SystemVerilog's priority & unique - A Solution to Verilog's "full_case" & "parallel_case" Evil Twins!

Clifford E. Cummings

Sunburst Design, Inc.

ABSTRACT

At Boston SNUG 1999, I introduced the evil twins of Verilog synthesis, "**`full_case`**" and "**`parallel_case`**.[2]" In the 1999 Boston SNUG paper I pointed out that the **`full_case`** and **`parallel_case`** directives are always most dangerous ... when they work!

At that time, the best guideline to follow was to avoid using **`full_case`** and **`parallel_case`** in all Verilog designs, with one noteworthy exception: add **`parallel_case`** for a special Verilog onehot **`case`** statement encoding style.

The SystemVerilog standard[12] has introduced two **`case`**-statement and **`if`**-statement modifiers, using the new SystemVerilog keywords "**`priority`**" and "**`unique`**."

The new keywords, **`priority`** and **`unique`** are part of the SystemVerilog language, not just comment-style directives, which means that simulation, synthesis and formal tools can all recognize and consistently implement proper cross-tool functionality and testing for RTL code written with these new constructs.

This paper summarizes the problems associated with the use of the **`full_case`** and **`parallel_case`** directives and details how the new SystemVerilog **`priority`** and **`unique`** keywords solve these problems while adding valuable RTL synthesis capabilities to the new and powerful SystemVerilog language.

1.0 The legend of full_case parallel_case

Prior to 1999, I found that engineers routinely added **`full_case parallel_case`** to all RTL **`case`** statements. Indeed, **`full_case parallel_case`** had been two of the most over-used and abused synthesis directives ever employed by Verilog synthesis design engineers. The reasons cited most often by engineers for using **`full_case parallel_case`** were:

- **`full_case parallel_case`** makes my designs smaller and faster.
- **`full_case`** removes latches from my designs.
- **`parallel_case`** removes large, slow priority encoders from my designs.

The above reasons were (and still are) either inaccurate or dangerous.
- Sometimes these directives don't affect a design at all.
- Sometimes these switches make a design larger and slower.
- Sometimes these directives change the functionality of a design.

And finally:

The full_case and parallel_case directives are always most dangerous when they work!

For these reasons, I call **`full_case`** and **`parallel_case`**, ***"the evil twins of Verilog synthesis!"***

This paper re-visits the **`full_case`** and **`parallel_case`** problem, and details how the new SystemVerilog enhancements, **`unique`** and **`priority`**, give the desired **`full_case parallel_case`** benefits without the dangers these evil twins potentially introduce to a design.

1.1 Synopsys tools, versions and useful command aliases

For this paper, Verilog and SystemVerilog experiments were conducted using the VCS Verilog simulator, version 7.2, and Design Compiler (DC) with Design Vision GUI, version V-2004.06-SP2. At the time of this publication, Design Compiler required a special license to enable recognition of SystemVerilog features. Engineers interested in using Design Compiler SystemVerilog features should contact their local sales or field personnel. I also used the following command aliases to run my Verilog-2001 & SystemVerilog simulations

VCS Verilog-2001 (**`+v2k`**) aliases to: compile & run; compile, run & dump

```
alias     vcsr="vcs     -R +v2k"
alias  vcsdump="vcs -PP -R +v2k +define+VPD"
```

VCS SystemVerilog (**`-sverilog`**) aliases to: compile; compile & run; compile, run & dump

```
alias     svcs="vcs        -sverilog +define+SV"
alias    svcsr="vcs     -R -sverilog +define+SV"
alias svcsdump="vcs -PP -R -sverilog +define+SV +define+VPD"
```

NOTE: the `-sverilog` switch is new with VCS7.2. Earlier versions of VCS used the switch `+sysvcs` to enable SystemVerilog simulation. The `-sverilog` switch is an improvement because now both VCS and Design Compiler use a similar `sverilog` switch name, as shown below.

Within Design Vision, the tcl command that is used to read SystemVerilog files is:

`read_sverilog` <*filename*>

1.2 Understanding Verilog case statements

To fully understand all the topics discussed in this paper, the reader must have a good understanding of Verilog `case` statements and the definitions of the constructs that make up the `case` statement. If you need a better working knowledge of Verilog `case` statement constructs, you should review Section 15.0 before proceeding with the rest of this paper.

2.0 Verilog & SystemVerilog case statement modifiers

Before going into detail about all of the `case` statement modifiers, we should look at the big picture as it relates to `full_case`, `parallel_case`, `priority` and `unique`.

First ask yourself the questions: Why are the `full_case parallel_case` synthesis directives so dangerous? And is there a way to avoid the danger while retaining their positive capabilities?

2.1 The full_case parallel_case dangers

The `full_case` and `parallel_case` directives are dangerous because they tell the synthesis tool something different about the design than what is told to the simulator.

To the Verilog simulator, `full_case` and `parallel_case` are buried inside of Verilog comments and are completely ignored. To the synthesis tool, `full_case` and `parallel_case` are command-directives that instruct the synthesis tools to potentially take certain actions or perform certain optimizations that are unknown to the simulator.

This explains the claim that the `full_case` and `parallel_case` directives are always most dangerous when they work!

The functionality and pitfalls of the `full_case` and `parallel_case` directives are explained in Section 3.0 and Section 5.0 of this paper.

2.2 The SystemVerilog priority and unique modifiers

The only way to avoid the dangers related to adding `full_case` and `parallel_case` is to give the simulator the same understanding of these directives as the synthesis tool. This has been accomplished by adding the `priority` and `unique case`-statement and `if`-statement modifiers.

The new **`priority`** and **`unique`** keywords are recognized by all tools, including the SystemVerilog simulators, so now all the tools have the same information about the design. The capabilities of the **`priority`** and **`unique`** enhancements are described in Section 4.0 and Section 6.0 of this paper.

2.3 SystemVerilog priority and unique assertions

Not only do **`priority`** and **`unique`** modify **`case`**-statements and **`if`**-statements, they are also both assertions. The engineer ***asserts*** that specified **`case`**-statements and **`if`**-statements contain all possible values for the expressions being tested. If the run-time simulation finds an untested expression, the assertion fails and an error message is immediately reported, helping to identify potential bugs in the design.

The engineer may also optionally ***assert*** that specified **`case`**-statements and **`if`**-statements are unique for the expressions being tested and that no tested expression should match more than one of the **`case`**-items or **`if`**-branches. If the run-time simulation finds any tested expression that could match more than one tested **`case`**-item or **`if`**-branch, the assertion fails and an error message is immediately reported, again helping to identify potential bugs in the design.

As can be seen from this short discussion, because these key words can not only be used to optimize designs, but can also identify assumption errors on the part of the design engineer, these new modifiers prove to be valuable design assertions that can be used to help rapidly debug a design. These new modifiers are both command directives and design assertions at the same time. Very nice!

2.4 Should priority and unique always be used?

And finally, after viewing the enhanced capabilities of the **`priority`** and **`unique`** keywords discussed in the remainder of this paper, one might ask if *all* SystemVerilog **`case`**-statements and **`if`**-statements should be coded with the **`priority`** and **`unique`** modifiers? The answer is no!

A common and efficient RTL coding style example is included in Section 12.1 of this paper that shows a coding style and functionality that break when using the **`priority`** and **`unique`** modifiers.

