the LHS variables in the *nonblocking assign update* events queue, the always block will self-trigger and update the **y**-outputs with the newly calculated **tmp1** and **tmp2** values. The **y**-output value will now be correct after taking two passes through the always block. Multiple passes through an always block equates to degraded simulation performance and should be avoided if a reasonable alternative exists.

```
module ao5 (y, a, b, c, d);
  output y;
  input  a, b, c, d;
  reg    y, tmp1, tmp2;

  always @(a or b or c or d or tmp1 or tmp2) begin
    tmp1 <= a & b;
    tmp2 <= c & d;
    y    <= tmp1 | tmp2;
  end
endmodule
```

Example 20 - Inefficient multi-pass combinational logic coding style with nonblocking assignments

A better habit to develop, one that does not require multiple passes through an always block, is to only use blocking assignments in always blocks that are written to model combinational logic.

```
module ao2 (y, a, b, c, d);
  output y;
  input  a, b, c, d;
  reg    y, tmp1, tmp2;

  always @(a or b or c or d) begin
    tmp1 = a & b;
    tmp2 = c & d;
    y    = tmp1 | tmp2;
  end
endmodule
```

Example 21 - Efficient combinational logic coding style using blocking assignments

The code in Example 21 is identical to the code in Example 19, except that the nonblocking assignments have been replaced with blocking assignments, which will guarantee that the **y**-output assumes the correct value after only one pass through the always block; hence the following guideline:

Guideline #3: When modeling combinational logic with an always block, use blocking assignments.

12.0 Mixed sequential & combinational logic - use nonblocking assignments

It is sometimes convenient to group simple combinational equations with sequential logic equations. When combining combinational and sequential code into a single always block, code the always block as a sequential always block with nonblocking assignments as shown in Example 22.

```
module nbex2 (q, a, b, clk, rst_n);
  output q;
  input  clk, rst_n;
  input  a, b;
  reg    q;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= a ^ b;
endmodule
```

Example 22 - Combinational and sequential logic in a single always block

The same logic that is implemented in Example 22 could also be implemented as two separate always blocks, one with purely combinational logic coded with blocking assignments and one with purely sequential logic coded with nonblocking assignments as shown in Example 23.

```
module nbex1 (q, a, b, clk, rst_n);
  output q;
  input  clk, rst_n;
  input  a, b;
  reg    q, y;

  always @(a or b)
    y = a ^ b;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= y;
endmodule
```

Example 23 - Combinational and sequential logic separated into two always blocks

Guideline #4: When modeling both sequential and combinational logic within the same always block, use nonblocking assignments.

13.0 Other mixed blocking & nonblocking assignment guidelines

Verilog permits blocking and nonblocking assignments to be freely mixed inside of an always block. In general, mixing blocking and nonblocking assignments in the same always block is a poor coding style, even if Verilog permits it.

The code in Example 24 will both simulate and synthesize correctly because the blocking assignment is not made to the same variable as the nonblocking assignments. Although this will work, the author discourages this coding style.

```
module ba_nba2 (q, a, b, clk, rst_n);
  output q;
  input  a, b, rst_n;
  input  clk;
  reg    q;

  always @(posedge clk or negedge rst_n) begin: ff
    reg tmp;
    if (!rst_n) q <= 1'b0;
    else begin
      tmp = a & b;
      q <= tmp;
    end
  end
endmodule
```

Example 24 - Blocking and nonblocking assignment in the same always block - generally a bad idea!

The code in Example 25 will most likely simulate correctly most of the time, but Synopsys tools will report a syntax error because the blocking assignment is assigned to the same variable as one of the nonblocking assignments. This code must be modified to be synthesizable.

```
module ba_nba6 (q, a, b, clk, rst_n);
  output q;
  input  a, b, rst_n;
  input  clk;
  reg    q, tmp;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q = 1'b0;  // blocking assignment to "q"
    else begin
      tmp = a & b;
      q <= tmp;            // nonblocking assignment to "q"
    end
endmodule
```

Example 25 - Synthesis syntax error - blocking and nonblocking assignment to the same variable

As a matter of forming good coding habits, the author encourages adherence to the following:

Guideline #5: Do not mix blocking and nonblocking assignments in the same always block.

