Yet Another Latch and Gotchas Paper

Don Mills

Microchip Technology, INC
Chandler, AZ, USA

www.microchip.com

ABSTRACT

This paper discusses four SystemVerilog coding topics that can lead to inadvertent design bugs. Old constructs such as casex, casez, full_case and parallel_case are briefly revisited. Newer constructs like unique case and priority case added by SystemVerilog 2005 are reviewed. This paper explores these constructs and explains where they break down. Presented are features from SystemVerilog 2009 and other modeling techniques for solving design bugs. An updated model for the Async-Set/Reset-FlipFlop is shown. Finally, clarification of the "logic" keyword as specified in the SystemVerilog 2009 standard is given.

Table of Contents

1.0 Introduction

This paper discusses and provides solutions to issues that designers using SystemVerilog for design must address, such as:

- Case expression issue for casez and casex
- Latches generated when using unique case or priority case
- SRFF coding style problems with synthesis
- SystemVerilog 2009 new definition of logic

The SystemVerilog **`casez`** and **`casex`** types of case statements have an inherent simulation/synthesis mismatch. The old Verilog workaround was to use **`casez`** because it was less likely to have problems. Shown in this paper are models that provide better than "less likely", with the use of assertions and the SystemVerilog 2009 case inside feature.

Many designers are led to believe that **`unique case`** and **`priority case`** solve their latch issues regarding **`case`** statements. This is simply not so. These constructs come close, but do not cover all the conditions where latches can inadvertently be generated. This paper explains where these constructs break down and provides alternative solutions that will always work. The new SystemVerilog 2009 **`unique0`** is discussed along with recommendations of usage.

The issue regarding asynchronous Set/Reset Flip-Flops (SRFF's) has been discussed in previous papers. [1] This paper provides a better solution. Even though the SRFF issue has been addressed in the past, many designers are not aware of it. This one issue is the most common problem I address when working with designers at different companies.

The final sections discuss how the definition of **`logic`** usage has changed from SystemVerilog-2005[3] to SystemVerilog-2009[2]. The old rules have changed, and, as a result, the usage model is different today. This paper will attempt to clarify if this is a concern for your designs, and, if so, provide a reasonable usage model.

Years ago, as I was developing my first training class, I reflected on the coding constructs I used for design. It occurred to me that in all of my design work, I primarily used only two coding structures to model hardware: **`if`** statements and **`case`** statements. All the other parts of the language were like supporting actors to these two stars. Many papers have been written and presented at conferences over the years about coding styles for RTL using **`if`** and **`case`** statements. Three of the most widely distributed papers are:

Clifford Cummings, *"'full_case parallel_case', the Evil Twins of Verilog Synthesis"* [4]

Clifford Cummings, "*SystemVerilog's priority & unique—A Solution to Verilog's 'full_case' & parallel_case' Evil Twins!*" [5]

Stuart Sutherland, "*SystemVerilog Saves the Day—the Evil Twins are Defeated! 'unique' and 'priority' are the new Heroes*" [6]

This paper adds a few new approaches not presented in earlier papers. To lay the foundation of these new items, we must review some of the old methods.

2.0 Combinational Case Coding

The goal, when writing combinational logic using **case** statements, is to write code that only models combinational logic for both simulation and synthesis, and which, in neither case, implies latches.

The terms **case** expression and **case** item are used frequently throughout this paper. These terms are demonstrated as follows:

```
case (sel)
  2'b00 : out = a;
  2'b01 : out = b;
  2'b10 : out = c;
  2'b11 : out = d;
endcase
```

Given this code snippet, the case expression is **sel**, and the terms on the left side of the colons (**2'b00, 2'b01, 2'b10, 2'b11**) are the case items.

2.1 The Plain Ole' Case Statement

For basic **case** statements, if the **case** expression does not match one of the **case** items, the output will retain the previous value. This most often happens when the **case** items list does not fully decode the **case** expression. In example 2.1a, the **sel** is partially decoded and will result in a latch being modeled for the non-decoded **sel** state in both simulation and synthesis.

```
module mod_21a
  (input [1:0] sel,
   input       a, b, c, d,
  output logic out);

  always_comb
    case (sel)
      2'b00 : out = a;
      2'b01 : out = b;
      2'b10 : out = c;
    endcase

endmodule:mod_21a
```

Code Example 2.1a

In example 2.1b, the **sel** is fully decoded from a binary perspective, so no latches, right? If the **sel** goes to X (or Z) in simulation, the unknown **sel** will be masked and the output will hold its previous value (latched output from previous state). For synthesis, the case expression is considered to always have a known value, so this latch condition does not exist in the synthesized results. This is a classic X-propagation problem with a latch simulation/synthesis mismatch.

```
module mod_21b
  (input [1:0] sel,
   input       a, b, c, d,
  output logic out);

  always_comb
    case (sel)
      2'b00 : out = a;
      2'b01 : out = b;
      2'b10 : out = c;
      2'b11 : out = d;
    endcase

endmodule:mod_21b
```

Code Example 2.1b

Another approach is to propagate X through the latch with a default statement, as shown in the code example 2.1c.

```
module mod_21c
  (input [1:0] sel,
   input       a, b, c, d,
   output logic out);

  always_comb  //similar to always@* but better
    case (sel)
      2'b00 : out = a;
      2'b01 : out = b;
      2'b10 : out = c;
      2'b11 : out = d;
      default : out = 'x;
    endcase

endmodule:mod_21c
```

Code Example 2.1c

Propagating X is a better solution than ignoring it. Most designers do not bother with the possibility that the **case** expression could be unknown during RTL simulation. A simple solution to monitoring and detecting this condition is shown in section 2.5.

2.2 SystemVerilog casex

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

statement. In example 2.2a below, if the **case** expression == 3'bxxx or if the **case** expression == 3'bzzz, the first **case** item would always be selected. In contrast, the corresponding post-synthesis simulation with these same unknown inputs will propagate X's through the gate-level model.