3.0 What is a full case statement?

A *full* **`case`** statement is a **`case`** statement in which all possible **`case`**-expression binary patterns can be matched to a **`case`** item or to a **`case default`**. If a **`case`** statement does not include a **`case default`** and if it is possible to find a binary **`case`** expression that does not match any of the defined **`case`** items, the **`case`** statement is not full.

The examples in this section includes the **`case`** statement report that is generated when DC reads each Verilog example. For a description of the meaning of the **`full_case`** reports shown in these examples, go to Section 3.6.

3.1 HDL full case statement

From an HDL simulation perspective, a **`full case`** statement is a **`case`** statement in which every possible binary, non-binary and mixture of binary and non-binary patterns is included as a **`case`** item in the **`case`** statement. Verilog non-binary values are, and VHDL non-binary values include, **`z`** and **`x`** and are called metalogical characters by both the IEEE VHDL-1999 RTL Synthesis Standard[9] and the IEEE Verilog-2002 RTL Synthesis Standard[8].

3.2 Synthesis full case statement

From a synthesis tool perspective, a *full* **`case`** statement is a **`case`** statement in which every possible binary pattern is included as a **`case`** item in the **`case`** statement.

Verilog does not require **`case`** statements to be either synthesis or HDL simulation full, but Verilog **`case`** statements can be made full by adding a **`case default`**. VHDL requires **`case`** statements to be HDL simulation full, which generally requires an **`others`** clause.

Example 1 shows a **`case`** statement, with **`case default`**, for a 3-to-1 multiplexer. The **`case default`** causes the **`case`** statement to be full. During Verilog simulation, when binary pattern **`2'b11`** is present on the select lines, the **`y`**-output will be driven to an unknown, but synthesis tools will treat the **`y`**-output as a "don't care" for the same select-line combination, potentially causing a mismatch to occur between simulation and synthesis. To insure that the pre-synthesis and post-synthesis simulations match, the **`case default`** could assign the **`y`**-output to either a predetermined constant value, or to one of the other multiplexer input values; however, the **`x`**-assignment in this example is being used as an assertion and is valuable during simulation to identify unexpected values on the select inputs. The **`x`**-assignment will also be treated as a don't-care for synthesis, which may allow the synthesis tool to further optimize the synthesized design.

```
module mux3c
  (output reg       y,
   input      [1:0] sel,
   input            a, b, c);

  always @*
    case (sel)
      2'b00:   y = a;
      2'b01:   y = b;
      2'b10:   y = c;
      default: y = 1'bx;
    endcase
endmodule
```

Example 1 - A case default, full case statement

```
Statistics for case statements in always block at line 7 in file
        '.../mux3c.v'
===============================================
|           Line           |  full/ parallel  |
===============================================
|            9             |    auto/auto     |
===============================================
```

Figure 1 - Case statement report for a case statement with a case default

3.3 Non-full case statements

Example 2 shows a **`case`** statement for a 3-to-1 multiplexer that is not full. The **`case`** statement does not define what happens to the **`y`**-output when binary pattern **`2'b11`** is placed on the select (**`sel`**) lines. In this example, the Verilog simulation will hold the last assigned **`y`**-output value and synthesis will infer a latch on the **`y`**-output as shown in the latch inference report of Figure 2.

```
module mux3a
  (output reg       y,
   input      [1:0] sel,
   input            a, b, c);

  always @*
    case (sel)
      2'b00:   y = a;
      2'b01:   y = b;
      2'b10:   y = c;
    endcase
endmodule
```

Example 2 - Non-full case statement

```
Statistics for case statements in always block at line 7 in file
        '.../mux3a.v'
===============================================
|           Line           |  full/ parallel  |
===============================================
|            9             |     no/auto      |
===============================================

Inferred memory devices in process
    in routine mux3a line 7 in file
        '.../mux3a.v'.
===============================================================================
|    Register Name    |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|        y_reg        |   Latch    |   1   |  -  | -  | N  | N  | -  | -  | -  |
===============================================================================
```

Figure 2 - Latch inference report for non-full case statement

3.4 Synopsys full_case

Synthesis tools recognize two directives when added to the end of a Verilog **`case`** header. The directives are **`// synopsys full_case parallel_case`**. The directives can either be used together or an engineer can elect to use only one of the directives for a particular **case** statement. The Synopsys **parallel_case** directive is described in section 5.3.

When **// synopsys full_case** is added to a **case** statement header, there is no change in the Verilog simulation for the **case** statement, since "**// synopsys ...**" is interpreted to be nothing more than a Verilog comment; however, Synopsys DC parses all Verilog comments that start with "**// synopsys ...**" and interprets the **full_case** directive to mean that if a **case** statement is not full, then the outputs are "don't care's" for all unspecified **case** item combinations. If the **case** statement includes a **case default**, the **full_case** directive will be ignored because a **default** automatically causes the **case** statement to become full.

Example 3 shows a **case** statement for a 3-to-1 multiplexer that is not full but the **case** header includes a **full_case** directive. During Verilog simulation, when binary pattern **2'b11** is present on the select lines, the **y**-output will behave as if it were latched, the same as in Example 2, but the synthesis will treat the **y**-output as a "don't care" for the same select-line combination, causing a functional mismatch to occur between simulation and synthesis.

```
module mux3b (y, a, b, c, sel);
  (output reg       y,
   input      [1:0] sel,
   input            a, b, c);

  always @*
    case (sel) // synopsys full_case
      2'b00:   y = a;
      2'b01:   y = b;
      2'b10:   y = c;
    endcase
endmodule
```

Example 3 - Non-full case statement with **full_case** directive

```
Warning: You are using the full_case directive with a case statement in which not all cases
are covered.

Statistics for case statements in always block at line 7 in file
        '.../mux3b.v'
===============================================
|           Line           |  full/ parallel  |
===============================================
|            9             |    user/auto     |
===============================================
```

Figure 3 - Case statement report for a non-full case statement with **full_case** directive

3.5 A full_case statement with a case default

What happens to a **case** statement that includes both the **full_case** directive and a **case default**?

Simply stated, any **`case`** statement with a **`case default`** will disable the effect of a **`full_case`** directive. This would be an example of a **`full_case`** that does not work, the **`full_case`** directive is completely ignored, it is just extra code in the file and so it does not cause any design problems (it was just wasted typing and a potential point of confusion for some engineers who read the code and tried to determine what the directive would do).

3.6 Synopsys case statement reports - full_case

For each **`case`** statement that is read by Synopsys DC, a **`case`** statement report is generated that indicates one of the following conditions with respect to the full-nature of a each **`case`** statement:

- **`full`** / **`auto`** (Figure 4) - Synopsys Design Compiler (DC) tools have determined that the **`case`** statement as coded is **`full`**.

```
Statistics for case statements in always block at line ...
===============================================
|           Line           |  full/ parallel  |
===============================================
|            X             |    auto/auto     |
===============================================
```

Figure 4 - full / auto - Case statement is full

- **`full`** / **`no`** (Figure 5) - The **`case`** statement was not recognized to be full by Synopsys DC.

```
Statistics for case statements in always block at line ...
===============================================
|           Line           |  full/ parallel  |
===============================================
|            X             |     no/auto      |
===============================================
```

Figure 5 - full / no - Case statement not full

- **`full`** / **`user`** (Figure 6) - A Synopsys **`full_case`** directive was added to the **`case`** statement header by the user.

```
Statistics for case statements in always block at line ...
===============================================
|           Line           |  full/ parallel  |
===============================================
|            X             |    user/auto     |
===============================================
```

Figure 6 - full / user - "// synopsys full_case" added to the case header

4.0 SystemVerilog priority case

SystemVerilog adds the new **`case`** statement modifier called **`priority`**. This raises the question, aren't **`case`** statements already *priority* statements? The answer is yes, **`case`** statements are already *priority* statements.

