World Class SystemVerilog & UVM Training

SystemVerilog Logic Specific Processes for Synthesis - Benefits and Proper Usage

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

ABSTRACT

SystemVerilog added the new logic specific processes: always_comb, always_latch and always_ff. These new processes seem simple enough but they also include important simulation checks and there are subtle recommendations that should be followed when using these processes that supersede recommendations that existed with Verilog RTL synthesis coding guidelines.

There is also an issue related to synthesis that should be addressed by all synthesis and linting tool implementations. Tools currently issue warnings when they should issue errors.

This paper will describe advantages to using the new logic specific processes and offer supplemental coding guidelines to ensure that pre-synthesis simulations match post-synthesis implementations. This paper will also suggest proper error conditions that should be reported by synthesis and linting tools. This paper will also detail the proper usage of functions and tasks with the new logic specific processes.

Table of Contents

Table of Figures

Table of Examples

Table of Tables

1. Introduction

SystemVerilog added logic specific processes to show designer intent. SystemVerilog added other features to facilitate RTL design. The new SystemVerilog features motivated a new set of recommended RTL coding guidelines. This paper details the benefits of many new SystemVerilog features and proposes guidelines for their proper usage.

The new SystemVerilog features should have also encouraged simulation and synthesis vendors to do additional checking and reporting of errors in the SystemVerilog RTL code, but the added checks do not go as far as they could to help RTL coders remove errors from simulated and synthesized designs. Appendix 2 of this paper details additional checks and errors that should be reported by vendor tools.

RTL coders are encouraged to demand reasonable additional checks from their vendors to help improve the quality of front-end RTL designs.

2. Find and fix bugs as early as possible

The earlier a bug is identified and fixed, the faster a high quality design will be released to market, enabling successful companies to become more profitable.

0 includes a detailed list of design steps and the types of bugs that are typically discovered within each step.

An abbreviated list of techniques and places to identify and correct bugs is:

- Use a colorized code editor to help find misspelled keywords and unterminated strings.
- Use the simulation compilation tool to identify syntax and semantic errors (illegal coding styles that if permitted could cause bugs to go undetected until late in the design flow).
- Use simulation to find functional RTL bugs.
- Use synthesis compilation tools to identify coding styles that do not match expressed designer intent. Many linting tools can do this task if the engineer has access to linting tools.
- Use synthesized gate-level netlists to run simulations to identify a mismatch between pre- and post-synthesis simulations.

Finding bugs, fixing bugs, and exhaustively identifying all bugs early are all reasons to set guidelines and add SystemVerilog language features. If a bug can be identified at an earlier design stage, every effort should be made to make that happen. If you understand the importance of finding and fixing bugs early, you can skip 0 that describes in more detail where bugs can be found.

3. RTL SystemVerilog - 8 coding guidelines

Good coding guidelines help designers avoid mistakes and help identify problems as early as possible.

In the year 2000, I presented a paper with 8 Verilog RTL coding guidelines to help avoid mismatches between pre- and post-synthesis simulations[2]. Those guidelines, shown below, still apply to SystemVerilog RTL coding!

RTL Coding Guidelines

Guideline #1: Sequential logic - use nonblocking assignments

Guideline #2: Latches - use nonblocking assignments

Guideline #3: Combinational logic in an always block - use blocking assignments

Guideline #4: Mixed sequential and combinational logic in 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 *(This guideline is now enforced by the SystemVerilog compiler - see Section 6.1 for details)*

Display Guideline

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

General Guideline

Guideline #8: Do not make #0 procedural assignments

RTL coders that follow these guidelines will remove 90%-100% of all SystemVerilog race conditions from their simulations.

4. functions, void functions and tasks

Verilog had both **`functions`** and **`tasks`**. SystemVerilog added the exceptionally useful **`void function`**. A quick description of the functionality and utility of SystemVerilog **`tasks`**, **`functions`** and **`void functions`** is included here before their preferred usage is described starting in Section 5 and continuing through the rest of the paper.

Tasks are time-consuming subroutines. SystemVerilog **`void functions`** are 0-time subroutines. Verilog **`functions`** were sub-programs that returned a single value and the **`function`** had to be part of another expression.

When comparing **`functions`**, **`void functions`** and **`tasks`**, the following table shows some of the similarities and differences.

	Timing	Simulation Note	Synthesis Note
`task`	No delays	Legal	Legal, but now discouraged
	#delays	Legal	Ignored by synthesis
	`@(posedge clk)`	Legal	Synthesis error
	`wait`	Legal	Synthesis error
`function` & `void function`	No delays	Legal	Legal ***and*** encouraged
	#delays	Compiler error	Illegal
	`@(posedge clk)`	Compiler error	Illegal
	`wait`	Compiler error	Illegal

Table 1 - Comparing SystemVerilog tasks, functions and void functions

As can be seen in Table 1, **`tasks`** are allowed to use a variety of delays and triggers that are either ignored or cause synthesis errors. **`Functions`** are not allowed to have any type of delay or trigger and the simulation compiler catches these problems early in the RTL design cycle.

The point is, **`tasks`** allow RTL designers to add time-dependent constructs and to simulate code that can cause problems in synthesis, while **`functions`** cannot. The SystemVerilog **`void function`** does the same thing as an untimed **`task`** and hence **`tasks`** should no longer be used in SystemVerilog RTL synthesis.

4.1 task-issues with logic specific processes

`always_comb` and **`always_latch`** will examine the internal equations of a **`void function`** to correctly add signals to the RTL simulation sensitivity lists, while internal **`task`** signals are not examined. This is another enhancement that is made available to RTL coders that was not present with the Verilog **`always @*`**. This is described in greater detail in Section 7.1.

4.2 Void functions as documentation

`Tasks` are exceptionally useful as a verification subroutine that can consume clock cycles, but Verilog **`tasks`** have had very limited value when used in RTL synthesis and that is still true for SystemVerilog .

RTL synthesis tools prohibit the use of edge-triggering code, such as **`@(posedge clk)`**, inside of a **`task`**, and delays in **`tasks`** are completely ignored, so anything that is delay or trigger related in a **`task`** is either ignored or illegal.

The only value that **`tasks`** had in Verilog synthesis was as a subroutine to assign constant values to multiple signals in multiple places; **`tasks`** were essentially large macro-like assignments. If one wanted to assign variable values to the signals, the variable arguments had to be listed as **`task`** inputs to be automatically recognized and added to the sensitivity list of an **`always @*`** process. This was a significant limitation imposed by **`tasks`**.

5. New logic specific processes

Any language or tool feature that identifies problems earlier in the design flow is a feature that helps engineers shorten the design cycle and improve the quality of the design. Any new language feature that could help identify problems earlier in the design flow is a feature worthy of addition to the SystemVerilog language.

In general, **initial** processes are used by Verilog testbenches while **always** processes are used for Verilog design modeling. SystemVerilog added three new types of **always** processes that were intended to show designer intent and to enable tools to help RTL coders identify problems earlier in the design flow.

The three new logic specific, ***always_type*** processes are: **always_comb**, **always_latch** and **always_ff**. These processes are described in the remainder of this section, along with some of the new simulation compilation checks that are included with these ***always_type*** processes.

5.1 always_comb

always_comb is used to describe combinational logic and automatically builds the proper sensitivity list required by the simulator. In fact, the **always_comb** process builds a better sensitivity list than the Verilog-2001 **always @***. The improvements are described in Section 7.1.

Guideline: use **always_comb** to code all RTL combinational blocks and quit using **always @***.

5.2 always_latch

always_latch is used to describe latch-based logic and also automatically builds the proper sensitivity list required by the simulator. Just like the **always_comb** process, **always_latch** builds a better sensitivity list than the Verilog-2001 **always @***.

Guideline: when doing latch-based design, use **always_latch** to code all RTL latch blocks and quit using **always @***.

5.3 always_ff

always_ff is used to describe clocked logic but it does NOT automatically build the proper sensitivity list required by the simulator.

The **always_comb** and **always_latch** processes can determine the correct sensitivity list from the code within those processes. All of the information necessary to generate a correct sensitivity list is available in the equations within those processes, but that is not true for **always_ff**.