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

Code example 2.2a below models a simple address decoder with an enable. Sometimes design errors in an external interface will cause the enable to glitch to an unknown state after initialization, before settling to a valid state. While the enable is in this unknown state, the **case** expression will erroneously match one of the **case** items, based on the value of **addr**. In the pre-synthesis design, this unknown condition is treated as a "don't care" which might mask a reset initialization problem that would only be visible in post-synthesis simulations. A similar situation could exist if the MSB of the address bus went unknown while **en** is asserted. This would cause either **memce0** or **memce1** to be asserted whenever the chip select (**cs**) signal should have been asserted.

```
module mod_22a
  (output logic   memce0, memce1, cs,
   input          en,
   input  [31:30] addr);

  always_comb begin
    {memce0, memce1, cs} = 3'b0;
    casex ({addr, en})
      3'b101: memce0 = 1'b1;
      3'b111: memce1 = 1'b1;
      3'b0?1: cs     = 1'b1;
    endcase
  end
endmodule:mod_22a
```

Example 2.2a - Casex Address Decoder

Guideline: Do not use **casex** for RTL coding without other X-trapping monitoring. It is too easy to match a stray, unknown signal. It is better to use the **casez** statement, as shown in the next section.

2.3 SystemVerilog casez

The use of **casez** statements can cause the same design problems as **casex**, but these problems are less likely to be missed during verification. With **casez,** a problem would occur if an input were initialized to a high-impedance state. Like the **casex**, the **casez** statement pro-

vides a short, concise, tabular method for coding certain useful structures, such as priority encoders, interrupt handlers, and address decoders. Therefore, the **casex** and **casez** statements should not be completely dismissed from a design engineer's repertoire of useful HDL coding structures. The **casez** has been promoted as the favored of the two because it is less likely to have an error condition occur.

Code example 2.3a is the same simple address decoder with enable as shown in example 2.2a above, except that it uses the **casez** statement instead of the **casex** statement. The same problem described in Section 2.2 will occur when one of the inputs goes to a high-impedance state rather than an unknown state. Once again, an erroneous **case** match will occur, depending on the state of the other inputs to the **case** statement. However, it is less likely that a stray match will occur with a **casez** statement (floating input or tri-state driven signal) than with a **casex** statement (signal goes unknown briefly), but a potential problem does exist.

(Old) Guideline : Use **casez** over **casex**, but use it sparingly and cautiously for RTL coding, since it is possible to match a stray tri-state signal in the **case** expression. In addition to stray tri-state values on the **casez** expression causing erroneous matches to occur, what happens if a bit in the **case** expression goes undefined? This model will not get erroneous matches as with the **casex**, but it can hide X's from propagating. Even a default **case** item will not always work to catch and propagate the X's that occur in the **case** sel of a **casez**. SystemVerilog provides better solutions than just using **casez**, as discussed in a later section of this paper.

```
module mod_23a
  (output logic   memce0, memce1, cs,
   input          en,
   input  [31:30] addr);

  always_comb begin
    {memce0, memce1, cs} = 3'b0;
    casez ({addr, en})
      3'b101: memce0 = 1'b1;
      3'b111: memce1 = 1'b1;
      3'b0?1: cs     = 1'b1;
    endcase
  end
endmodule:mod_23a
```

Example 2.3a - Casez Address Decoder

2.4 SystemVerilog `case inside`[1]

The **casex** and **casez** statements allow the mask bit to be set on either side of the comparison. In the preceding **casex** examples, if **{addr, en}** has a value of **3'bxxx** (or **3'bzzz**), all bits are masked from the comparison, which means the first branch of the case statement will be executed. A partial solution to this *gotcha* is to use **casez** instead of **casex,** as discussed in the previous section. In the example used in this section, if a **casez** were used, a design

[1] This section is updated from Sutherland/Mills 2006 Gotcha paper [7]

problem that causes an **instruction** of **3'bxxx** (or even just an X in the left-most bit) will not be masked, and an invalid **instruction** will be reported by the default branch (old-fashioned assertion). However, a design problem that causes an **instruction** of **3'bzzz** (or just a Z in the left-most bit) will still be masked, and an invalid **instruction** will not be trapped.

SystemVerilog offers two solutions to this *gotcha*. The first solution is a special one-sided, wild-card comparison operator, **==?** (there is also a **!=?** operator). This wildcard operator works similarly to **casex**, in that bits can be masked from the comparison using X, Z or ?. However, the mask bits can only be set on the right-hand side of the comparison. In the following example, any X or Z bits in **instruction** will not be masked, and the invalid **instruction** will be trapped:

```
if (instruction ==? 4'b0???)
  opcode = instruction[2:0];
else if ... // decode other valid instructions
else begin
    $display ("ERROR: invalid instruction!");
    opcode = 3'bxxx;
  end
```

Example 2.4a "==?" wildcard comparison operator

A second solution to this *gotcha* is the SystemVerilog **case() inside** statement. This statement allows mask bits to be used in the **case** items using X, Z or ?, as with **casex,** but **case()inside** uses a one-way, asymmetric masking for the comparison. Any X or Z bits in the **case** expression are not masked. In the following example, any X or Z bits in **instruction** will not be masked, and the invalid instruction will be trapped by the default condition:

```
always_comb begin
  case (instruction) inside
    4'b0???: opcode = instruction[2:0]; //only test msb bit
    4'b1000: opcode = 3'b001;
    ... // decode other valid instructions
    default:
      begin
        //synthesis translate_off
          $display ("ERROR: invalid instruction!");
          opcode = 3'bxxx; // propagate the X
        //synthesis translate_on
      end
  endcase
end
```

Example 2.4b – Case Inside

SystemVerilog **case inside** will give the functionality of **casez** or **casex** without the erroneous matches resulting from **case** expression being **X** or **Z** (unknown). However, the X propagation problems can still exist with **case inside,** even if there exists a **default case** item to attempt to propagate the X's. In code example 2.4b, when the instruction value is 4'b0x01, the X will not propagate. Whether the X matters in this case can be questioned, because the matching **case** item declares that since the msb is 0, the other three bits are "don't cares". This question is relative to specific designs.