14.0 Multiple assignments to the same variable

Making multiple assignments to the same variable from more than one always block is a Verilog race condition, even when using nonblocking assignments.

In Example 26, two always blocks are making assignments to the `q`-output, both using nonblocking assignments. Since these always blocks can be scheduled in an order, the simulation output is a race condition.

```
module badcode1 (q, d1, d2, clk, rst_n);
  output q;
  input  d1, d2, clk, rst_n;
  reg    q;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= d1;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= d2;
endmodule
```

Example 26 - Race condition coding style using nonblocking assignments

When Synopsys tools read this type of coding example, the following warning message is issued:

```
Warning: In design 'badcode1', there is 1 multiple-driver net with
unknown wired-logic type.
```

When the warning is ignored and the code from Example 26 is compiled, two flipflops with outputs feeding a 2-input and gate are inferred. The pre-synthesis simulation does not even closely match the post-synthesis simulation in this example.

Guideline #6: Do not make assignments to the same variable from more than one always block.

15.0 Common nonblocking myths

15.1 Nonblocking assignments & $display

Myth: "Using the $display command with nonblocking assignments does not work"

Truth: Nonblocking assignments are updated after all $display commands

```
module display_cmds;
  reg a;

  initial $monitor("\$monitor: a = %b", a);

  initial begin
    $strobe  ("\$strobe : a = %b", a);
    a = 0;
    a <= 1;
    $display ("\$display: a = %b", a);
    #1 $finish;
  end
endmodule
```

The below output from the above simulation shows that the **`$display`** command was executed in the *active events* queue, before the *nonblocking assign update* events were executed.

```
$display: a = 0
$monitor: a = 1
$strobe : a = 1
```

15.2 #0-delay assignments

Myth: “#0 forces an assignment to the end of a time step”

Truth: #0 forces an assignment to the "inactive events queue"

```
module nb_schedule1;
  reg a, b;

  initial begin
    a = 0;
    b = 1;
    a <= b;
    b <= a;

       $monitor ("%0dns: \$monitor: a=%b  b=%b", $stime, a, b);
       $display ("%0dns: \$display: a=%b  b=%b", $stime, a, b);
       $strobe  ("%0dns: \$strobe : a=%b  b=%b\n", $stime, a, b);
    #0 $display ("%0dns: #0      : a=%b  b=%b", $stime, a, b);

    #1 $monitor ("%0dns: \$monitor: a=%b  b=%b", $stime, a, b);
       $display ("%0dns: \$display: a=%b  b=%b", $stime, a, b);
       $strobe  ("%0dns: \$strobe : a=%b  b=%b\n", $stime, a, b);
       $display ("%0dns: #0      : a=%b  b=%b", $stime, a, b);

    #1 $finish;
  end
endmodule
```

The below output from the simulation on the previous page shows that the #0-delay command was executed in the *inactive events* queue, before the *nonblocking assign update* events were executed.

```
0ns: $display: a=0  b=1
0ns: #0      : a=0  b=1
0ns: $monitor: a=1  b=0
0ns: $strobe : a=1  b=0

1ns: $display: a=1  b=0
1ns: #0      : a=1  b=0
1ns: $monitor: a=1  b=0
1ns: $strobe : a=1  b=0
```

Guideline #7: Use $strobe to display values that have been assigned using nonblocking assignments.