Examine the **always_ff** RTL code for a flip-flop as shown in Example 1.

```
module dff1 (
  output bit_t q,
  input  bit_t d, clk, rst_n);

  always_ff @(posedge clk, negedge rst_n)
    if (!rst_n) q <= 0;
    else        q <= d;
endmodule
```

Example 1 - always_ff flip-flop with asynchronous low-true reset

- There is no information in the **always_ff** process to identify the name of the clock.
- There is no information in the **always_ff** process to identify the polarity of the clock.
- There is no information in the **always_ff** process to identify if the reset is synchronous or asynchronous.

For these reasons, the **always_ff** block still needs a sensitivity list.

Guideline: use **always_ff** to code all RTL clocked blocks and quit using **always @(posedge clk ...)**.

6. New checks using always_type processes

One of the great advantages of using the new ***always_type*** processes is that the simulation compiler will identify and issue errors for improper coding styles that are not necessarily related to logic that would be inferred by synthesis tools. The new errors that are identified could cause mismatches between pre-and post-synthesis simulations. This means that some important errors will be caught much earlier than using older Verilog techniques before wasting time running invalid simulations and synthesis. The additional checks are described below.

6.1 Multiple processes assigning to the same varialble is illegal

New check - RTL code that uses the new ***always_type*** processes - making assignments to the same variable from more than one ***always_type*** process is now illegal.

For years, all synthesis tools have given errors if the same output variable was assigned from more than one Verilog **always** process, but simulators allowed the same coding style to be simulated. This meant that an engineer could write and simulate illegal RTL code and not discover the problem until synthesis. In other words, the problem would be found but it would be found late in the design flow after wasting significant time doing simulations of illegal RTL code.

Consider the d-flip-flop code in Example 2. This code will simulate, albeit with a race condition and questionable results. Both **always** processes make assignments to the same **q** variable, but there is no guarantee which **always** process will finish last. During simulation, whichever assignment finishes last will make the final assignment seen in a waveform display. As can be seen in Figure 1, VCS has assigned the **q**-variable from the first **always** block followed by the second **always** block. This simulation behavior is legal but not guaranteed. Another simulator or even a different simulation might cause the first **always** process assignment to execute last, which would give different results.

```
module dff_2a (
  output logic q,
  input  logic d1, d2, clk, rst_n);

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= '0;
    else        q <= d1;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= '0;
    else        q <= d2;
endmodule
```

Example 2 - Multi-driver SystemVerilog model - always processes - simulates (poorly) - does not synthesize

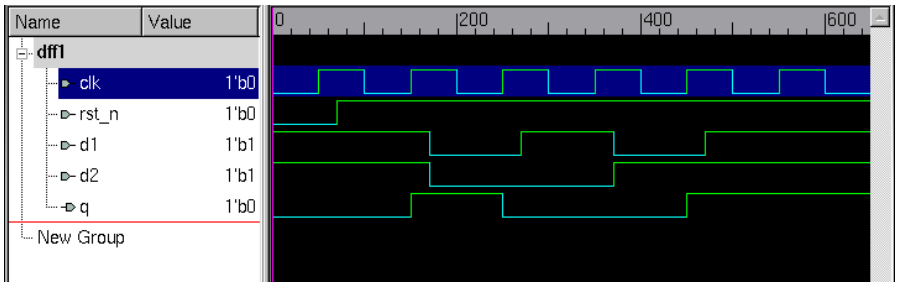

Figure 1 - Multi-driver simulation waveform (bad results)

For this reason, RTL Coding Guideline #6 of Section 3 states: Do not make assignments to the same variable from more than one always block. Ignoring this guideline causes the simulation behavior to be nondeterministic.

When the Example 2 code is read into Design Compiler (DC), the error message shown in Figure 2 is properly reported.

```
Error:  .../dff_2a.sv:9: Net 'q' or a directly connected net is driven by
more than one source, and not all drivers are three-state. (ELAB-366)
*** Presto compilation terminated with 1 errors. ***
Error: Can't read 'sverilog' file '.../dff_2a.sv'. (UID-59)
```

Figure 2 - Design Compiler multi-driver error message

The point is, using Verilog **always** processes allows the designer to construct flawed RTL code that might not be discovered until simulation (if the race condition is recognized from the waveform display), or until read into DC. Finding the bug during simulation or while reading code into synthesis tools is needlessly late in the design flow. Designers prefer to find this type of mistake earlier.

What was once just a guideline is now enforced by the SystemVerilog simulation compiler, allowing the RTL designer to find this bug very early in the design flow.

```
module dff_2b (
  output logic q,
  input  logic d1, d2, clk, rst_n);

  always_ff @(posedge clk or negedge rst_n)
    if (!rst_n) q <= '0;
    else        q <= d1;

  always_ff @(posedge clk or negedge rst_n)
    if (!rst_n) q <= '0;
    else        q <= d2;
endmodule
```

Example 3 - Multi-driver SystemVerilog model - always_ff processes - fails simulation compilation

The SystemVerilog RTL code of Example 3 is exactly the same as the RTL code of Example 2, except the **always** processes have been replaced with **always_ff** processes. Any attempt to compile this model for simulation will fail and cause an error message to be reported. The error message for this model as reported by VCS is shown in Figure 3.

```
Error-[ICPD] Illegal combination of drivers
dff_2b.sv, 2
  Illegal combination of procedural drivers
  Variable "q" is driven by an invalid combination of procedural drivers.
  Variables written on left-hand of "always_ff" cannot be written to by any
  other processes, including other "always_ff" processes.
  This variable is declared at "dff_2b.sv", 2: logic q;
  The first driver is at "dff_2b.sv", 9: always_ff @(posedge clk or negedge
  rst_n) if (!rst_n) begin
  q <= '0;
  The second driver is at "dff_2b.sv", 5: always_ff @(posedge clk or
negedge
  rst_n) if (!rst_n) begin
  q <= '0;
   ...
```

Figure 3 - VCS error messages for illegal same-variable assignments from more than one always_ff process

So what happens if assignments are made to the same variable from a SystemVerilog **always_ff** process and a Verilog **always** process as shown in Example 4?

```
module dff_2c (
  output logic q,
  input  logic d1, d2, clk, rst_n);

  always_ff @(posedge clk or negedge rst_n)
    if (!rst_n) q <= '0;
    else        q <= d1;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= '0;
    else        q <= d2;
endmodule
```

Example 4 - Multi-driver SystemVerilog model - always_ff & always processes

When assignments are made to a variable from any SystemVerilog ***always_type*** process and also assigned from a Verilog **always** process, the simulation compilation will fail. VCS reports the error as shown in Figure 4.

```
Error-[ICPD] Illegal combination of drivers
dff_2c.sv, 2
  Illegal combination of procedural drivers
  Variable "q" is driven by an invalid combination of procedural drivers.
  Variables written on left-hand of "always_ff" cannot be written to by any
  other processes, including other "always_ff" processes.
  This variable is declared at "dff_2c.sv", 2: logic q;
  The first driver is at "dff_2c.sv", 10: q <= '0;
  The second driver is at "dff_2c.sv", 5: always_ff @(posedge clk or
negedge
  rst_n) if (!rst_n) begin
  q <= '0;
   ...
```

Figure 4 - VCS error message for illegal same-variable assignments from always_ff & always processes

Guideline: Use the new ***always_type*** processes to help identify flawed RTL coding styles earlier and quit using the older Verilog **always** process.

6.2 Blocking delay assignments are illegal

New check - RTL code that uses the new ***always_type*** processes - making blocking delay assignments to variables is now illegal. [1] [2]

Synthesis tools ignore all delays in RTL code, but it was possible using the Verilog **always** process to add blocking delays to the RTL code, run Verilog simulations, and cause the pre-synthesis simulation to behave incorrectly. This meant that an engineer could code and simulate illegal RTL code and not discover the functional problem until post-synthesis simulations. In other words, the problem would only be found very late in the design flow after wasting significant time doing simulations and synthesis of flawed RTL code, and finding the problem required designers to run gate-level simulations to discover that the pre-synthesis functionality simulation did not match the post-synthesis simulation.