2.5 The Assertive Solution

This solution for dealing with **case/casez/casex** unknown **case** expressions is similar to using **case inside**. This approach monitors the **case** expression for the error condition using an assertion statement immediately preceding the **case** statement. Where **case inside** still allows the **case** expression to be unknown, it does not "wild card" on the **case** expression when X or Z. The assertion solution proposed here will monitor the **case** expression and assert an error when the **case** expression is unknown. One might question, once the **case** expression is asserted as unknown, if there is still a need to propagate X's.

Assertions can be disabled during resets, or at other times during simulation, as needed. Also, the severity level can be adjusted between warning, error or fatal, as desired. When a **case** statement covers all the binary conditions and uses an assertion as shown below, the **case** default may no longer be needed to propagate X's through the **case** statement as was done in example 2.1c. The **case** items can still use the X's as "don't cares" for simulation and synthesis modeling. Since assertions will trap all occurrences when the **case** expression goes to X or Z, it no longer matters whether **casex** or **casez** are used. Both become safe constructs.

In code example 2.5a, the same code from example 2.1c is used, but with the assertion added, to monitor for unknowns in the **case** expression.

```
module mod_25a
  (input a, b, c, d,
   input [1:0]  sel,
   output logic out);

  always_comb begin
    assert (!$isunknown(sel))
       else $error("%m : case_Sel = X");
    case (sel)
      2'b00 : out = a;
      2'b01 : out = b;
      2'b10 : out = c;
      2'b11 : out = d;
    endcase
  end
endmodule:mod_25a
```

Example 2.5a case with assert

In code example 2.5b, a **casex** is used with an assertion.

```
module mod_25b
  (output logic  memce0, memce1, cs,
   input         en,
   input [31:30] addr);

    always_comb begin
      {memce0, memce1, cs} = 3'b0;
      assert (!$isunknown({addr,en}))
        else $error("%m : case_Sel = X");
      casex ({addr, en})
        3'b101: memce0 = 1'b1;
        3'b111: memce1 = 1'b1;
        3'b0?1: cs     = 1'b1;
      endcase
    end
  endmodule:mod_25b
```

Example 2.5b - Casex Address Decoder

Using the assertion before the **case** statement, as in example 2.5b, is preferred over a **case** default for **casez** and **casex** because an unknown **case** expression might not take the **default** branch.

In code example 2.5c, an assertion is placed in the **case default** replacing the **$display** statement in the code example from 2.4b. In this code, the assertion is present and the X is also propagated.

```
always_comb begin
  case (instruction) inside
    c0???: opcode = instruction[2:0]; //only test msb bit
    4'b1000: opcode = 3'b001;
    ... // decode other valid instructions
    default:
      begin
        assert (^{instruction}!== 1'bx);
          else $error("case_Sel = X");
      opcode = 'X;
  endcase
end
```

Example 2.5c – Case Inside

The author recommends using assertions prior to the case statement to monitor all **case/casez/casex/case inside** statements for unknown **case** expression values. Assertions can be disabled as a whole or individually, so designers will not get false negatives during reset or coming out of low-power modes. Assertions are automatically ignored by synthesis; therefore, designers will not need to add the extra **translate_off/translate_on** lines. Note that with the assertion preceding the case statements to trap **case** expression unknowns, there is very little difference between the **casex** and **casez**.

With **case inside** available now, the old recommendation to use **casez** over **casex** should now be rescinded. The new SystemVerilog **case inside** replaces both **casez** and **casex** and provides functionality that matches the synthesis model. As noted in the previous paragraph, **case inside** case statements should be preceded by an assertion to monitor for unknown bit in the case expression.

2.6 The Conditional Operator " ?: " Option

An approach many designers use, with the intention of propagating all **X**'s, is to use the conditional (commonly referred to as the ternary) operator for conditional combinational logic. The belief is that if the condition expression for the operator is **X**, the **?:** operator will always output an X. This is not always true under all conditions. Consider the following code snippet:

```
logic [3:0] out, data0, data1;
logic sel;

always_comb
  out = sel ? data1 : data0;
```

Example 2.6a – Conditional Operator

In this code, there are three possible outcomes based on the value of sel.

1. when **sel == 1'b0**, **out** is assigned **data0**
2. when **sel == 1'b1**, **out** is assigned **data1**
3. when **cc** or
   sel == 1'bZ, the simulator will test each bit position of the two data words.

When the bits match in value, that bit value will pass on to the output. When the bits differ, an **X** will be assigned, as shown in line three of the following table:

sel	data1	data0	out
4'b0	4'b1100	4'b0101	4'b0101
4'b1	4'b1100	4'b0101	4'b1100
4'bX	4'b1xx0	4'b1zz0	4'b1xx0
4'bX	4'b1100	4'b1100	4'b1100

In the third test above, **sel == 1'bX** and some of the **out** bits are assigned to **X**. However, in the fourth test, where again **sel == 1'bX**, **out** is "known" and the **X** in **sel** is not propagated.

Therefore, even with the conditional operator, an immediate assertion should be used to monitor for X's.

```
logic [3:0] out, data0, data1;
logic sel;

always_comb begin
  assert (^{sel}!== 1'bx);
  else $error("%m sel = X", $time);
  out = sel ? data1 : data0;
end
```

Example 2.6b – Conditional Operator with immediate assertion

3.0 Synthesis Case Directives History

In the early days of synthesis, Synopsys defined two synthesis directives called **full_case** and **parallel_case**. The company widely encouraged the use of these directives for all case statements because they could, in some cases, make a design smaller and faster. The problem is that this optimization often changed the design to an incorrect design. Many papers[2] have been written regarding the correct usage of **full_case** and **parallel_case** . The following sections briefly review and describe the issues with these directives. Section 4.0 will provide SystemVerilog 2005 [5] enhancements and SystemVerilog 2009 [2] updates.

3.1 Synthesis Directive full_case

Using the synthesis tool directive **full_case** gives more information about the design to the synthesis tool than is provided to the simulation tool. This particular directive is used to inform the synthesis tool that the **case** statement is fully defined, and that the output assignments for all unused cases are "don't cares". The functionality between pre- and post-synthesized designs may or may not remain the same when using this directive.