So what does **`priority`** mean? When used in the context of a **`case`** statement, the **`priority`** modifier means that the **`case`** statement is a priority statement and that all possible legal cases have been listed. If during simulation the **`case`** expression ever becomes a value that cannot be matched to any of the **`case`** items, a runtime error shall be reported.

The biggest difference between a **`full_case`** directive and a **`priority`** modified **`case`** statement is that the **`priority`** keyword is part of the SystemVerilog syntax that will be interpreted the same by simulators, synthesis tools and formal verification tools. In essence, the **`priority case`** statement is a "safe" **`full_case case`** statement.

If the **`priority case`** statement includes a **`case default`** statement, then the effect of the **`priority`** keyword is disabled because the **`case`** statement is full through the use of the **`case default`** statement, and no runtime checking will ever find a **`case`** expression that cannot match one of the defined **`case`** items or **`default`** statement (this is the same as **`full_case`** with a **`case default`** - see Section 3.5),..

Just like the **`full_case`** directive, the **`priority case`** statement does not guarantee the removal of unwanted latches. Any **`case`** statement that makes assignments to more than one output in each **`case`** item statement can still generate latches if one or more output assignments are missing from other **`case`** item statements.

4.1 An editorial comment about the priority keyword

Editorial comment: I do not like the keyword **`priority`**.

As noted above, **`case`**-statements and **`if-else`**-statements are already priority statements and there will be confusion surrounding the **`priority`** keyword. What this modifier does is to allow the engineer to ***assert*** that all possible cases or if-conditions have been defined so a synthesis tool is free to optimize the **`case`**-logic or **`if`**-logic assuming that all other testable conditions are don't cares. Of course in synthesis, don't cares frequently lead to favorable logic optimization. Since an engineer has asserted that all testable conditions have been included, simulation tools are required to report an error if an unexpected condition is tested. The engineer asserted that all conditions were tested but the assertion failed!

A better keyword would have been either **`full`** or **`all_possible`**, but the former is a very commonly used identifier in legacy Verilog designs and the latter is somewhat verbose and clumsy. Since SystemVerilog design and verification engineers have already used the **`priority`** keyword in multiple designs and verification suites, we are now stuck with it. Remember,

priority case behaves just like **full_case**, except the simulator now knows about it and is required to react if the assertion is proven false during simulation.

5.0 What is a parallel case statement?

A *parallel* **case** statement is a **case** statement in which it is only possible to match any **case** expression to one and only one **case** item. If it is possible to find a **case** expression that would match more than one **case** item, the matching **case** items are called "overlapping" **case** items and the **case** statement is not **parallel**.

The examples in this section includes the **case** statement report that is generated when DC reads each Verilog example. For a description of the meaning of the **parallel_case** reports shown in these examples, go to Section 5.6.

5.1 Non-parallel case statements

Example 4 shows a **casez** statement that is not parallel because if the 3-bit **irq** bus has any one of the patterns **3'b011**, **3'b101**, **3'b110** or **3'b111**, more than one **case** item could potentially match the **irq** value. This will simulate like a priority encoder where **irq[2]** has **priority** over **irq[1]**, which has priority over **irq[0]**. This example will also infer a priority encoder when synthesized, as shown in Figure 8.

```
module intctl1a
  (output reg       int2, int1, int0,
   input      [2:0] irq             );

  always @* begin
    {int2, int1, int0} = 3'b0;
    casez (irq)
      3'b1??: int2 = 1'b1;
      3'b?1?: int1 = 1'b1;
      3'b??1: int0 = 1'b1;
    endcase
  end
endmodule
```

Example 4 - Non-parallel case statement

```
Statistics for case statements in always block at line 6 in file
        '.../intctl1a.v'
===============================================
|           Line           |  full/ parallel  |
===============================================
|            9             |      no/no       |
===============================================
```

Figure 7 - Case statement report for Example 4

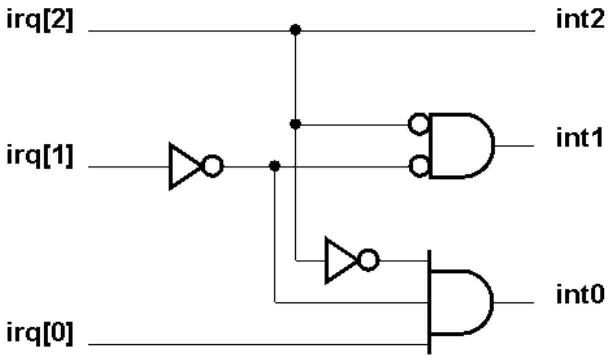

Figure 8 - Correctly implemented interrupt control logic

5.2 Parallel case statements

Example 5 is a modified version of Example 4 such that each of the **`case`** items is now unique and therefore **`parallel`**. Even though the **`case`** items are **`parallel`**, this example also happens to infer the same priority encoder logic of Figure 8 when synthesized.

```
module intctl2a
  (output reg       int2, int1, int0,
   input      [2:0] irq             );

  always @* begin
    {int2, int1, int0} = 3'b0;
    casez (irq)
      3'b1??: int2 = 1'b1;
      3'b01?: int1 = 1'b1;
      3'b001: int0 = 1'b1;
    endcase
  end
endmodule
```

Example 5 - Parallel case statement

```
Statistics for case statements in always block at line 6 in file
        '.../intctl2a.v'
===============================================
|           Line           |  full/ parallel  |
===============================================
|            9             |     no/auto      |
===============================================
```

Figure 9 - Case statement report for Example 5

5.3 Synopsys parallel_case

Example 6 is the same as Example 4 except that a **`// synopsys parallel_case`** directive has been added to the **`case`** header. This example will simulate like a priority encoder but will infer non-priority encoder logic when synthesized, as shown in Figure 11.

```
module intctl1b
  (output reg       int2, int1, int0,
   input      [2:0] irq             );

  always @* begin
    {int2, int1, int0} = 3'b0;
    casez (irq) // synopsys parallel_case
      3'b1??: int2 = 1'b1;
      3'b?1?: int1 = 1'b1;
      3'b??1: int0 = 1'b1;
    endcase
  end
endmodule
```

Example 6 - Non-parallel case statement with parallel_case directive

```
Warning: You are using the parallel_case directive with a case statement in
which some case-items may overlap

Statistics for case statements in always block at line 6 in file
        '.../intctl1b.v'
===============================================
|           Line           |  full/ parallel  |
===============================================
|            9             |     no/user      |
===============================================
```

Figure 10 - Case statement report for Example 6

In Example 6, the **`parallel_case`** directive has "worked" and now the synthesized logic does not match the simulated Verilog functional model.