15.3 Multiple nonblocking assignments to the same variable

Myth: “Making multiple nonblocking assignments to the same variable in the same always block is undefined”

Truth: Making multiple nonblocking assignments to the same variable in the same always block is defined by the Verilog Standard. The last nonblocking assignment to the same variable wins! Quoting from the IEEE 1364-1995 Verilog Standard [2], pg. 47, section 5.4.1 - Determinism

> "Nonblocking assignments shall be performed in the order the statements were executed. Consider the following example:
>
> ```
> initial begin
>  a <= 0;
>  a <= 1;
> end
> ```
>
> When this block is executed, there will be two events added to the nonblocking assign update queue. The previous rule requires that they be entered on the queue in source order; this rule requires that they be taken from the queue and performed in source order as well. Hence, at the end of time-step 1, the variable a will be assigned `0` and then `1`."

Translation: “Last nonblocking assignment wins!”

Summary of guidelines and conclusions

All of the guidelines detailed in this paper are listed below:

Guideline #1: When modeling sequential logic, use nonblocking assignments.

Guideline #2: When modeling latches, use nonblocking assignments.

Guideline #3: When modeling combinational logic with an always block, use blocking assignments.

Guideline #4: When modeling both sequential and combinational logic within the same always block, use nonblocking assignments.

Guideline #5: Do not mix blocking and nonblocking assignments in the same always block.

Guideline #6: Do not make assignments to the same variable from more than one always block.

Guideline #7: Use $strobe to display values that have been assigned using nonblocking assignments.

Guideline #8: Do not make assignments using #0 delays.

Conclusion: Following the above guidelines will accurately model synthesizable hardware while eliminating 90-100% of the most common Verilog simulation race conditions.

16.0 Errata and Changes

Readers are encouraged to send email to Cliff Cummings (cliffc@sunburst-design.com) any time they find potential mistakes or if they would like to suggest improvements.

16.1 Revision 1.3 (September 2009) - What Changed?

The original Figure 1 - Verilog "stratified event queue" was replaced with an updated diagram to show the interaction between the event regions. This new diagram forms the foundation for the updated event regions found in the SystemVerilog standard [4].

Errata - A colleague, Bill Dalton, pointed out that there was a typo at the end of section 8.0. The second to last paragraph has been corrected to state that, 4 out of 4 ***nonblocking*** assignment coding styles were guaranteed to synthesize correctly.

17.0 Final note: the spelling of "nonblocking"

The word nonblocking is frequently misspelled as "non-blocking." The author believes this is the Microsoftization of the word. Engineers have inserted a dash between "non" and "blocking" to

satisfy Microsoft, as well as other, spell checkers. The correct spelling of the word as noted in the IEEE 1364-1995 Verilog Standard is in fact: nonblocking.

References

[1] IEEE P1364.1 Draft Standard For Verilog Register Transfer Level Synthesis

[2] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, IEEE Computer Society, IEEE Std 1364-1995

[3] Clifford Cummings, "Correct Methods For Adding Delays To Verilog Behavioral Models," International HDL Conference 1999 Proceedings, pp. 23-29, April 1999.

[4] "IEEE Standard For SystemVerilog - Unified Hardware Design, Specification and Verification Language," IEEE Computer Society, IEEE, New York, NY, IEEE Std 1800-2005

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 27 years of ASIC, FPGA and system design experience and 17 years of SystemVerilog, synthesis and methodology training experience.

Mr. Cummings has presented more than 100 SystemVerilog seminars and training classes in the past six years and was the featured speaker at the world-wide SystemVerilog NOW! seminars.

Mr. Cummings has participated on every IEEE & Accellera SystemVerilog, SystemVerilog Synthesis, SystemVerilog committee, and has presented more than 40 papers on SystemVerilog & SystemVerilog related design, synthesis and verification techniques.

Mr. Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

Sunburst Design, Inc. offers World Class Verilog & SystemVerilog training courses. For more information, visit the www.sunburst-design.com web site.
Email address: cliffc@sunburst-design.com

An updated version of this paper can be downloaded from the web site: www.sunburst-design.com/papers
(Last updated September 8, 2009)