In code example 3.1a, a **case** statement is coded without using any synthesis directives. The resultant design is a decoder built from 3-input **AND** gates and inverters. The pre- and post-synthesis simulations will match.

```
// no full_case
// Decoder built from four 3-input AND gates
//   and two inverters
module mod_31a
```

[2] See list of papers in reference section

```
  (output logic [3:0] y,
   input        [1:0] a,
   input              en);

  always_comb begin
    y = 4'h0;  // latch prevention default assignment
    case ({en,a})
      3'b1_00: y[a] = 1'b1;
      3'b1_01: y[a] = 1'b1;
      3'b1_10: y[a] = 1'b1;
      3'b1_11: y[a] = 1'b1;
    endcase
  end
endmodule:mod_31a
```

Example 3.1a – Decoder with no synthesis directives

Code example 3.1b uses a **case** statement with the synthesis directive **full_case**. The only difference between code example 3.1a and 3.1b is the synthesis directive. Because of the synthesis directive, the **en** input is optimized away during synthesis and left as a dangling input. The pre-synthesis simulator results of the modules from both examples 3.1a and 3.1b will match the post-synthesis simulation results of example 3.1a, but will not match the post-synthesis simulation results of example 3.1b.

```
// full_case applied
// Decoder built from four 2-input AND gates
//   and two inverters
module mod_31b
  (output logic [3:0] y,
   input        [1:0] a,
   input              en);

  always_comb begin
    y = 4'h0;  // latch prevention default assignment
    case ({en,a})  // synopsys full_case
      3'b1_00: y[a] = 1'b1;
      3'b1_01: y[a] = 1'b1;
      3'b1_10: y[a] = 1'b1;
      3'b1_11: y[a] = 1'b1;
    endcase
  end
endmodule:mod_31b
```

Example 3.1b – Decoder with synthesis directive full_case

The synthesis result from the example 3.1b is a decoder, synthesized into four 2-input **NOR** gates and two inverters. The enable input is left unused as it was optimized away. The full_case treats

the **case** items as entries to a Karnaugh map. All non-specified **case** item conditions are considered as "don't care" in the Karnaugh map. Since **en** is set for the four **case** items listed and is a "don't care" for the remaining conditions due to the full_case pragma, it is optimized away. Bummer!

As widely defined in previous papers [1] [4] on synthesis directives, the guideline is to only use the **full_case** directive with the inverse-case statement one-hot coding style.

```
// full_case applied to one-hot state machine
...
logic [3:0]  state, next_state;
...
always_comb begin  //next state logic decode
  next_state = '0;  // latch prevention
  case (1'b1)  // synopsys full_case
    state[0]: next_state[1] = 1'b1;
    state[1]: next_state[2] = 1'b1;
    state[2]: next_state[3] = 1'b1;
    state[3]: next_state[0] = 1'b1;
  endcase
end
...
```

Example 3.1c – Best Practice use of synthesis directive full_case

3.2 Synthesis Directive parallel_case

Using the synthesis tool directive **parallel_case** also gives more information about the design to the synthesis tool than is provided to the simulation tool. This particular directive is used to inform the synthesis tool that all cases should be tested in parallel, even if there are overlapping cases which would normally cause a priority encoder to be inferred. When a design does have overlapping cases, the functionality between pre- and post-synthesis designs will be different.

Years ago, when adding **parallel_case** was the "in" thing to do, one consultant related the experience of adding **parallel_case** to an RTL design to improve optimized area and speed. The RTL model (behaving like a priority encoder) passed the test bench, but testing missed that the gate-level model was implemented as non-priority parallel logic. Result: the design was wrong, the simulation/synthesis mismatch was not discovered until ASIC prototypes were delivered, and the ASIC had to be redesigned at significant cost in both dollars and schedule. Today, these types of errors should be found using equivalence checking tools such as Formality. Better still is to follow methodologies that guard against such modeling problems.

The pre-synthesis simulations for the modules in examples 3.2a and 3.2b below, as well as the post-synthesis design of the module in example 3.2a, will infer priority encoder functionality.

```
// no parallel_case
// Priority encoder - 2-input nand gate driving an
//   inverter (z-output) and also driving a
//   3-input AND gate (y-output)
module mod_32a
  (output logic y, z,
   input        a, b, c, d);

  always_comb begin
    {y, z} = 2'b0;
    casez ({a, b, c, d})
      4'b11??: z = 1;
      4'b??11: y = 1;
    endcase
  end
endmodule:mod_32a
```

Example 3.2a – Priority Encoder Decoder no synthesis directives

The post-synthesis structure for module in example 3.2b will be two **AND** gates. The use of the synthesis tool directive **parallel_case** will cause priority encoder case statements to be implemented as parallel logic, causing pre- and post-synthesis simulation mismatches.

```
// parallel_case
// Priority encoder – (not really)
//   two 2-input AND gates
module mod_32b
  (output logic y, z,
   input        a, b, c, d);

  always_comb begin
    {y, z} = 2'b0;
    casez ({a, b, c, d}) // synopsys parallel_case
      4'b11??: z = 1;
      4'b??11: y = 1;
    endcase
  end
endmodule:mod_32b
```

Example 3.2b – Priority Encoder Decoder synthesis directive parallel_case

As widely defined in previous papers [1][4] on synthesis directives, the guideline is to only use the **parallel_case** directive with the inverse-case statement one-hot coding style in conjunction with **full_case**.

```
// full_case applied to one-hot state machine
...
logic [3:0]  state, next_state;
...
always_comb begin  //next state logic decode
  next_state = '0;  // latch prevention assignment
  case (1'b1)  // synopsys full_case parallel_case
    state[0]: next_state[1] = 1'b1;
    state[1]: next_state[2] = 1'b1;
    state[2]: next_state[3] = 1'b1;
    state[3]: next_state[0] = 1'b1;
  endcase
end
...
```