For the flip-flop design with asynchronous low-true **rst_n** shown in Example 5, the RTL code includes **#2** blocking delays placed in front of the assignments to the **q** variable. In the timing diagram for this example, shown in Figure 5, when **clk** goes high, there is no active-low **rst_n**, so the RTL code will immediately jump to the **else** branch of the code. The **else** branch must first wait for the **#2** delay, and then will assign the **d**-input to the **q**-output. The problem with this example is that the **rst_n** signal went active-low **#1** after the **posedge clk**, but since the simulator is stuck waiting for 2 nanoseconds before executing the **else** branch, the active **rst_n** is missed until the next **posedge clk** during the RTL simulation. This means that the RTL simulation is wrong and might not be discovered until post-synthesis gate-level simulations.

```
module dff1a (
  output logic q,
  input  logic d, clk, rst_n);

  always @(posedge clk, negedge rst_n)
    if (!rst_n) #2 q <= '0;
    else        #2 q <= d;
endmodule
```

Example 5 - dff model using always process with #2 delays - simulation problems

[1] Blocking statements are not the same as blocking assignments. Blocking statements refer to statements that can force the simulation time to advance before allowing the subsequent statements to be executed, including #delays and calls to procedures that could block such as **task** calls that contain blocking delays.

[2] Delays on the right-hand-side of nonblocking assignments are not blocking delays and are therefore will legal when using ***always_type*** processes.

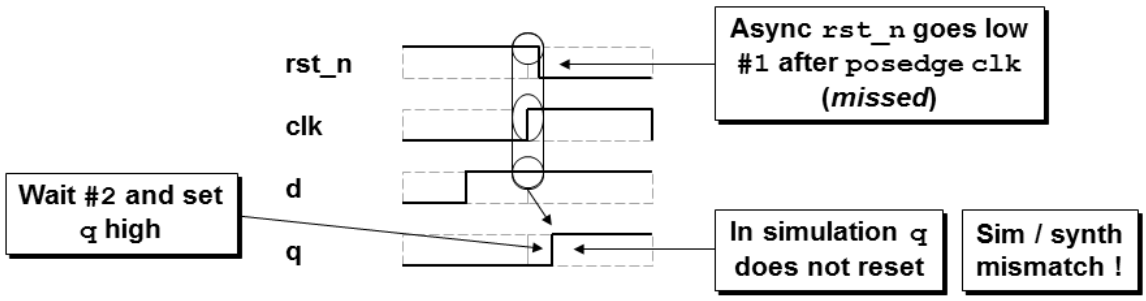

Figure 5 - Timing diagram of RTL flip-flop with missed reset

Note that synthesis tools ignore delays in RTL code and would infer the correct flip-flop with asynchronous reset, which will not match the pre-synthesis simulation. This is an error that will not be caught until gate-simulations fail to match pre-synthesis RTL simulations or until equivalence checking tools mathematically prove that the RTL does not match the inferred gate-level representation. That is undesirably late in the design flow.

If the Verilog **always** process is replaced by the SystemVerilog **always_ff** process, the included blocking delays to the RTL code as shown in Example 6 will be flagged as an error during simulation compilation, and thus the error will be identified very early in the design flow.

```
module dff1c (
  output logic q,
  input  logic d, clk, rst_n);

  always_ff @(posedge clk, negedge rst_n)
    if (!rst_n) #2 q <= '0;
    else        #2 q <= d;
endmodule
```

Example 6 - dff model using always_ff process with #2 delays - properly fails to compile

VCS issues the error message shown in Figure 6, which indicates that the blocking delay is illegal inside of the **always_ff** process.

```
Error-[SV-BCACF] Blocking construct in always_comb/ff
dff1c.sv, 6
"#(2) q <= 0;"
Statements in an always_comb shall not include those that block, have
blocking timing or event controls, or forkjoin statements. The always_ff
procedure imposes the restriction that it contains one and only one event
control and no blocking timing controls.
Try using simple always blocks.
```

(NOTE: this error message is repeated for line 8)

Figure 6 - VCS error message when blocking delays are added to always_type processes

7. New advantages to using always_type processes

In addition to the two design checks described in Section 6, the new ***always_type*** processes offer additional design advantages described in the remainder of this Section.

7.1 Improved automatic sensitivity list creation

Verilog-2001 added the ability to code an **always** process with the **@*** sensitivity list. The motive behind this enhancement was to allow the simulation and synthesis tools to automatically create a complete sensitivity list for combinational and latch based logic. The one shortfall with the **@*** sensitivity list is that it required all tasks and function calls within the scope of the **always** process to list the task/function calling arguments as part of the task/function port definitions to allow **@*** to find those signals to add to the sensitivity list. Any signal used within a task/function that was not part of the argument declarations for the task and function headers would be missed and not added to the sensitivity list. This is illustrated in Figure 7.

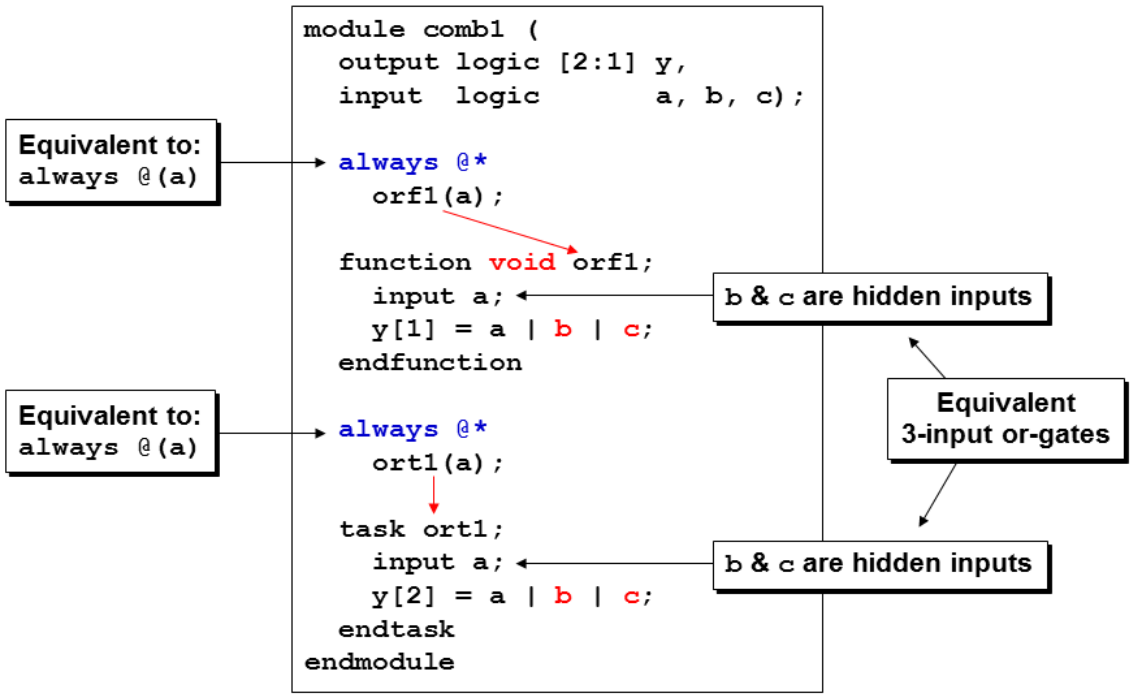

Figure 7 - always process sensitivity list generation when calling tasks and functions

The Figure 7 example includes both a **void function** and a **task** that represent 3-input **or**-gates, but only the **a**-input is listed in the **function** and **task** headers while the **b** and **c**-inputs are read directly from the module scope. The **always @*** processes will only examine the headers of the **function** and **task** and will therefore omit the **b** and **c**-inputs from the process sensitivity lists. The pre-synthesis simulations will not match the post-synthesis simulations.