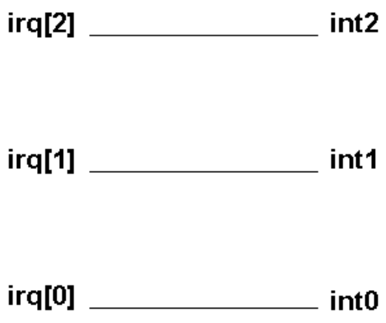

Figure 11 - Incorrectly implemented interrupt control logic

This is an example that demonstrates that adding the **`parallel_case`** directive makes the design smaller and faster, but in the process it also adversely changes the functionality of the design.

5.4 A parallel case statement with parallel_case directive

The **casez** statement in Example 7 is parallel! If a **parallel_case** directive is added to the **casez** statement, it will make no difference. The design will synthesize the same with or without the **parallel_case** directive.

The point is, the **parallel_case** directive is always most dangerous when it works! When it does not work, it is just extra characters at the end of the **case** header.

```
module intctl2b
  (output reg       int2, int1, int0,
   input      [2:0] irq             );

  always @* begin
    {int2, int1, int0} = 3'b0;
    casez (irq) // synopsys parallel_case
      3'b1??: int2 = 1'b1;
      3'b01?: int1 = 1'b1;
      3'b001: int0 = 1'b1;
    endcase
  end
endmodule
```

Example 7 - Parallel case statement with parallel_case directive

5.5 Verilog & VHDL case statements

VHDL **case** statements are required to have no overlap in any of the **case** items, and therefore are parallel and cannot infer priority encoders. VHDL **case** items are constants that are used to compare against the VHDL **case** expression. For this reason, it is also easy to parse multiple VHDL **case** item statements without the need to include **begin** and **end** keywords for **case** item statements.

Verilog **case** statements are permitted to have overlapping **case** items. Verilog **case** items can be separate and distinct Boolean expressions where one or more of the expressions can evaluate to "true" or "false." In those instances where more than one **case** item can match a "true" or "false" **case** expression, the first matching **case** item has priority over subsequent matching **case** items, therefore priority logic will be inferred by synthesis tools.

Verilog **casez** and **casex** statements can also include **case** items with constant vector expressions that include "don't-cares" that would permit a **case** expression to match multiple **case** items in the **casez** or **casex** statements, also inferring a **priority** encoder.

If all goes well, **full_case parallel_case** will do nothing to your design, and your design will work fine. The problem happens when **full_case parallel_case** DO work to change the functionality of your design or increase the size and area of your design.

There is one other style of Verilog **`case`** statement that frequently infers a priority encoder. The style is frequently referred to as the "**`case`** if true" or "reverse **`case`**" statement coding style.

This **`case`** statement style evaluates expressions for each **`case`** item and then tests to see of they are "true" (equal to **`1'b1`**). This coding style is used to infer very efficient one-hot finite state machines, but is otherwise a somewhat dangerous coding practice[1][3][4].

5.6 Synopsys case statement reports - parallel_case

For each **`case`** statement that is read by Synopsys DC, a **`case`** statement report is generated that indicates one of the following conditions with respect to the parallel nature of each **`case`** statement:

- **`parallel`** / **`no`** (Figure 12) - The **`case`** statement was not recognized to be parallel by Synopsys Design Compiler (DC) tools.

```
Statistics for case statements in always block at line ...
===============================================
|           Line           |  full/ parallel  |
===============================================
|            X             |     auto/no      |
===============================================
```

Figure 12 - parallel / no - Case statement not parallel

- **`parallel`** / **`auto`** (Figure 13) - Synopsys DC has determined that the **`case`** statement as coded is **`parallel`**.

```
Statistics for case statements in always block at line ...
===============================================
|           Line           |  full/ parallel  |
===============================================
|            X             |    auto/auto     |
===============================================
```

Figure 13 - parallel / auto - Case statement is parallel

- **`parallel`** / **`user`** (Figure 14) - A Synopsys **`parallel_case`** directive was added to the **`case`** statement header by the user.

```
Statistics for case statements in always block at line ...
===============================================
|           Line           |  full/ parallel  |
===============================================
|            X             |    auto/user     |
===============================================
```

Figure 14 - parallel / user - "// synopsys parallel_case" added to the case header

5.7 Coding priority encoders

Non-parallel **`case`** statements infer priority encoders; however, it is a poor coding practice to code priority encoders using **`case`** statements. It is better to code priority encoders using **`if-else-if`** statements.

Guideline: Code all intentional priority encoders using **`if-else-if`** statements. It is easier for the typical design engineer to recognize a priority encoder when it is coded as an **`if-else-if`** statement.

Guideline: Case statements can be used to create tabular coded parallel logic. Coding with **`case`** statements is recommended when a truth-table-like structure makes the Verilog code more concise and readable.

Although good priority encoders can be inferred from **`case`** statements, following the above coding guidelines will help to prevent mistakes and mismatches between pre-synthesis and post-synthesis simulations.

6.0 SystemVerilog unique case

SystemVerilog adds the new **`case`** statement modifier called "**`unique`**."

Any Verilog **`case`** statement is permitted to have overlapping **`case`** items, which means that a **`case`** expression could match more than one **`case`** item. When **`case`** items overlap, priority encoders are simulated and synthesized.

The **`unique`** keyword shall cause the simulator to report a run-time error if a **`case`** expression is ever found to match more than one of the **`case`** items; hence, the **`unique`** keyword is a both a **`case`** statement modifier and an assertion. The engineer ***asserts*** that the **`case`** items are unique and that it is impossible to match a **`case`** expression to more than one of the **`case`** items.

The **`unique`** keyword informs the synthesis tool that no priority encoders are needed to build the logic defined by the **`unique case`** statement. Eliminating unnecessary priority encoders typically guarantees smaller and faster logic.

The **`unique`** keyword also does the same runtime existence checking that is performed by the simulator. If a **`case`** expression does not match any of the **`unique case`** items, a runtime error is reported. As with the SystemVerilog **`priority case`** statement, adding a **`case default`** statement nullifies the testing for nonexistent matches between the **`case`** expression and the **`case`** items; however, adding the **`case default`** to a **`unique case`** statement does not remove the uniqueness testing.

The biggest difference between a **parallel_case** directive and a **unique case** statement modifier is that the **unique** keyword is part of the SystemVerilog syntax that will be interpreted the same by simulators, synthesis tools and formal verification tools. In essence, the **unique case** statement is a "safe" **parallel_case case** statement.

7.0 Mapping full_case & parallel_case to priority & unique

From the preceding sections, we can now determine which combinations of **priority** and **unique** will infer the exact same synthesis functionality as **full_case** and **parallel_case**. These equivalents are shown in Table 1.

full_case parallel_case version	**SystemVerilog version**

`case (...)`
`  ...`
`endcase`
	`case (...)`
	`  ...`
	`endcase`

`case (...) // full_case`
`  ...`
`endcase`
	`priority case (...)`
	`  ...`
	`endcase`

`case (...) // parallel_case`
`  ...`
`endcase`
	`unique case (...)`
	`  ...`
	`  default: ...`
	`endcase`

`case (...) // full_case parallel_case`
`  ...`
`endcase`
	`unique case (...)`
	`  ...`
	`endcase`

Table 1 - full, parallel, priority, unique case mapping

From Table 1, it is important to note that **unique case** *without* **case default** is equivalent to adding both of the older **full_case parallel_case** directives, while using **unique case** *with* **case default** is equivalent to only adding the older **parallel_case** directive.