Example 3.2c – Best Practice use of synthesis directives

4.0 SystemVerilog Enhancements

In an attempt to bring to the simulation environment the same capabilities that **`full_case`** and **`parallel_case`** provide in synthesis, SystemVerilog 2005[3] added two **`case`** decision modifiers. These new **`case`** decision modifiers are called **`priority`** and **`unique`**. Both of these decision modifiers come with built-in assertion checking to help prevent unexpected results. Many papers, such as:

> "*SystemVerilog Saves the Day—the Evil Twins are Defeated! 'unique' and 'priority' are the new Heroes*"[6]
>
> "*SystemVerilog's priority & unique—A Solution to Verilog's 'full_case' & parallel_case' Evil Twins!*"[5]

have been written about these two features, discussing how they improve upon, and are a good replacement for, **`full_case/parallel_case`**. As noted in the conclusions of the Cummings paper [5] and further elaborated previously in this paper, these decision modifiers still should be used with caution.

Verilog's **`if...else`** and **`case`** statements (including **`casez`** and **`casex`**) have four gotchas that often result in design problems:

- Not all possible branches need to be specified (incomplete decisions)
- Redundant (duplicate) decision branches can be specified
- Software simulation evaluates decisions in the order listed (priority decoding), but the decision might be able to be evaluated in any order in hardware (parallel decoding).
- X-hiding

The **`priority`** and **`unique`** modifiers eliminate the gotchas listed above with incomplete and redundant decision statements, and prevent the gotchas common to **`synopsys full_case`**

and **parallel_case** pragmas. The benefits of the **unique** and **priority** decision modifiers are described in two other SNUG papers [5] [6].

There are still some gotchas left hanging around when using these modifiers. First, these modifiers only illuminate, or warn against, some of the conditions that cause latches. Section 4.5 will show a way to always illuminate latches. Second, X-hiding is still hanging around. The techniques shown in section 2.5 about using assertions and in the paper "*Being Assertive With Your X*" [8] provide methods to trap X's around all of these other features.

4.1 SystemVerilog priority

The **case priority** decision modifier tells the tools that the **case** items must be evaluated in the order listed, which is already the default in SystemVerilog. What this decision modifier really provides is a run-time violation report if the **case** statement is entered and there is no matching condition.

The most practical usage of **priority case** is with inverse-case statements as shown in code example 4.1a.

```
always_comb
  priority case (1'b1)
    state[0]: nextstate = 3'b010;
    state[1]: nextstate = 3'b100;
    state[2]: nextstate = 3'b001;
  endcase
```

Example 4.1a – priority case

The code example 4.1a above will not give a violation report if multiple conditions match, i.e. **state === 3'111**. A violation report will occur if there are no matches, i.e. **state === 3'b000**. If there are no matches, then the code is modeling a latch condition. But what if the following condition occurs when **state !== 3'b000**?

```
always_comb
  priority case (1'b1)
    state[0]: nextstate1 = 3'b010;
    state[1]: nextstate2 = 3'b100;
    state[2]: nextstate1 = 3'b001;
  endcase
```

Example 4.1b – priority case with latches

In example 4.1b, no violation reports are reported while state !== 3'b000, yet latches will be modeled, because all the outputs have not been assigned for all the conditions.

One other word of caution regarding the keyword "*priority*": a **priority case** would appear to imply that the order of a multi-branch decision statement will be maintained by synthesis. DC

does not do this. DC will still optimize **priority case** decision ordering, the same as with a regular **case** decision statement. While gate-level optimization is a good thing, it is a *gotcha* if the designer is expecting a **priority case** statement to automatically have the identical priority decoding logic after synthesis.

4.2 SystemVerilog unique

The **unique case** decision modifier tells the tools that the **case** items may be evaluated in parallel and that all the items listed are the complete set of items to be considered. The tools will give a violation report if overlapping **case** items exist. The tools will give a run-time violation report if the **case** statement is entered and there are no matching **case** items. This decision modifier is similar to the combined **full_case/parallel_case** synthesis pragmas.

```
always_comb
  unique case (1'b1)
    state[0]: nextstate = 3'b010;
    state[1]: nextstate = 3'b100;
    state[2]: nextstate = 3'b001;
  endcase
```

Example 4.2a – unique case

The code example 4.2a above will give a violation report if multiple conditions match, i.e. **state === 3'111**. A violation report will also occur if there are no matches, i.e. **state === 3'b000**. Again, what if the following condition occurs?

```
always_comb
  unique case (1'b1)
    state[0]: nextstate1 = 3'b010;
    state[1]: nextstate2 = 3'b100;
    state[2]: nextstate1 = 3'b001;
  endcase
```

Example 4.2b – unique case with latches

In example 4.2b, if one and only one bit of **state** is set when this **case** statement is tested, no violation reports are generated, yet latches will be modeled, because all the outputs have not been assigned for all the conditions.

4.3 SystemVerilog unique0

SystemVerilog 2009[2] added **unique0 case** decision modifier, bringing in one more **case** decision modifier for designers to choose from. The **unique0 case** decision modifier tells the tools that the **case** expression is to only match, at most, one **case** item (no overlapping **case** items), but is not required to match any **case** items. This is different from plain **unique** in that it does not require a match. At first glance, this may appear to be of no use, but in reality, this decision modifier allows a latch prevention methodology to be used which was not available

previously with just the **unique case**. The tools will also give a violation report if overlapping **case** items exist. This decision modifier is similar to the **parallel_case** synthesis pragma.

```
always_comb
  unique0 case (1'b1)
    state[0]: nextstate = 3'b010;
    state[1]: nextstate = 3'b100;
    state[2]: nextstate = 3'b001;
  endcase
```

Example 4.3a – unique0 case

The code example 4.3a above will give a violation report if multiple conditions match, i.e. **state === 3'111**, but will not give a violation report if there are no matches, i.e. **state === 3'b000**.