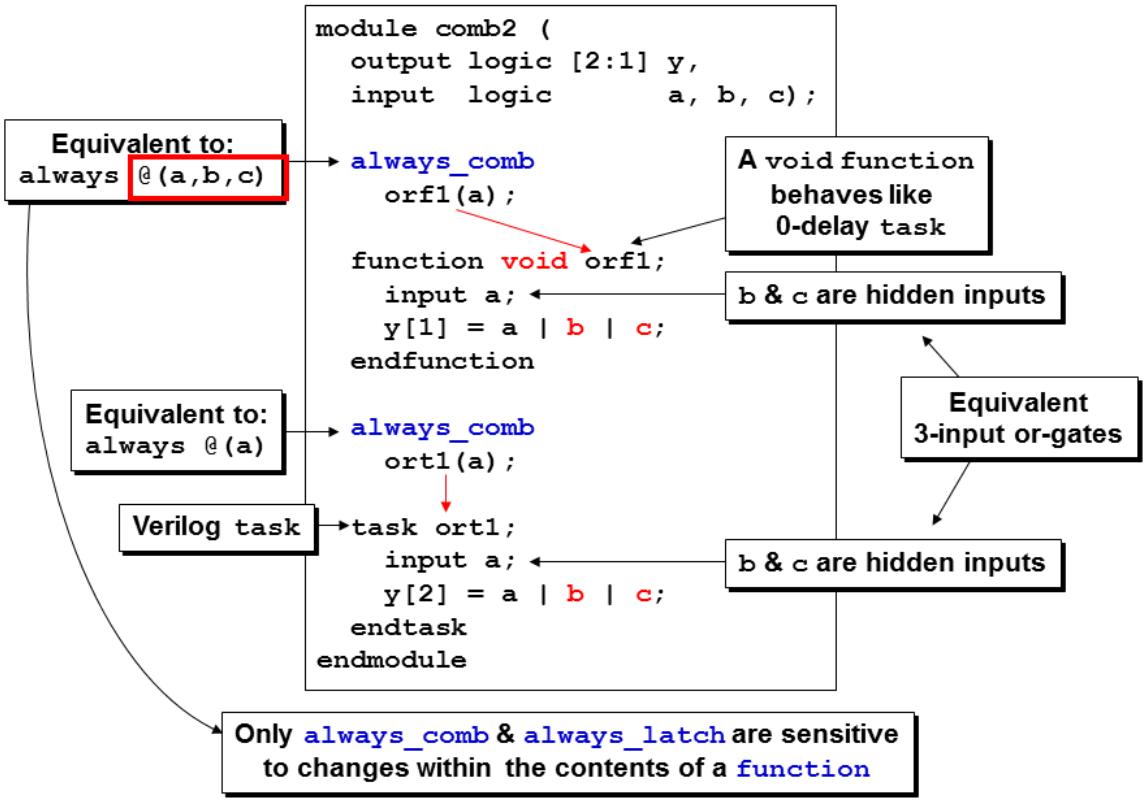

Figure 8 - always_type process sensitivity list generation when calling tasks and functions

In Figure 8, the same two **function** and **task** 3-input **or**-gates exist, but the **always** processes have been replaced with **always_comb** processes, which allows the **always_comb** to build the full and proper sensitivity list for the process that calls the **void function** while still not building the full sensitivity list for the process that calls the **task**.

The reason that calling a **task** does not build the full sensitivity list from the internal **task** signals is because a **task** can consume time and it may not even make sense to trigger on all of the internal **task** signals since they may not change until later in the simulation.

Synthesis tools ignore delays and do not permit **tasks** to include any internal triggering signals such as **@(posedge clk)**. Since delays in **tasks** are either ignored or illegal when doing synthesis, the preferred SystemVerilog solution is to replace all **tasks** with **void functions**. A **void function** is basically a 0-delay **task** call with the advantage that internal **function** signals are added to an **always_comb** sensitivity list.

7.2 Ability to call void functions without argument lists

The ability to create a full sensitivity list from **void functions** with all internal function signals removed from the header actually allows another very interesting and productive coding style in SystemVerilog.

Consider the large combinational logic block shown on the left side in Figure 9. If the combinational logic does indeed require 44 lines of code (or more) to describe the combinational functionality of this block, this code could be quite difficult to understand and maintain. One solution is to split the code into smaller **always_comb** blocks as shown on the right side of Figure 9, but splitting the code

does not necessarily document the code any better than the one huge **always_comb** process.

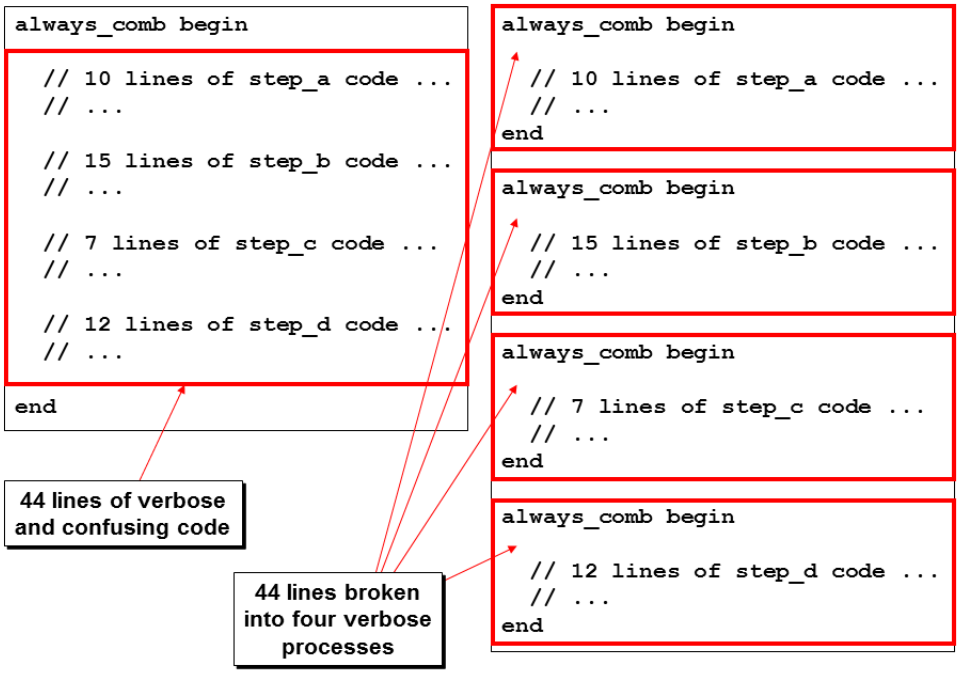

Figure 9 - Large combinational logic and split combinational logic

Section 7.1 showed that an **always_comb** process can now call **void functions** that do not require argument lists and those **void functions** can have very descriptive names. The **void function** calls can now document the higher functionality of the combinational logic. The **void function** calls can now tell a story about the combinational logic. This is shown in Figure 10.

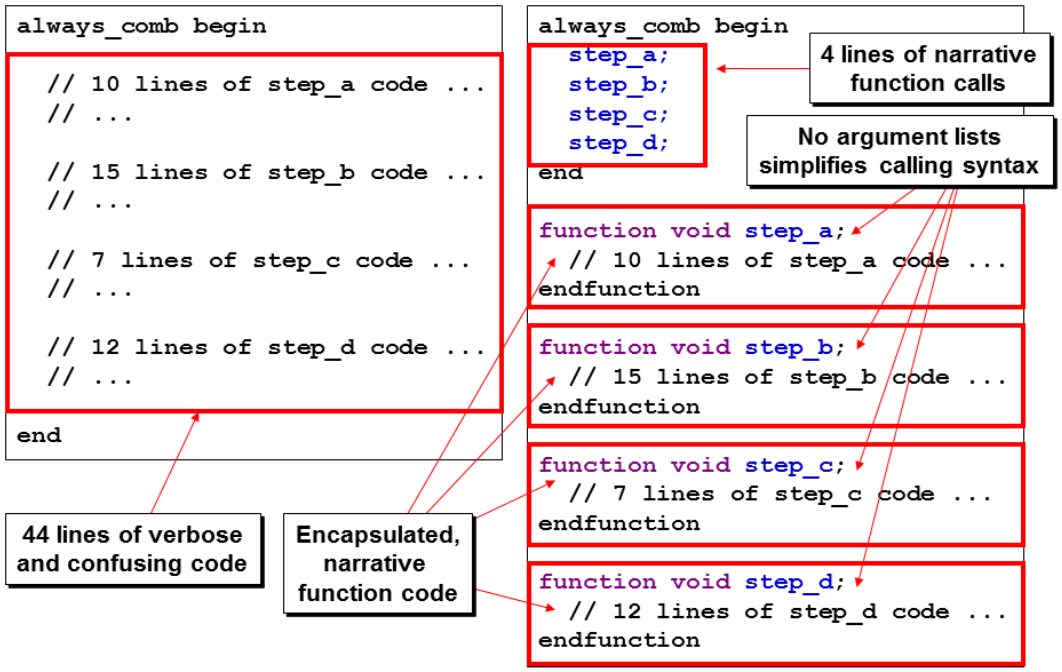

Figure 10 - Large combinational logic and narrative combinational logic calling void functions

In this contrived example, the **always_comb** process executes **step_a**, followed by **steb_b**, followed by **step_c**, and concludes with **step_d**. The high-level intent of the **always_comb** process is more clear and concise, and when an engineer questions what should happen in **step_b**, the engineer can reference the **step_b void function** to see the functionality low-level details.