Of course, the advantage to using the SystemVerilog **case** modifiers is that not only do the modifiers affect synthesis, but they are also recognized as design assertions and SystemVerilog simulators will report errors whenever the assertions are violated during simulation.

8.0 SystemVerilog priority & unique if

SystemVerilog adds the new **if** statement modifiers, **priority** and **unique** and only one of these keywords can be used before an **if-else-if** statement. A nested **if-else-if** statement could include another **priority** or **unique** modifier, but any of the **else**-branches of an **if-else-if** statement cannot be modified with **priority** or **unique**.

```
priority if (...) ...
else     if (...) ...
else     if (...) ...
```

Editorial comment: A useful enhancement to the SystemVerilog language would be a binary **`else`**, one that does not default-trigger when the **`if`**-tests fail due to **`X`**'s or **`Z`**'s.

From a purely coding style perspective, **`if-else-if`** statements suggest priority encoders, while **`case`** statements suggest parallel logic, even though a **`case`** statement can infer the exact same priority encoder.

So what does **`priority if`** mean? When used in the context of an **`if`** statement, the **`priority`** modifier means that the **`if-else-if`** must be true for at least one of the **`if`**-tests. If during simulation the **`if`** expression ever becomes a value that cannot be matched to any of the **`if-else-if`** tests, a runtime error shall be reported. Of course a final **`else`**-statement will disable any **`priority if`** testing.

So what does **`unique if`** mean? The **`unique`** keyword shall cause the simulator to report a run-time error if an **`if-else-if`** statement is ever found that matches more than one of the **`if-else-if`** tests during the current execution of the **`if-else-if`** statement. Just like the **`case`** statement, the **`unique if`** testing also does the equivalent **`priority if`** run-time testing.

9.0 Synthesis coding styles

Sunburst Design Assumption: it is generally a bad coding practice to give the synthesis tool different information about the functionality of a design than is given to the simulator.

Whenever either **`full_case`** or **`parallel_case`** directives are added to the Verilog source code, more information is potentially being given about the design to the synthesis tool than is being given to the simulator.

Guideline: In general, do not use **`full_case parallel_case`** directives with any Verilog **`case`** statements.

Guideline: Replace the older **`full_case parallel_case`** directives with new and safer SystemVerilog **`priority`** and **`unique`** directives.

Guideline: Educate (or fire) any employee or consultant who routinely adds **`full_case parallel_case`** to all **`case`** statements in their Verilog code, especially if the project involves the design of medical diagnostic equipment, medical implants, or detonation logic for thermonuclear devices!

Guideline: Do not use the **`priority`** and **`unique`** directives with all SystemVerilog **`case`** statements and **`if-else-if`** statements. There are still some efficient coding styles that do not define all possible testing conditions within conditional statement and these efficient coding styles would trigger run-time simulation errors if **`priority`** and **`unique`** directives were added to all **`case`** statements and **`if-else-if`** statements.

Other exceptions and guidelines will surely be discovered as engineers use the new SystemVerilog modifiers. These exceptions and guidelines will be added to later versions of this paper and will be freely downloadable from the www.sunburst-design.com/papers web site.

10.0 Latch example using full_case

Myth: **`// synopsys full_case`** removes all latches that would otherwise be inferred from a **`case`** statement.

Truth: The **`full_case`** directive and now the SystemVerilog **`priority`** and **`unique case`**-statement and **`if`**-statement modifiers only remove latches from a **`case`** statement for missing **`case`** items or incomplete **`if-else-if`** statements. One of the most common ways to infer a latch is to make assignments to multiple outputs from a single **`case`** statement but neglect to assign all outputs for each **`case`** item. Even adding the **`full_case`** directive or SystemVerilog **`priority`** or **`unique`** modifiers to this type of **`case`** statement will not eliminate latches[5].

Example 8 shows Verilog code for a simple address decoder that will infer a latch for the **`mce0_n`** **`mce1_n`** and **`rce_n`** outputs (as reported in Figure 15 and shown in Figure 16), despite the fact that the **`full_case`** directive was used with the **`case`** statement. In this example, the **`case`** statement is full but not all outputs are assigned for each **`case`** item; therefore, latches were inferred for all three outputs.

```
module addrDecode1a
  (output reg         mce0_n, mce1_n, rce_n,
   input      [31:30] addr                 );

  always @*
    casez (addr) // synopsys full_case
      2'b10: {mce1_n, mce0_n} = 2'b10;
      2'b11: {mce1_n, mce0_n} = 2'b01;
      2'b0?:          rce_n  = 1'b0;
    endcase
endmodule
```

Example 8 - **`full_case`** directive with latched outputs

```
Statistics for case statements in always block at line 6 in file
        ...'addrDecode1a.v'
===============================================
|           Line           |  full/ parallel  |
===============================================
|            8             |    user/auto     |
===============================================

Inferred memory devices in process
    in routine addrDecode1a line 6 in file
        ...'addrDecode1a.v'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|      mce0_n_reg     |   Latch   |   1   |  -  | -  | N  | N  | -  | -  | -  |
|      mce1_n_reg     |   Latch   |   1   |  -  | -  | N  | N  | -  | -  | -  |
|       rce_n_reg     |   Latch   |   1   |  -  | -  | N  | N  | -  | -  | -  |
===============================================================================
```

Figure 15 - Case statement report and latch report for **`full_case`** latched example

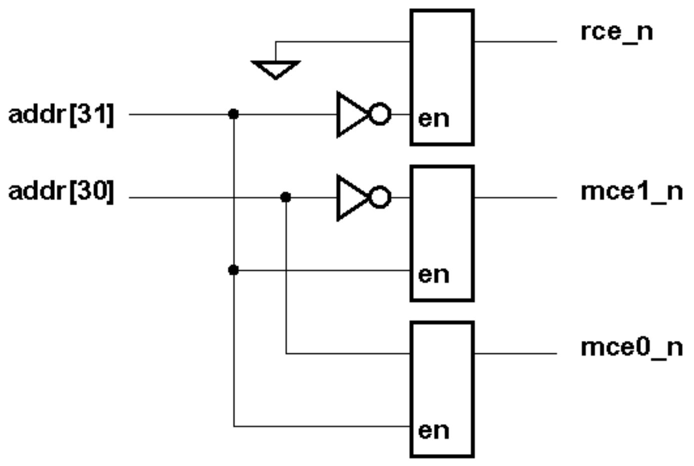

Figure 16 - Address decoder with unwanted latches

The easiest way to eliminate latches is to make initial default value assignments to all outputs immediately beneath the sensitivity list, before executing the **case** statement, as shown in Example 9. The correctly synthesized logic is shown in Figure 18.

```
module addrDecode1d (mce0_n, mce1_n, rce_n, addr);
  (output reg         mce0_n, mce1_n, rce_n,
   input      [31:30] addr                 );

  always @* begin
    {mce1_n, mce0_n, rce_n} = 3'b111;
    casez (addr)
      2'b10: {mce1_n, mce0_n} = 2'b10;
      2'b11: {mce1_n, mce0_n} = 2'b01;
      2'b0?:           rce_n  = 1'b0;
    endcase
  end
endmodule
```