*At the time this paper was written, the **unique0** construct was not supported by simulation or synthesis tools tested by the author.*

4.4 SV 2009 Violation Report

In SystemVerilog 2005, a warning was issued when **unique** or **priority case** conditions warranted reporting problems such as no matching **case** items. Only warnings were issued because vendors did not want to report false errors. SystemVerilog 2009 replaces this warning with a "violation report". A violation report will default to issuing a warning, but the user can elevate the reporting of a "violation report" to other levels, such as an error.

One side note regarding violation reporting is that the tools will make these violations immune to false reports due to zero-delay glitches in the active region.

4.5 Only You Can Prevent Latches!!![3]

The **case** examples discussed thus far in this paper showed various coding tricks and SystemVerilog **case** decision modifiers that help reduce the possibility of a combinational logic **case** statement inadvertently modeling latches. In all these examples, the focus is strictly on the **case** item and the **case** expression matching. Multiple matches infer priority encoder behavior. No match infers a latch. Novice engineers may jump to the conclusion that all that is missing to prevent latches then is a **case** item **default** condition – wrong. The **default** does the same as a **full_case** or **unique**, in that it guarantees that a match will always be made. In fact, **unique** will never warn of the no-match condition when a default is present, since the default will cover all remaining non-specified decodes of the **case** expression.

If all the **case** items and **case** expressions are fully covered, can there still be latches? Consider the following **case** example:

[3] See Brekstickers [9] for additional details on this subject

```
always_comb
  unique case (sel)
    cond1: begin
        out1 = in1a;
        out2 = in2a;
      end

    cond2: out2 = in2b;

    cond3: out1 = in1c;

    default:
      begin
        out1 = in1a;
        out2 = in2a;
      end
  endcase
```

Example 4.5a – unique case with default and latches

In code example 4.5a, the **case** is fully defined because of the **default**. The **unique case** would still issue violation reports if there were overlapping conditions, which do not exist in this example. Unfortunately, since not all outputs are defined for all the states, latches will also be inferred. It does not matter that the **default** lists all the outputs, the **default** condition will only be reached when there are no other **case** item matches.

One solution is to assign all outputs within all conditions, as shown in the next code example.

```
always_comb
  unique case (sel)
    cond1: begin
        out1 = in1a;
        out2 = in2a;
      end

    cond2: begin
        out1 = in1a;
        out2 = in2b;
      end

    cond3: begin
        out1 = in1c;
        out2 = in2a;
      end

    default:
```

```
        begin
          out1 = in1a;
          out2 = in2a;
        end
    endcase
```

Example 4.5b – unique case with all outputs assigned for each case item

The approach shown in example 4.5b does work, but it can become very hard to read and maintain when there are many outputs from this combinational block. I have worked with some state machines that drive ten or more outputs from state decoding, like in this example. Additionally, output assignments that are unique from state to state get lost in all the redundant output assignments. Contrast the two **always** blocks above. In first example, it is easy to see the unique output conditions for each decode. In the second example, it can be easy to miss differences, and these examples have only two outputs.

Consider the following code example where output defaults are placed at the beginning of the combinational **always** block. Now only the conditions that modify the initial defaults need to be decoded.

```
always_comb begin
  out1 = in1a;
  out2 = in2a;
  case (sel)
    cond2: out2 = in2b;
    cond3: out1 = in1c;
  endcase
end
```

Example 4.5c – case with defaults listed before case statement

This code is very concise and is exactly the same functionality as example 4.5b. This coding style lists the defaults first, before any conditional **if** or **case** statements. Then it uses the conditional statements **case** and/or **if** to modify the outputs as needed. Only the conditions that cause the output to be different from the default will need to be listed.

What happens if **unique** is added to the code in example 4.5c, providing the checks and violation reports given by the **unique** decision modifier?

```
always_comb begin
  out1 = in1a;
  out2 = in2a;
  unique case (sel) // bad design – don't use!!
    cond2: out2 = in2b;
    cond3: out1 = in1c;
  endcase
```

```
end
```

Example 4.5d – unique case with defaults listed before case statement

Using **unique** with defaults listed outside the **case** statement will not synthesize to the same design as simulated. The **unique case** in code example 4.5d will synthesize to simply **out1 = in1c** and **out2 = in2b**. Wow, talk about logic reduction! This occurs because **unique case** implies to the synthesis tool that all the conditions cared about are listed in the **case** statement and all others are "don't cares" (like Karnaugh map "don't cares"). Since the defaults are listed before (and outside) the **case** statement, they are ignored by synthesis when **unique case** is used.

A better solution would be to use the unique0. This will only check for non-overlapping case items and does not require a matching case item. Synthesis will not logic reduce away the default outputs with this case decision modifier.

```
always_comb begin
  out1 = in1a;
  out2 = in2a;
  unique0 case (sel) // GOOD design – USE IT (When supported)
    cond2: out2 = in2b;
    cond3: out1 = in1c;
  endcase
end
```

Example 4.5e – unique0 case with defaults listed before case statement

5.0 Asynchronous Set/Reset Flip-Flop Bug

One of the biggest "ignored" bugs of the Synopsys Synthesis HDL reader (Presto) is the required, but functionally incorrect, coding style for asynchronous Set/Reset Flip-Flops (SRFF's).

```
// DFF with asynchronous set and reset
// required Synopsys coding style for Synthesis
//   This model can fail in simulation
module mod_50a
  (output logic q,
   input  d, clk, rstn, setn);

  always_ff @(posedge clk or negedge rstn or negedge setn)
    if      (!rstn)  q <= 0;  // asynchronous reset
    else if (!setn)  q <= 1;  // asynchronous set
    else             q <= d;
endmodule:mod_50a
```

Example 5.0a –synthesizable asynchronous set/reset DFF

As a consultant and trainer, I am constantly asked about this coding style. When I was a junior designer, I was under the opinion that if you are doing synchronous design, you should never need an asynchronous SET/RESET FF. The opinion was that once you initialized your FF with either a SET or a RESET, you would not be applying the asynchronous SET/RESET signal again during the simulation. Later in my career, I worked for companies that use an active asynchronous SET/RESET after the initialization phase of the simulation. These devices use an approach to reconfigure the chip after the chip starts actively running. That is, once the initial setup is complete, the asynchronous **reset** is applied to the configuration circuit a second time. During this second **reset** mode, the configuration settings are applied to the asynchronous **set**/**reset** inputs to the FF's in the configuration circuit. Next the **reset** is removed, allowing for the asynchronous configuration settings to be applied to the FF's. The FF's with the **set** still active should change from their **reset** state to their **set** state at this point. Unfortunately, the FF model in example 5.0a will hold its **reset** state until the next clock cycle starts. If the **set** is removed before this clock occurs, the **set** value is never registered and the simulation fails. In an actual FF, this approach works fine because the **set**/**reset** inputs are true level-sensitive inputs. But with the SystemVerilog FF modeling restrictions for synthesis, simulation fails!!!!!!! This means that the model for synthesis does not represent the actual design.