This strategy has the advantage that each ***step_X*** **void function** has encapsulated part of the functionality. This strategy has the advantage that if the **step_b** code needs to be modified or if signals need to be added or subtracted from **step_b**, those modifications can be done in **step_b** isolation without any requirement to modify the calling code in the **always_comb** process!

This enhancement is a hidden gem that we get for free when we use **always_comb** processes that call **void functions**! This same capability exists when using **always_latch**, but most engineers avoid coding latches this complex (or coding latches altogether!)

7.3 Always_type processes automatically trigger at time-0

Time-0 can be a tricky place in Verilog simulations and time-0 race simulation conditions are not uncommon. Due to this potential problem, simulation vendors have implemented time-0 execution of simulations such that **always** processes are started before **initial** processes. This is not defined in the Verilog and SystemVerilog Standards, but is common practice among the major simulation vendors. This hidden practice removes many of the time-0 simulation race conditions because RTL designs are typically modeled using **always** processes and are therefore activated before the **initial** processes that are typically employed in testbench stimulus generation.

The ***always_type*** processes added one more layer of time-0 race protection not available with **always** processes. All ***always_type*** processes trigger at the end of time-0, so if the input stimulus were to be sent before the ***always_type*** processes were activated, the ***always_type*** processes would still trigger at the end of time-0 and would therefore examine all process inputs and react at time-0 appropriately.

Proper stimulus timing generation techniques and practices are both significant and very important to testbench strategies, but they are beyond the scope of this paper.

7.4 Differences between always_comb and always @*

A summary of the difference between the Verilog **always @*** and the SystemVerilog **always_comb** process is shown below:

- **always_comb** may allow checking for illegal latches. The Verilog **always @*** is used for both combinational and latch-based logic.
- **always_comb** variables cannot be assigned from another process. This allows the simulation compiler to enforce RTL Coding Guideline #6 and avoid synthesis errors.
- **always_comb** cannot include blocking delays, which removes a common pre-synthesis RTL simulation problem.
- **always_comb** is sensitive to changes within the contents of a function
- **always_comb** triggers once automatically at the end of time-0.
- **@*** is permitted to be nested within an **always** process while **always_comb** cannot be nested. It should be noted that nesting **@*** is not a recommended practice, but is allowed by the simulator.

Each of these differences helps to reduce design and simulation mistakes and therefore they show why **always_comb** should be used over the Verilog **always @***.

7.5 Possible future synthesis enhancement - dual edge flip-flop

This section describes a capability that does not currently exist in synthesis tools, but it could in the future. In other words, "don't try this at home, yet!"

Synthesis tools require that either every signal in a sensitivity list has an edge specification or that none of the signals in the sensitivity list has an edge specification.

Consider the poorly coded **always** process flip-flop model shown in Example 1.

```
module dff1a (
  output logic q,
  input  logic d, clk, rst_n);

  always @(clk, negedge rst_n)
    if (!rst_n) q <= 0;
    else        q <= d;
endmodule
```

Example 7 - Erroneous edge / no-edge flip-flop model

This model will compile and simulate like a dual **clk**-edge flip-flop, but DC does not support this coding style and will report an error when the model is read into DC as shown in Figure 11. One of the issues with this coding style is that the synthesis tool cannot determine if the **always** process is an incorrectly coded combinational **always** process or if it is an incorrectly coded flip-flop **always** process. There simply is not enough information in the RTL model to determine what the intended logic should be.

```
Error:  .../dff1a.sv:5:
The event depends on both edge and nonedge expressions,
    which synthesis does not support. (ELAB-91)
*** Presto compilation terminated with 1 errors. ***
```

Figure 11 - DC reports error for edge / no-edge flip-flop model

With the addition of the SystemVerilog ***always_type*** processes, it would now be possible to determine designer intent. Consider the dual-edge **always_ff** flip-flop model in Example 8. The **always_ff** keyword shows the designer's intent to build clocked logic. This model simulates like a dual-edge flip-flop but DC still does not support this coding style.

```
module dff1c (
  output logic q,
  input  logic d, clk, rst_n);

  always_ff @(clk, negedge rst_n)
    if (!rst_n) q <= 0;
    else        q <= d;
endmodule
```

Example 8 - Dual-clk-edge flip-flop - always_ff-NO-edge version

There is no reason that synthesis tools could not allow this model to be accepted and infer a dual-edge flip-flop. The coding style would require that the signal in the sensitivity list without a **posedge** / **negedge** keyword would have to be absent from the equations within the **always_ff**

process. That is how the synthesis tool could identify the signal as being a legal clock signal for a dual-edge flip-flop.

There is one important negative issue with the coding style of Example 8. What if an engineer simply forgot to add the **posedge** to the **clk** signal? Now the synthesis tool would build a perfectly legal but unintended dual-edge flip-flop. For this reason, I propose that a dual-edge flip-flop model be required to add the **edge**[3] keyword to the **clk** signal (as shown in Example 9) to affirm that the engineer intended to have a dual-edge **clk** for the flip-flop model.

```
module dff1d (
  output logic q,
  input  logic d, clk, rst_n);

  always_ff @(edge clk, negedge rst_n)
    if (!rst_n) q <= 0;
    else        q <= d;
endmodule
```

Example 9 - Dual-clk-edge flip-flop - always_ff-edge version

The combination of **always_ff** and the **edge clk** show the designer's intent. There is no ambiguity related to whether the **always** process was combinational or sequential or if the **clk** was mistakenly left blank versus intentionally chose to be a dual **edge clk**.

VCS will simulate the Example 8 code as a dual-edge flip-flop, but currently DC still applies existing rules and reports that the **edge** keyword is not allowed in synthesis, as shown in Figure 12.

```
Error:  .../dff1d.sv:5:
The construct 'edge' is not supported in synthesis. (VER-700)
*** Presto compilation terminated with 1 errors. ***
```

Figure 12 - DC error message for dual-clk-edge flip-flop - always_ff-edge version

If a future ASIC or FPGA library included dual-edge flip-flops, the RTL coding style of Example 9 could be a legal representation of the desired dual-**edge clk** logic.

8. Simulators can warn of incorrectly inferred logic - they don't

Section 7 showed simulation-related errors that are now detected by the simulation compiler but does not address errors related to incorrectly inferred logic by synthesis tools.

The IEEE Std 1800-2012[5] permits simulators the option to warn users if the RTL coding style would not infer the intended and requested logic when synthesized. This action is purely optional and there are no known simulators that issue these useful warnings.

The warnings would be exceptionally useful to the RTL coders, so why don't simulators report these warnings? In some cases, it can be quite difficult to determine what type of logic would be inferred by a synthesis tool strictly by examining the RTL source code.

Although the RTL simulation warnings would be nice, it falls to RTL synthesis tools to truly report

[3] The edge keyword was added to IEEE Std 1800-2005 and made legal in this context in IEEE Std 1800-2009.

errors related to mismatches between the expressed designer intent and actual inferred logic.

9. Synthesis tools warn of incorrectly inferred logic - they should report errors

Although simulators do not issue RTL related warnings regarding designer intent when using ***always_type*** processes, synthesis tools *DO* warn the user when the inferred logic does not match the designer-intended, requested logic.