Example 9 - Initial default value assignments to remove latches

```
Statistics for case statements in always block at line 6 in file
        ...'addrDecode1d.v'
===============================================
|           Line           |  full/ parallel  |
===============================================
|            9             |    auto/auto     |
===============================================
```

Figure 17 - Case statement report for Example 9

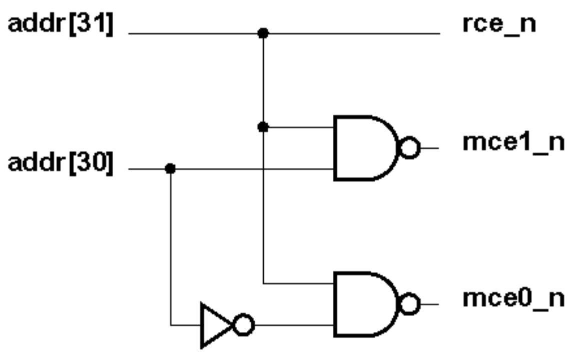

Figure 18 - Correctly implemented address decoder

11.0 Synopsys warnings

When Verilog files are read by DC, Synopsys issues warnings when the **`full_case`** directive is used with a **`case`** statement that was not full (see Synopsys full_case description in section 3.4). The same warnings are now issued when the new SystemVerilog **`priority`** and **`unique`** keywords are used in equivalent **`full_case`** coding styles with a **`case`** statement that is not full.

Example 10 shows a non-full **`case`** statement with **`full_case`** directive. Figure 19 shows the warning that is reported when the **`full_case`** directive is used with a non-full **`case`** statement.

```
module fcasewarn1b
  (output reg y,
   input      d, en);

  always @*
    case (en) // synopsys full_case
      1'b1: y = d;
    endcase
endmodule
```

Example 10 - Non-full case statement with full_case directive

```
"Warning: You are using the full_case directive with a case statement in which not all
cases are covered."

Statistics for case statements in always block
           at line 6 in file ..."/fcasewarn1b.v"
===============================================
|           Line           |  full/ parallel  |
===============================================
|            8             |    user/auto     |
===============================================
```

Figure 19 - Synopsys full_case warning

The warning in Figure 19 should really say, "watch out! the **full_case** directive might work and cause your design to break!!" Unfortunately this warning is easy to miss when running a synthesis script and the design might be adversely affected by the **full_case** directive.

Similarly, when Verilog files are read by DC, Synopsys issues warnings when the **parallel_case** directive (or new SystemVerilog **unique** case modifier) is used with a **case** statement that was not **parallel**. (see Synopsys parallel_case description in section 5.3).

Example 11 shows a non-parallel **case** statement with **parallel_case** directive. Figure 20 shows the warning that is reported when the **parallel_case** directive is used with a non-parallel **case** statement.

```
module pcasewarn1b
  (output reg y, z,
   input      a, b, c, d);

  always @* begin
    {y,z} = 2'b00;
    casez ({a,b,c,d}) // synopsys parallel_case
      4'b11??: y = 1'b1;
      4'b??11: z = 1'b1;
    endcase
  end
endmodule
```

Example 11 - Non-parallel case statement with parallel_case directive

```
"Warning: You are using the parallel_case directive with a case statement in which some
case-items may overlap."

Statistics for case statements in always block
           at line 6 in file ..."/pcasewarn1b.v"
===============================================
|           Line          |  full/ parallel  |
===============================================
|            9            |     no/user      |
===============================================
```

Figure 20 - Synopsys parallel_case warning

The warning in Figure 20 should really say, "watch out! the **parallel_case** directive might work and cause your design to break!!" Unfortunately this warning, like the **full_case** warning, is also easy to miss when running a synthesis script and the design might be adversely affected by the **parallel_case** directive.

12.0 Actual full_case design problem

The 2-to-4 decoder with enable in Example 12, uses a **case** statement that is coded without using any synthesis directives. The resultant design was a decoder built from 3-input and gates and inverters as shown in Figure 22. No latches were inferred because all outputs are given a default assignment before the **case** statement. For this example, the pre-synthesis and post-synthesis designs and simulations matched.

```
module code4a
  (output reg [3:0] y,
   input      [1:0] a,
   input            en);

  always @* begin
    y = 4'h0;
    case ({en,a})
      3'b1_00: y[a] = 1'b1;
      3'b1_01: y[a] = 1'b1;
      3'b1_10: y[a] = 1'b1;
      3'b1_11: y[a] = 1'b1;
    endcase
  end
endmodule
```

Example 12 - Decoder example with no **full_case** directive

```
Statistics for case statements in always block at line 9 in file
        '.../code4a.v'
===============================================
|           Line           |  full/ parallel  |
===============================================
|            12            |     no/auto      |
===============================================
```

Figure 21 - Case statement report for Example 12

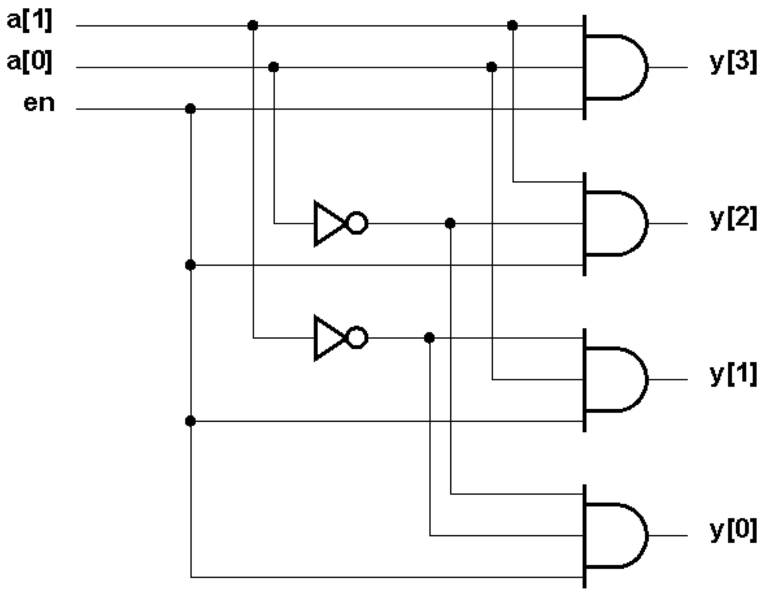

Figure 22 - Correctly synthesized 2-to-4 decoder with output enable

The 2-to-4 decoder with enable in Example 13, uses a **case** statement with the **full_case** synthesis directive. Because of this synthesis directive, the enable input (**en**) was optimized away during synthesis and left as a dangling input as shown in Figure 24. The pre-synthesis simulation results of modules **code4a** and **code4b** matched the post-synthesis simulation results of module **code4a**, but did not match the post-synthesis simulation results of module **code4b** [2].

```
// full_case example
// Decoder built from four 2-input nor gates
//   and two inverters
// The enable input is dangling (has been optimized away)
module code4b
  (output reg [3:0] y,
   input      [1:0] a,
   input            en);

  always @* begin
    y = 4'h0;
    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
```

Example 13 - Decoder example with **full_case** directive

```
Warning: You are using the full_case directive with a case statement in which
not all cases are covered

Statistics for case statements in always block at line 10 in file
        '.../code4b.v'
===============================================
|           Line           |  full/ parallel  |
===============================================
|            13            |    user/auto     |
===============================================
```