What if the **negedge** condition is removed from the sensitivity list, so that the code is sampled on the trailing edge of **reset,** as well as the leading edge?

```
// Bad DFF with asynchronous set and reset.  This design
//   will not compile from Synopsys, and the design will
//   also not simulate correctly.
module mod_50b
  (output logic q,
   input  d, clk, rstn, setn);

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

endmodule:mod_50b
```

Example 5.0b –non-synthesizable bad design asynchronous set/reset DFF

The code in example 5.0b is non-synthesizable because the synthesis reader requires that if one item in a sensitivity list has an edge specified, then all the items in the sensitivity list must have edges specified. Also, the RTL model does not match the intended model. Whenever a **set** or **reset** goes from low to high, the block is entered and unintended clocks could be modeled. Assume **clock**, **reset**, and **set** are all at a high state. **Reset** then goes low for a while and then back high. When **reset** goes low, the **always** block will be entered, and **q** will be put in its reset state. However, when **reset** goes back high, the **always** block will be entered again, and the **if** condition check will fall through to the final **else** test, assigning **q <= d**. In this

case, the rising edge of **reset** will cause a false clock to occur, and **q** will be assigned to **d** erroneously.

Over the years, there have been many solutions proposed to this problem. One solution described by a SNUG paper in 1999[1] recommended using non-synthesizable **force** and **release** constructs. The **force** and **release** statements will force a correct pre-synthesis model to accurately model the post-synthesis model.

```
// Good DFF with asynchronous set and reset
//  and self-correcting set-reset assignment
module mod_50c
  (output logic q,
   input  d, clk, rstn, setn);

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

  // synthesis translate_off
  always @(rstn or setn)
    if   (rstn && !setn) force q = 1;
    else                 release q;
  // synthesis translate_on

endmodule:mod_50c
```

Example 5.0c – synthesizable DFF with asynchronous set/reset

The solution in example 5.0c works, but is ugly for many reasons. First, in simulation, the signals are assigned from multiple **always** blocks, meaning the **always** block modeling the SRFF cannot use the SystemVerilog **always_ff**, eliminating the RTL simulation checks that are provided by **always_ff**. Second, the use of **force/release** is used to override the real code. When is the signal assigned from the design **always** block and when is it overridden by the **force/release**? Third, the signals are assigned by both blocking and non-blocking assignment operators. These issues violate RTL for synthesis guidelines, hence the **synthesis translate_off, synthesis translate_on** switches.

Consider the following as a cleaner solution. This solution only uses the original **always** block. The **synthesis translate_off, synthesis translate_on** comment statements are replaced by a compiler directive which is automatically set when the code is read by the synthesis tool. The code inside the compiler directive **'ifndef ... 'endif** adds a special sensitivity list entry condition in the sensitivity list.

```
// Trigger when rst_n goes high and set_n is low
always_ff @(    posedge clk
             or negedge rst_n
             or negedge set_n
 `ifndef SYNTHESIS
             or posedge (rst_n & ~set_n)
 `endif
           )
  if      (!rst_n) q <= '0;
  else if (!set_n) q <= '1;
  else             q <=  d;
```

Example 5.0d – better model of DFF with asynchronous set/reset that both simulates and synthesizes correctly

My proposal to Synopsys is to make a special input condition to Presto to allow/ignore this additional test in the sensitivity list so that even the conditional compilation **`ifdef/`endif** or **translate_off/translate_on** pragmas could be illuminated.

6.0 Old logic Type vs. New logic Value Set

In the pre-IEEE SystemVerilog days, SuperLog introduced rash new ideas, laying the foundation for Verilog enhancements that later, combined with Verilog and other recommendations, became SystemVerilog. One of the significant SuperLog design enhancements, that is now part of System Verilog, is the term **logic,** introduced as a complete replacement for **reg**. SuperLog also modified the usage for **logic** or **reg** variables such that they could be used anywhere a **wire** could be used, but restricted in that they must only have a single-source driver. This means that a designer could use the **logic** type everywhere in the design, except when a signal has multiple drivers. This gives us a simplistic, simple type selection, removing the confusing data type rules of Verilog. For those designers who have taken this approach (and there are many), there is a significant change in the way simulators are treating **logic** today. As a support consultant, I have seen simulation problems resulting from this issue.

The SystemVerilog 2009 standard made a subtle change to the meaning of the keywords **reg**, **logic** and **bit**. Prior to SV-2009, these keywords were considered to be declarations of variables. In SV-2009, they were changed to be indicators of the **logic** value set that either a variable or net could use. The **logic** and **reg** keywords are synonymous; both indicate a 4-state kind. The **bit** keyword indicates a 2-state kind. The **wire** net types, and all other net types, are always a logic (4-state) value set. Some variable types, such as integer, are logic (4-state) value set, while other variable types, such as **byte** and **int**, are bit (2-state) value set. The pure SystemVerilog declaration of a signal is: type "value set" size name. For example:

```
var  logic [7:0] a,
wire logic [3:0] b,
```

What gets confusing is that, for backward compatibility, **bit** and **logic** can be used without a type and will infer the SystemVerilog variable type **var**. In other words, using **bit** or **logic** alone infers variables of type **var bit** or **var logic**. Also for backward compatibility, the net type **wire** by itself infers **wire logic**. The type **var** is ugly and you will most likely never see it or use it, with the one exception shown below.