9.1 Design Compiler always_comb warnings

For an improperly coded **always_comb** process, Design Compiler (DC) issues warnings when the RTL code is read into DC.

```
module ao1e (
  output logic q,
  input  logic en, d);

  always_comb
    if (en) q <= d;
endmodule
```

Example 10 - always_comb - Improperly coded combinational logic - latch style #1

DC shows a latch inference report for this coding style and then warns that a latch was synthesized as shown in Figure 13.

```
Warning: .../ao1e.sv:5: Netlist for always_comb block contains a latch.
(ELAB-974)
```

Figure 13 - Design Compiler always_comb warning message

```
module ao1g (
  output logic q,
  input  logic d, clk, rst_n);

  always_comb @(posedge clk, negedge rst_n)
    if (!rst_n) q <= '0;
    else        q <= d;
endmodule
```

Example 11 - always_comb - Improperly coded combinational logic - ff

For the improperly coded **always_comb** process of Example 11, the simulator will report an error because VCS expects no sensitivity list after the **always_comb** keyword and therefore interprets the **@(posedge clk** ... as a blocking delay that precedes the **if**-test on the next line. Blocking delays used in ***always_type*** processes are not legal as will be explained in Section 6.2. In this example, the VCS compiler (and not DC) reports a timing related error not related to the incorrect **always_comb**:

```
Error-[SV-BCACF] Blocking construct in always_comb/ff
ao1g.sv, 5
ao1g
  Statements in an always_comb shall not include those that block, have
  blocking timing or event controls, or forkjoin statements. The always_ff
  process imposes the restriction that it contains one and only one event
  control and no blocking timing controls.
  Try using simple always blocks.
```

Figure 14 - VCS error message for illegal always_comb code with sensitivity list

The code in Example 11 never even compiles for simulation so it was not read into DC.

9.2 Design Compiler always_latch warning

For an improperly coded **always_latch** process, Design Compiler (DC) issues warnings when the RTL code is read into DC.

```
module lat1d (
  output logic q,
  input  logic en, d);

  always_latch
    q = en & d;
endmodule
```

Example 12 - always_latch - Improperly coded latch logic - combinational

DC shows NO latch inference report for this coding style and then warns that a latch was not synthesized as shown in Figure 15.

```
Warning: .../lat1d.sv:5: Netlist for always_latch block does not contain a
latch. (ELAB-975)
```

Figure 15 - always_latch - Improperly coded latch logic - combinational logic

```
module lat1e (
  output logic q,
  input  logic d, clk, rst_n);

  always_latch @(posedge clk, negedge rst_n)
    if (!rst_n) q <= '0;
    else        q <= d;
endmodule
```

Example 13 - always_latch - Improperly coded latch logic - combinational

For the improperly coded **always_latch** process of Example 13, the simulator will report an error because VCS expects no sensitivity list after the **always_latch** keyword and therefore interprets the **@(posedge clk** ... as a blocking delay ("***a single statement which is an event control***") that precedes the **if**-test on the next line. Blocking delays used in ***always_type*** processes are not legal as will be explained in Section 6.2. In this example, the VCS compiler (and not DC) reports a timing related error not related to the incorrect **always_comb**:

```
Error-[OECAB] Singular control statement in always
lat1e.sv, 5
lat1e
'always_comb'/'always_ff' block has a single statement which is an event
control. Such usage is disallowed due to likelihood of user error.
```

Figure 16 - VCS error message for illegal always_latch code with sensitivity list

9.3 Design Compiler always_ff warning

For an improperly coded **always_ff** process, Design Compiler (DC) issues warnings when the RTL code is read into DC.

```
module dff1d (
  output logic q,
  input  logic d, clk, rst_n);

  always_ff @(clk, rst_n)
    if (!rst_n) q <= '0;
    else        q <= d;
endmodule
```

Example 14 - always_ff - Improperly coded ff logic

The **always_ff** sensitivity list of Example 14 is missing edge specifications. If this example only used the older Verilog **always** keyword, the synthesis tool would assume that the code represented combinational logic, but the **always_ff** keyword is recognized by DC so DC warns that a flip-flop was not synthesized as shown in Figure 17.

```
Warning: .../dff1d.sv:6: Netlist for always_ff block does not contain a
flip-flop.
```

Figure 17 - Design Compiler always_ff warning message

The **always_ff** code of Example 14 also causes DC to report a potential problem and instructs the user to run the **check_design** command as shown in Figure 18.

```
Information: There are 1 potential problems in your design.
Please run 'check_design' for more information.

check_design:
Warning: In design 'dff1d', port 'clk' is not connected to any nets.
```

Figure 18 - always_ff check_design warning

After running the **check_design** command, DC reports that the design is missing the **clk** signal from any assignments in the process and then proceeds to build the 2-input **and**-gate shown in Figure 19. Also shown in Figure 19 is that the **clk** signal is unconnected and is just a dangling signal.

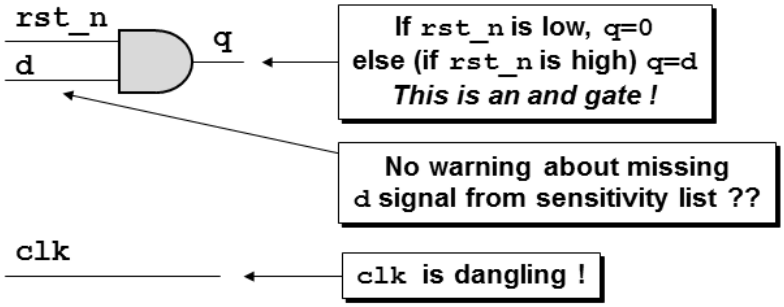

Figure 19 - Incorrect synthesized and-gate from erroneous always_ff process sensitivity list

What is disturbing about this implementation is that there was no warning issued by DC that the **d**-input signal was not part of the sensitivity list. The missing **d**-input signal means that pre-synthesis changes to the **d**-input will not cause the **always_ff** process to re-evaluate the assignments in the **always_ff** process, while the post-synthesis gate-level simulation will indeed re-evaluate the **q**-output assignment every time the **d**-input changes. This is a potential source for mis-matches between pre-synthesis and post-synthesis simulations.

9.3.1 Did Design Compiler quit issuing sensitivity list warnings?

I remember that DC used to build combinational logic from an **always** process without considering the process sensitivity list, but after building the logic, would then warn the user if signals were missing from the sensitivity list. This was a valuable warning but it appears that the missing sensitivity list signals are no longer being checked.

My colleague Brian Kane noticed this same problem in 2013 and reported the issue. The assigned case number was "`Case 8000639472 : incomplete combinatorial sensitivity`"

The Synopsys response to Brian at that time was

```
I was able to reproduce what you described, and I was as surprised by it as
you were, since historically the tool would issue an ELAB-292 warning for an
incomplete sensitivity list.  I did some research, and it turns out that the
behavior you're seeing is part of a larger effort to remove lint-like (i.e.
code quality) checks from HDL Compiler.  According to R&D, "HDL Compiler was
issuing too many 'false positives', thus the message was unreliable and it
was removed."  The expectation nowadays is that users will do their code
quality checks with a dedicated linting tool.
```

Removal of linting checks from DC does not benefit users but at least with the addition of **always_comb** and **always_latch**, there is no need to check the sensitivity list for these ***always_type*** processes.

Unfortunately if an **always_ff** process is used incorrectly, DC now warns that a flip-flop was not inferred, builds combinational logic, and then does not warn users that the combinational pre-synthesis RTL is missing important sensitivity list signals. If the synthesis tool would issue an error for this flawed coding style, there would be no need for DC to check the sensitivity list signals but that currently is not the case.

9.4 Why warnings and not errors?

Why do synthesis vendors issue warnings instead of errors?

First, it should be noted that synthesis behavior regarding the implementation of ***always_type*** processes is not defined by any IEEE Standard.

Second, vendors are concerned about issuing false error messages ("false positives"). If the vendor tool issues an error message where no error is present, customers will lose confidence in vendor error messages.

Third, vendor synthesis tools have issued ***always_type*** warnings for years and if they change the warnings to errors, this might cause poorly coded designs that previously compiled to fail synthesis compilation. There are two possible solutions to this latter problem:

a) My preference is that vendors change the ***always_type*** warnings to errors and force designers to use good and informed coding styles. If an **`always_comb`** process infers latches, I believe synthesis tools should force the engineer to change the flawed-**`always_comb`** keyword to the actual-intent keyword of **`always_latch`**. As an alternative, the designer can change the **`always_comb`** to **`always`** keyword to avoid additional checking, or tool vendors could give engineers a new command switch that converts ***always_type*** violations from errors into the currently reported warnings.
b) A second and less-attractive alternative is to give designers a synthesis switch to convert ***always_type*** warnings into errors. The flaw with this alternative is that the synthesis tool is issuing warnings when errors would help designers identify design-intent errors. If this alternative is implemented, I would encourage all synthesis engineers to turn this switch on permanently.