Figure 23 - Case statement report for Example 13

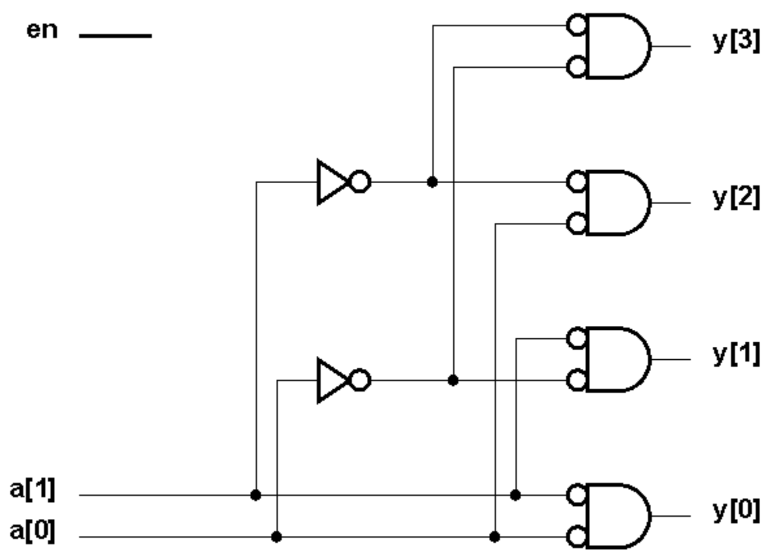

Figure 24 - Incorrectly synthesized 2-to-4 decoder with output enable

This same problem occurs when either the **`priority`** or **`unique`** modifiers are used with this **`case`** statement. The difference is that during simulation, a run-time error would be reported anytime the **`en`** input goes to **`0`**. This would be a hint that the **`priority`** and **`unique`** keywords are going to cause problems when this coding style is synthesized.

12.1 Should you always use priority and unique case modifiers?

Should SystemVerilog **`case`** and **`if-else-if`** statements always add the **`priority`** and **`unique`** modifiers? The answer is NO! It can be seen from Example 13 that **`priority`** or **`unique`** would break this efficient RTL coding style.

Remember that the **`priority`** modifier asserts that all possible cases have been defined within the **`case`** statement, and that if a **`case`** expression does not match any of the defined **`case`** items, then an error shall be reported. This causes problems with some commonly recommended and efficient RTL coding styles.

One of the easiest ways to avoid unwanted latches is to make a default assignment at the top of an **`always`** block to all of the outputs that are assigned within the **`always`** block as was shown in the 2-to-4 decoder code of Example 12.

The equivalent coding style using **`priority case`** is shown below. This coding style actually causes two problems: (1) the synthesis results will be the same faulty design shown in Figure 24 due to the **`full_case`** directive that was added to the Example 13 code, and (2) the pre-synthesis simulation will cause errors whenever the enable is **`0`**. because the **`en=0`** case was covered by the default assignment at the top of the **`always`** block and not defined within the **`priority case`** statement.

```
// priority case example
// Decoder built from four 2-input nor gates
//   and two inverters
// The enable input is dangling (has been optimized away)
module code4c
  (output reg [3:0] y,
   input      [1:0] a,
   input            en);

  always_comb begin
    y = '0;
    priority 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
```

Figure 25 - Decoder example with **`priority case`** modifier

The problem in this example could be solved by adding a **`case default`** assignment to set all of the **`y`**-bits to **`0`**, but as discussed earlier, adding a **`case default`** nullifies the effects of the **`priority`** modifier, so there is no point in adding the extraneous **`priority`** keyword if you are also going to cancel the modifier with a **`case default`**.

Similarly, if you are going to design a priority encoder using **`if-else`** statements, almost all priority encoder designs depend on overlapping **`if-else-if`** tests, and most RTL-coded priority encoders end with an **`else`** statement, again nullifying the **`priority`** modifier. If the **`priority`** modifier is replaced with a **`unique`** modifier, simulation with overlapping **`case`** items would cause run-time errors.

At the time this paper was written, the author was of the opinion that most **`case`** statements can be reasonably and safely coded using the **`unique`** modifier and adding the **`case default`**. This is the simulation equivalent of a **`parallel_case`** assertion but not a **`full_case`** assertion. This opinion may change over time as more designs and experiments are conducted and the author welcomes feedback from other users on their experiences using the **`unique`** and **`priority case`**-statement modifiers.

13.0 Actual parallel_case design problem

One consultant shared the experience where **`parallel_case`** was added to the Verilog code for a large ASIC design to remove stray priority encoders and infer a smaller and faster design. The Verilog **`case`** statement was coded as a priority encoder and all RTL simulations worked correctly. Unfortunately, the gate-level design without priority encoder did not function correctly and the gate-level simulations did not catch the problem. This ASIC had to be re-designed, costing $100,000's of actual dollars, delayed product release, and unknown lost dollars for being months late to market.

14.0 Summary of guidelines and conclusions

Guideline: Code all intentional priority encoders using `if-else-if` statements. It is easier for the typical design engineer to recognize a priority encoder when it is coded as an `if-else-if` statement.

Guideline: Coding with `case` statements is recommended when a truth-table-like structure makes the Verilog code more concise and readable.

Guideline: In general, do not use `full_case parallel_case` directives with any Verilog `case` statements.

Guideline: Replace the older `full_case parallel_case` directives with new and safer SystemVerilog `priority` and `unique` directives.

Guideline: Educate (or fire) any employee or consultant who routinely adds `full_case parallel_case` to all `case` statements in their Verilog code.

Guideline: Do not use the `priority` and `unique` directives with all SystemVerilog `case` statements and `if-else-if` statements.

Other exceptions and guidelines will surely be discovered as engineers use the new SystemVerilog modifiers. These exceptions and guidelines will be added to later versions of this paper and will be freely downloadable from the www.sunburst-design.com/papers web site. Please email the author if you believe you have other additional useful coding guidelines.

Conclusion: `full_case` and `parallel_case` directives are most dangerous when they work!

Conclusion: the SystemVerilog `priority` and `unique` case statement modifiers are safe replacements for the `full_case parallel_case` evil twins, but an engineer still needs to know how they work and when their usage is most advantageous. The SystemVerilog `priority` and `unique` keywords should not be indiscriminately used with all `case` statements.

The SystemVerilog `priority` and `unique` keywords are not only valuable `case` statement modifiers, they are also valuable simulation, synthesis and formal verification ***assertions***.

It is exciting to see so many SystemVerilog features already implemented by both VCS and Design Compiler. VCS now recognizes and reports run-time errors when the `priority` and `unique` design assertions are violated, and DC recognizes the new `priority` and `unique` modifiers, allowing for safe `full_case parallel_case` equivalent synthesis optimization.

15.0 Case statement definitions

To fully understand how the "**full_case parallel_case**" directives work, a common set of terms is needed to describe the different parts of a **case** statement. This section defines a common set of terms that are used to describe **case** statement functionality throughout the rest of the paper.

15.1 Case statement

In Verilog, a **case** statement includes all of the code between the Verilog keywords, **case** (**casez**, **casex**) and **endcase**.