Until recently, SystemVerilog simulators followed the SuperLog and IEEE 1800-2005[3] implementation of defining **logic** as a variable type everywhere it was used. In the recent implementation of some simulators, the usage definition of **logic** has changed to be compliant with IEEE 1800-2009[2] standard using **logic** as a value set, rather than a type.

Remember, this discussion applies to those designers who are using **logic** as a single source variable everywhere and only using net types for multi-driven signals. The problem (and this is significant) is that now **logic** will infer a "variable logic" for all usages, except for input ports. Under SV-2009 rules, an input port declared as **input logic** infers **input wire logic**, not **input var logic**. This is an important difference! An input port that is a net data type can have multiple drivers, including an internal continuous assignment that "back drives" the input port. An input port that is a variable type is restricted to a single source (driver). Back driving an input port that is a variable is not allowed. Designers who use the **logic** only implementation, and want **input** ports to be variables, now need to update their already verbose **input** port declaration. To actually get a variable type **input** port, the designer needs to add the SystemVerilog variable type **var** to the input port declaration.

```
module mod_input
 (input     logic a, b,  // implies net (wire) input port
  input var logic c, d,  // implies variable input port
…
```

YUCK!

7.0 Conclusions and Guidelines

This paper discussed and provided solutions to issues that designers using SystemVerilog for design must address, such as:

- Case expression issue for **`casez`** and **`casex`**
- Latches generated when using **`unique case`** or **`priority case`**
- SRFF coding style problems with synthesis
- SystemVerilog 2009 new definition of logic

The SystemVerilog **`case inside`** is a good replacement for **`casez`** and **`casex`**. Adding an assertion preceding each **`case`** statement (**`case`** or **`case inside`**) to monitor for unknowns in the **`case`** expression contributes to a very robust design. The assertions could also illuminate the need for X propagation through RTL code since the X's are now visible.

The old synthesis pragmas **`full_case`** and **`parallel_case`** were attempted to be replaced by SystemVerilog **`case`** decision modifiers such as **`unique`**. The idea was to bring the same functionality with built-in checks to the simulator that existed in the synthesis tool. Unfortunately, these **`case`** definition modifiers can only help to reduce unintended latches, they cannot cover all the conditions that cause unintended latches. The only way to fully prevent unintended latches in combinational logic blocks is to assign every output for every condition. This can be done two ways: first, by literally assigning all outputs within the decodes of all the conditions. This style requires lots of code and much redundancy. The second, and by far less verbose method, is to assign all the outputs at the top of the combinational logic block, before any conditional statements. Then, within the conditional statements, only decode and assign the conditions that would change the output from the default assignments previously declared.

The asynchronous **`set`**/**`reset`** flip-flop model required by synthesis is functionally wrong and must have a fix applied to make it simulation right for RTL. This must not be ignored.

Finally, the designers who have taken on the modeling style of declaring all single-driven signals as **`logic`** types, and strictly use **`wire`** (or **`tri`**) only for multi-driven signals, must now deal with a change of definition. The usage of **`logic`** is the same as before in all cases, except for module **`input ports`** where the default is **`wire`** even if **`logic`** is listed. Yuck! Depending on how pure the designers want to be, if the desire is to follow the previously stated guideline, then the **`input ports`** must be declared as **`var logic`**. Yuck! Yuck! Yuck! (Let me tell you what I really think about this.)

8.0 References

[1] Don Mills and Clifford Cummings, *"RTL Coding Styles That Yield Simulation and Synthesis Mismatches,"* in SNUG 1999 Proceedings.

[2] *"IEEE 1800-2009 IEEE Standard SystemVerilog – Unified Hardware Design, Specification and Verification Language,"* IEEE, New York, NY, 2009. ISBN 978-0-7381-6129-7.

[3] *"IEEE 1800-2005 IEEE Standard SystemVerilog – Unified Hardware Design, Specification and Verification Language,"* IEEE, New York, NY, 2005. ISBN 0-7381-4810-5.

[4] Clifford Cummings, *"'full_case parallel_case', the Evil Twins of Verilog Synthesis,"* SNUG Boston, 1999.

[5] Clifford Cummings, *"SystemVerilog's priority & unique—A Solution to Verilog's 'full_case' & parallel_case' Evil Twins!,"* Israel SNUG, 2005

[6] Stuart Sutherland, *"SystemVerilog Saves the Day—the Evil Twins are Defeated! 'unique' and 'priority' are the new Heroes,"* San Jose SNUG, 2005

[7] Stuart Sutherland and Don Mills, *"Standard Gotchas, Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know,"* Boston SNUG, 2006

[8] Don Mills, *"Being Assertive With Your X,"* Boston SNUG 2004

[9] Shalom Bresticker, *"Just When You Thought It Was Safe to Start Coding Again…Return of the SystemVerilog Gotchas"* Israel SNUG 2009, Boston SNUG 2008

9.0 Acknowledgements

Thanks are in order to those who have helped review this paper for accuracy. Ilana Flyer, Cliff Cummings, Stuart Sutherland and Bernard Miller have provided enormous technical support in addition to basic format and grammar checking. Thanks to all others who have volunteered and offered their services to help proofread this paper.

10.0 About the Author

Mr. Don Mills has been involved in ASIC design since 1986. During that time, he has worked on more than 30 ASIC projects. Don started using top-down design methodology in 1991 (Synopsys DC 1.2). Don has developed and implemented top-down ASIC design flows at several companies.

His specialty is integrating tools and automating the flow. Don works for Microchip Technology Inc. as an internal SystemVerilog consultant. Don is a member of the IEEE Verilog and System Verilog committees that are working on language issues and enhancements. Don has authored and co-authored numerous papers, such as *"SystemVerilog Assertions are for Design Engineers Too!"* and *"RTL Coding Styles that Yield Simulation and Synthesis Mismatches"*. Copies of these papers can be found at www.lcdm-eng.com. Mr. Mills can be reached at

mills@lcdm-eng.com or don.mills@microchip.com