The ***always_type*** synthesis compilation warnings should be replaced with errors for the following reasons.

If I, as an RTL designer, took the time to specify the type of logic I intended to be inferred in my design, and if that logic is not inferred by the synthesis tool, then I want an error that will force me to correct my design before DC legally reads the design, not a warning that is easily lost as it scrolls past my visible synthesis transcript window. The whole intent of adding ***always_type*** processes is to allow me to show enforceable designer intent and warnings just do not adequately achieve that goal. If the proposed synthesis tool errors prove to be problematic, I have an existing fallback position. I can remove all of the new SystemVerilog ***always_type*** processes and go back to using a plain old Verilog **`always`** process, which is mostly ignored by synthesis tools as it pertains to intent-checking.

There is some concern regarding which inferred logic should be legal with existing ***always_type*** coding styles. As someone who has worked with RTL coders for about 22 years and who also participated on the creation of the IEEE Std. 1364.1-2002[6] standard (RTL synthesis standard), I propose that the following styles should be legal or illegal as outlined in Table 2.

Legal Inferred Logic →	Combinational Logic	Combinational Feedback Logic	Latch Logic	Flip-Flop Logic
`always_comb`	✓	Error	Error	Error
`always_latch`	✓*	Error	✓	Error
`always_ff`	✓**	Error	Error	✓

✓* – Must also infer latch
✓** – Must also infer flip-flop

Table 2 - Proposed synthesis tool legal-logic checking

As noted beneath Table 2, it should be legal for **`always_latch`** and **`always_ff`** to simultaneously infer combinational logic as long as they also drive the data pin of the requested latches and flip-flops respectively.

Also as noted in Table 2, combinational feedback logic can be problematic for designs. Full Static Timing Analysis (STA) cannot be performed on a combinational feedback loop and such feedback loops have latch-behavior without checking the setup and hold times of the latching control signals inherently associated with such logic. For those reasons, I propose that currently defined ***always_type*** blocks be prohibited from legally building logic with combinational feedback loops. If such logic is desired, engineers should be forced to use older Verilog **`always`** processes so that they are aware of the potential dangers of these types of designs. See Section 10 for a proposed solution to this issue.

The problem with the proposed error checking is that, as noted above, synthesis tools have allowed engineers to get by with warnings on existing designs. Synthesis tool vendors might find it acceptable to provide a **`-warn_only`** or **`-relax_fatal`** switch (the exact name is not important) to allow engineers to use the tools without errors if they choose to ignore important design intent violations.

As an alternative, synthesis tools could issue "violations," which could be put under user control to either convert violations into errors or into the current set of warnings as selected by the user.

The point is, synthesis tools are currently issuing warnings when errors are incorrectly coded into RTL designs. Warnings are almost useless. Errors/violations will help engineers to build better designs!

10. Proposed: always_comb_fb

One observation that I made while compiling the list of important synthesis tool errors shown in Table 2, is that I believe RTL designers actually need another ***always_type*** process.

RTL models should be re-targetable to different devices to allow for reuse in future projects and technologies, and therefore the RTL code cannot know if the target implementation will have latches available

Although **`always_comb`**, **`always_latch`** and **`always_ff`** cover 95%+ of RTL coding styles, there is still one more style that, although rarely used, might be useful to some designers. That style is a

combinational feedback loop that acts like a latch.

Although latches are frequently discouraged in RTL designs by most companies, there are times when latch-based logic is useful. There are many FPGA families that do not include usable latches and for those FPGA devices, a latch must be modeled as combinational logic with a feedback-loop to hold the desired latched values. I propose that one new ***always_type*** process be added to SystemVerilog: **`always_comb_fb`** (combinational logic with intentional feedback).

RTL simulations frequently do not have enough context to know if a latch will be available in the implementation logic. ASICs could allow latches while equivalent prototype FPGA devices might not have access to dedicated latches. Since this is a real possibility, simulators should just treat **`always_comb_fb`** the same as **`always_comb`**. For simulators, **`always_comb`** and **`always_comb_fb`** would be synonyms.

For synthesis tools, there are multiple scenarios that should be addressed:

(1) If a latch is inferred from an **`always_comb`** process, the synthesis tool should report one of two errors, with messages similar to, "**`always_comb`** inferred a latch" or "**`always_comb`** inferred combinational logic with a feedback loop."

(2) If no latch is inferred from an **`always_latch`** process, the synthesis tool should report one of two errors, with messages similar to, "**`always_latch`** used but no latch inferred" or "**`always_latch`** inferred combinational logic with a feedback loop."

(3) If a latch is inferred from an **`always_comb_fb`** process, the synthesis tool should report an error, with a message similar to, "**`always_comb_fb`** inferred a latch."

(4) If non-latch and non-feedback logic is inferred from an **`always_comb_fb`** process, the synthesis tool should report an error, with a message similar to, "**`always_comb_fb`** inferred combinational logic but no combinational feedback logic as requested."

I have also considered a coding style that infers both latches and flip-flops from the same process, but I consider that to be somewhat questionable since an RTL coder could easily separate latches from the flip-flops by separately coding the **`always_latch`** and **`always_ff`** processes.

11. Conclusions

Follow the 8 RTL coding guidelines shown in Section 3. These guidelines, originally shown for Verilog RTL designs still apply to SystemVerilog RTL designs.

Guideline: Quit using **`tasks`** in RTL design. Replace all **`tasks`** with **`void functions`**.

Guideline: Quit using the **`always @*`** process for RTL combinational and latching logic. The **`always @*`** does not do deep-searching into called functions to build the complete process sensitivity list.

Guideline: Quit using all **`always`** processes for RTL designs and instead use the ***always_type*** processes exclusively.

The new SystemVerilog ***always_type*** logic processes are far superior to existing **`always`** processes. The ***always_type*** processes do more checking during simulation compilation for potential error conditions and they currently allow synthesis tools to at least warn engineers when the intended logic is not synthesized.

Synthesis tool warnings related to **`always_type`** processes are not nearly as useful as errors that would cause the synthesis tool to stop before compiling a design. Warnings are too easily missed.

Synthesis tool vendors should issue errors when the designer's intended and requested logic is not inferred as described in Table 2. Designers have a readily available fallback position if they want to avoid the errors, and that fallback position is to use the simple Verilog `always` process. Synthesis tool vendors could help accelerate correct RTL synthesis by allowing the current ***always_type*** warnings to be converted into errors.

Call to action! Ask your synthesis tool vendor to please give us a way to generate errors instead of warnings when the RTL designs are read into the synthesis tool. Ask the synthesis tool vendor to disallow compilation of the incorrect designs until the errors have been addressed by the design engineer.

12. Acknowledgements

I am grateful to my colleagues Stuart Sutherland and Brian Kane for their reviews and suggested improvements to this paper.

13. References

[1] Clifford E. Cummings, '"full_case parallel_case", the Evil Twins of Verilog Synthesis,' SNUG'99 Boston (Synopsys Users Group Boston, MA, 1999) Proceedings, 1999. Also available online at www.sunburst-design.com/papers

[2] Clifford E. Cummings, "Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!," SNUG (Synopsys Users Group) 2000 User Papers, section-MC1 (1st paper), March 2000. Also available at www.sunburst-design.com/papers

[3] Clifford E. Cummings, "SystemVerilog Implicit Port Enhancements Accelerate System Design & Verification," SNUG (Synopsys Users Group) September 2007 (Boston, MA), September 2007. Also available at www.sunburst-design.com/papers

[4] Don Mills and Clifford E. Cummings, "RTL Coding Styles That Yield Simulation and Synthesis Mismatches," SNUG (Synopsys Users Group) 1999 Proceedings, section-TA2 (2nd paper), March 1999. Also available at www.lcdm-eng.com/papers.htm and www.sunburst-design.com/papers

[5] IEEE Standard Verilog Hardware Description Language, IEEE Computer Society, IEEE, New York, NY, IEEE Std 1364-2001.

[6] IEEE Standard for Verilog Register Transfer Level Synthesis, IEEE Computer Society, IEEE, New York, NY, IEEE Std 1364.1-2002

[7] "IEEE Standard For SystemVerilog - Unified Hardware Design, Specification and Verification Language," IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group, IEEE, New York, NY, IEEE Std 1800™-2012

14. Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 34 years of ASIC, FPGA and system design experience and 24 years of SystemVerilog, synthesis and methodology training experience.