A **case** statement is a select-one-of-many construct that is roughly equivalent to an **if-else-if** statement. The general **case** statement in Figure 26 is equivalent to the general **if-else-if** statement shown in Figure 27.

```
case (case_expression)
  case_item1 : case_item_statement1;
  case_item2 : case_item_statement2;
  case_item3 : case_item_statement3;
  case_item4 : case_item_statement4;
  default    : case_item_statement5;
endcase
```

Figure 26 - Case Statement - General Form

```
if      (case_expression === case_item1) case_item_statement1;
else if (case_expression === case_item2) case_item_statement2;
else if (case_expression === case_item3) case_item_statement3;
else if (case_expression === case_item4) case_item_statement4;
else                                     case_item_statement5;
```

Figure 27 - If-else-if Statement - General Form

As can be seen from the coding syntax shown in Figure 26 and Figure 27, when testing against a common expression, **case** statements offer a very nice and concise shorthand for describing the same problem.

15.2 Case statement header

A **case** statement header consists of the **case** (**casez**, **casex**) keyword followed by the **case** expression, usually all on one line of code.

Typically, when adding **full_case** or **parallel_case** directives to a **case** statement, the directives are added as a comment immediately following the **case** expression at the end of the **case** statement header and before any of the **case** items on subsequent lines of code.

```
// Synopsys comment
case (case_expression) // synopsys full_case parallel_case
  ...
```

```
// Synplicity comment
case (case_expression) // synthesis full_case parallel_case
  ...
```

The IEEE Verilog-2002 RTL Synthesis Standard[8] modified the way these directives should be added to a **`case`** statement. Instead of using comments to express a synthesis directive, the directives were added to attributes prior to the **`case`** statement.

```
(* synthesis, full_case, parallel_case *) case (case_expression) ...
```

The attribute method for adding these directives meant that tools did not have to parse Verilog comments to find synthesis directives. A properly coded design would allow tools to treat all comments as comments and not as potential commands.

Even though an IEEE Verilog-2002 RTL Synthesis-compliant tool now reads synthesis directives within attributes instead of directives within comments, the evil and flawed nature of the **`full_case`** **`parallel_case`** "*evil twins*" still exist. The attribute method for adding directives did not correct the fundamental problems related to these directives.

15.3 Case expression

A Verilog **`case`** expression is the expression enclosed between parentheses immediately following the **`case`** keyword. In Verilog, a **`case`** expression can either be a constant, such as **`1'b1`** (one bit of **`1`**, or "true"), it can be an expression that evaluates to a constant value, or most often it is a bit or vector of bits that are used to compare against **`case`** items.

15.4 Case item

The **`case`** item is the bit, vector or Verilog expression that is used to compare against the **`case`** expression.

Unlike other high-level programming languages such as 'C', the Verilog **`case`** statement includes an implied break statement. The first **`case`** item that matches the current **`case`** expression causes the corresponding **`case`** item statement to be executed and then all of the rest of the **`case`** items are skipped (ignored) for the current pass through the **`case`** statement.

15.5 Case item statement

A **`case`** item statement is one or more Verilog statements that are executed if the corresponding **`case`** item matches the current **`case`** expression.

Unlike VHDL, Verilog **`case`** items can themselves be expressions. To simplify parsing of Verilog source code, Verilog **`case`** item statements must be enclosed between the keywords **`begin`** and **`end`** if more than one statement is to be executed for a selected **`case`** item. This is one of the few places were Verilog syntax requirements are considered by VHDL-literate engineers to be too verbose.

15.6 Case default

An optional **`case default`** can be included in the **`case`** statement to indicate what actions to perform if none of the defined **`case`** items matches the current **`case`** expression. It is good coding style to place the **`case default`** last, even though the Verilog standard does not require it.

15.7 Casez

In Verilog there is a **`casez`** statement, a variation of the **`case`** statement that permits **`z`** and **`?`** values to be treated during **`case`**-comparison as "don't care" values. **`z`** and **`?`** are treated as a don't care if they are in the **`case`** expression *and/or* if they are in the **`case`** item.

More information on the precautions that should be taken when using **`casez`** for RTL modeling and synthesis are detailed in a paper authored by myself and Don Mills in 1999[5].

Guideline: Exercise caution when coding synthesizable models using the Verilog **`casez`** statement.

Coding Style Guideline: When coding a **`case`** statement with "don't cares," use a **`casez`** statement and use **`?`** characters instead of **`z`** characters in the **`case`** items to indicate "don't care" bits.

15.8 Casex

In Verilog there is a **`casex`** statement, a variation of the **`case`** statement that permits **`z`**, **`?`** and **`x`** values to be treated during comparison as "don't care" values. **`x`**, **`z`** and **`?`** are treated as a don't care if they are in the **`case`** expression *and/or* if they are in the **`case`** item.

More information on the dangers of using **`casex`** for RTL modeling and synthesis are detailed in a paper authored by myself and Don Mills in 1999[5]. In our 1999 paper, Don and I gave a guideline to not use **`casex`** in RTL models. To date, I have not found a good safe usage for the **`casex`** statement in either RTL, behavioral or testbench coding so my current guideline is:

Guideline: Do not use the **`casex`** statement.

References

[1] Clifford E. Cummings, "Coding And Scripting Techniques For FSM Designs With Synthesis-Optimized, Glitch-Free Outputs," SNUG (Synopsys Users Group Boston, MA 2000) Proceedings, 2000. Also available at www.sunburst-design.com/papers

[2] 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

[3] Clifford E. Cummings, "State Machine Coding Styles for Synthesis," SNUG'98 (Synopsys Users Group San Jose, CA, 1998) Proceedings, 1998. Also available at www.sunburst-design.com/papers

[4] Clifford E. Cummings, "Synthesizable Finite State Machine Design Techniques Using the New SystemVerilog 3.0 Enhancements," SNUG (Synopsys Users Group San Jose, CA 2003) Proceedings, 2003. Also available at www.sunburst-design.com/papers

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

[6] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, IEEE Computer Society, IEEE, New York, NY, IEEE Std 1364-1995

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

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

[9] IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, IEEE Computer Society, New York, NY, IEEE Std 1076.6-1999

[10] SystemVerilog 3.0 Accellera's Extensions to Verilog, Accellera, 2002. Available online at www.eda.org/sv

[11] SystemVerilog 3.1 Accellera's Extensions to Verilog, Accellera, 2003. Available online at www.eda.org/sv

[12] SystemVerilog 3.1a Language Reference Manual, Accellera's Extensions to Verilog, Accellera, 2004. Available online at www.eda.org/sv

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 23 years of ASIC, FPGA and system design experience and 13 years of Verilog, synthesis and methodology training experience.

Mr. Cummings, a member of the IEEE 1364 Verilog Standards Group (VSG) since 1994, is the only Verilog and SystemVerilog trainer to co-develop and co-author the IEEE 1364-1995 & IEEE 1364-2001 Verilog Standards, the IEEE 1364.1-2002 Verilog RTL Synthesis Standard and the Accellera SystemVerilog 3.0, 3.1 & 3.1a Standards. Mr. Cummings is also an active member of the IEEE P1364-2005 Verilog and IEEE P1800-2005 SystemVerilog Standards Groups.

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

Sunburst Design, Inc. offers Verilog, Verilog Synthesis and SystemVerilog training courses. For more information, visit the www.sunburst-design.com web site.

Email address: cliffc@sunburst-design.com

An updated version of this paper can be downloaded from the web site:
www.sunburst-design.com/papers

(Data accurate as of January 24th, 2005)