Mr Cummings has presented more than 100 SystemVerilog seminars and training classes in the past 13 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

Last Updated: April 2016

Appendix 1 **Tools and OS versions**

The examples in this paper were run using the following Linux and Synopsys tool versions:

64-bit Linux laptop: CentOS release 6.5

VCS version K-2015.09-SP1_Full64

Running vcs and dve each required the command line switch -full64

Without the -full64 command line switch, vcs compilation would fail with the message:

```
g++: /home/vcs/linux/lib/ctype-stubs_32.a: No such file or directory
make: *** [product_timestamp] Error 1
Make exited with status 2
```

Design Compiler version K-2015.06-SP4

Appendix 2 **The design flow - Finding and fixing bugs**

The earlier a bug is identified and fixed, the faster a high quality design will be released to market. The subsections in this appendix give somewhat detailed descriptions of how and where bugs can be identified and fixed.

Appendix 2.1 **Coding**

If a bug can be spotted while the RTL is being entered, the bug can be quickly fixed before any tool is used to identify bugs. A colorizing and indenting editor will help the skilled coder to identify bugs before the design is even compiled by any tool.

SystemVerilog does not necessarily add any new features that a colorizing editor would use to identify bugs that were not already found using a colorizing editor with Verilog.

Appendix 2.2 **Linting**

Linting tools examine code to identify suspect declarations, assignments and/or coding styles. If a company has access to a linting tool, the design should be "linted" (compiled by the linting tool) before it is compiled for simulation.

Not all designers have access to linting tools. Designers that do have access to linting tools often can identify problems that otherwise would not be found until a design is compiled for simulation, simulated, compiled for synthesis or synthesized. It should be noted that a synthesis tool is a very expensive linting tool that only reports a few errors at a time while commercial linting tools have been optimized to find and report as many errors as is possible with just a few passes through the tool and before using any other tool.

The same SystemVerilog features that enable simulators and synthesis tools to identify bugs early will also help linting tools to become more valuable.

I am frequently asked if there are any public domain linting tools. As of this writing, I know of no freely available, public domain linting tools. If I become aware of any publically available linting tools, I will report that information in an updated version of this paper on the sunburst-design.com web page. Please do not email to ask if I know of any updates! Feel free to email me if YOU know of any updates!

Appendix 2.3 **Compiling for simulation**

Compilation typically refers to both: (a) syntax checking during a compilation phase, and (b)

connectivity and parameter resolution during an elaboration phase. Some simulators do both when you call the compilation command, while other simulators do part (a) during compilation and part (b) when a separate simulation command is called.

Finding bugs during compilation saves lengthy and wasted simulations. Every designer who executes a simulation command automatically calls a tool that performs these syntax and connectivity checks, and the designer is required to fix problems before simulation will start.

SystemVerilog enhancements add value to the compilation and elaboration steps, helping to identify additional problems that previously went undetected during Verilog compilation. Examples of this have been shown in this paper. The bottom line is that SystemVerilog helps catch more bugs earlier in the design compilation stage.

Appendix 2.4 **Simulation**

Functional simulation (pre-synthesis simulation) is used to prove that the logic, as coded, will function properly in an actual design, assuming that there are no coding issues that would cause a mismatch between pre-synthesis and post-synthesis simulations.

Simulation requires the design and verification teams to put together an elaborate set of test cases to apply useful stimulus to the design. It should be noted that simulation is not inherently exhaustive, and is only as thorough as what a verification engineering team coded for the testbench stimulus and output checking.

Simulation should be used to prove the design functionality is correct before launching a synthesis tool. If the design is not proven to be functionally correct before synthesis, then synthesis runs are largely a waste of project time.

Appendix 2.5 **Simulation - code coverage**

When a full set of tests is run against the design, code coverage can be enabled through vendor-specific command line switches. Code coverage does not require any special effort on the part of the verification engineer, aside from enabling the switches and the time spent to analyze the code coverage results, which can be significant.

At the same time, code coverage does not answer the question regarding completeness of design or completeness of test generation. The only question answered by code coverage is if there is any code that has not been touched by any of the tests and hence, if there is any code that has not been tested. Untouched code could still have functional bugs. Code coverage is used to ensure that all the code has been touched.

Appendix 2.6 **Simulation - functional coverage**

While code coverage was a semi-automatic process to ensure that all code had been touched by tests, functional coverage is not automatic and requires a test plan. Functional coverage is used to ensure that all of the specified design features have been exercised. Functional coverage does not ensure that the features work properly. It is a combination of simulation and functional coverage reports (ideally with 100% functional coverage) that indicates that the design is functionally correct and ready to be delivered to customers. In a perfect scenario, functional coverage indicates that ideally 100% of the specified features have been tested and if the functional simulation passes, those features are working.

SystemVerilog adds features to significantly improve functional coverage capabilities, but those features are not described in this paper.

Appendix 2.7 **Synthesis compilation**

Synthesis compilation is basically the synthesis linting step to identify illegal RTL coding styles per the RTL coding policies of the synthesis tool, followed by inference of logic from the RTL equations and mapping the design into target-specific gates. The logic inference is also influenced by the availability of common gates in specific vendor libraries. For example, most ASIC families can infer latches while many FPGA families do not have latches.

SystemVerilog could potentially add significant value to identify problems during synthesis compilation, helping to identify additional problems that previously went undetected during Verilog synthesis. Examples have been shown in this paper. SystemVerilog helps catch more bugs earlier in the design stage.

It should also be noted that SystemVerilog synthesis tools should do a better job of identifying problems than is currently true. How SystemVerilog synthesis tools can improve were shown in Section 9.

Where possible, synthesis tools should report errors as opposed to reporting warnings. Warnings are easily missed and indeed are frequently missed. Valid error reporting is much more valuable than reporting warnings as the latter can often lead to wasted synthesis runs.

Appendix 2.8 **Post-synthesis simulation / Equivalency checking**

After the design is synthesized and a gate-level netlist is available, a designer should perform post-synthesis simulations or equivalency checking before tape-out of a design.

If an engineer does not have access to equivalency checking tools, the engineer must run gate-level simulations (gate-sims) to prove that the post-synthesis gate-level design matches the pre-synthesis RTL representation of the design.

A frequently asked question regarding gate-sims is, if the design passed RTL simulations and passed synthesis, why do I have to run gate-sims? There are many well documented coding styles and coding practices that can lead to a mis-match between pre-synthesis simulations and post-synthesis simulations[1][4]. These mismatches are indications of functional bugs.

If a designer has access to equivalency checking tools, these tools can be used to reduce lengthy gate-level simulations. Equivalency checking tools attempt to mathematically prove that the post-synthesis gate-level representation of the design matches the pre-synthesis RTL representation of the design. If it can be proven mathematically that the gate-level representation matches the RTL representation, and if the RTL simulations proved that the designed worked and was fully functional, then there is reduced need to run gate-sims.

Gate-level simulations can help check timing constraints or timing exceptions and can test logic that does not exist in RTL simulations such as DFT logic.

SystemVerilog does not necessarily add any new features that gate-sims or equivalence checking tools would use to identify bugs that were not already found using Verilog versions of these tools.

Appendix 2.9 **Silicon testing**

Placing the completed ASIC or FPGA into a design for system testing is the second-to-last place an engineer wants to find a bug. For ASICs and FPGAs, finding a bug this late in the development cycle will require engineering teams to identify deficiencies in the functional coverage model, followed by the deficiency in the simulation tests, followed by the deficiency in the RTL code itself.

Replacing a failing ASIC that cannot be fixed by software patches can be a very costly and time-

consuming process. Replacing the FPGA is less painful but still requires all of the same ASIC re-testing steps followed by successful synthesis, placement and routing of the FPGA design.

One of the goals of SystemVerilog is to identify more of these issues early to help avoid these costly problems.

Silicon testing is used to find any final bugs before delivering the chip to the customer.

Appendix 2.10 **Customer testing**

Finding the bug once the design is in the hands of customers is the last place an engineer wants to find a bug. Fixing bugs found by a customer can require a costly recall of the product, or going bankrupt!

One of the goals of SystemVerilog is to keep your company from going bankrupt!