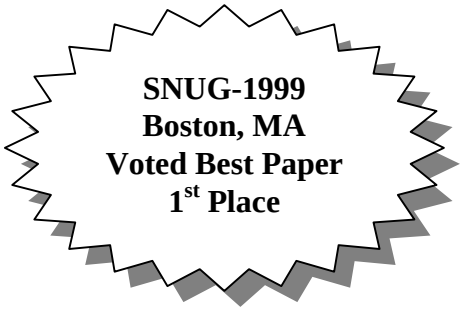
"full_case parallel_case", the Evil Twins of Verilog Synthesis

Clifford E. Cummings

Sunburst Design, Inc.



ABSTRACT

Two of the most over used and abused directives included in Verilog models are the directives "//synopsys full_case parallel_case". The popular myth that exists surrounding "full_case parallel_case" is that these Verilog directives always make designs smaller, faster and latch-free. This is false! Indeed, the "full_case parallel_case" switches frequently make designs larger and slower and can obscure the fact that latches have been inferred . These switches can also change the functionality of a design causing a mismatch between pre-synthesis and post-synthesis simulation, which if not discovered during gate-level simulations will cause an ASIC to be taped out with design problems.

This paper details the effects of the "full_case parallel_case" directives and includes examples of flawed and inefficient logic that is inferred using these switches. This paper also gives guidelines on the correct usage of these directives.

1.0 Introduction

The "full_case parallel_case" commands are two of the most abused synthesis directives employed by Verilog synthesis design engineers. The reasons cited most often to the author for using "full_case parallel_case" are:

- "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 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 ***these directives are always most dangerous when they work!***

This paper will define "full" and "parallel" case statements, detail case statement usage and show the effects that the "full_case parallel_case" directives have on synthesized code. An alternate title for this paper could be: "How to add $200,000 to the cost and 3-6 months to the schedule of your ASIC design without trying!"

2.0 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 will be used to describe case statement functionality throughout the rest of the paper.

2.1 Case statement

In Verilog, a case statement includes all of the code between the Verilog keywords, "case" ("casez", "casex") and "endcase" [1].

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 1 is equivalent to the general if-else-if statement shown in Figure 2.

```
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 1 - 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 2 - If-else-if Statement - General Form

2.2 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.

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.

2.3 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.

2.4 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 implied break statements. 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.

2.5 Case item statement

A case item statement is one or more Verilog statements that are executed if the 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.

2.6 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.

2.7 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 by Mills [2].

Guideline: Exercise caution when coding synthesizable models using the Verilog casez statement [2].

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.

2.8 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 by Mills [2]

Guideline: Do not use casex for synthesizable code [2].

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."

3.1 Synopsys case statement reports - "full_case"

For each case statement that is read by Synopsys tools, 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 3) - Synopsys 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 3 - full / auto - Case statement is "full"

- Full / no (Figure 4) - The case statement was not recognized to be "full" by Synopsys.

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

Figure 4 - full / no - Case statement not "full"

- Full / user (Figure 5) - 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 5 - full / user - "// synopsys full_case" added to the case header

3.2 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 Draft Standard For VHDL RTL Synthesis [3] and the IEEE Draft Standard For Verilog RTL Synthesis [4].

3.3 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 driven onto the select lines, the y-output will be driven to an unknown, but the synthesis will treat the y-output as a "don't care" for the same select-line combination, 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.

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

  always @(a or b or c or sel)
      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 6 - Case statement report for a case statement with a case default

3.4 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 driven onto the select 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 7.

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

  always @(a or b or c or sel)
      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 7 - Latch inference report for non-full case statement

3.5 Synopsys "full_case"

Synopsys 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 4.4.

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 parses all Verilog comments that start with "// synopsys ..." and interprets the "full_case" directive to mean that if a case statement is not "full" that the outputs are "don't care's" for all unspecified case items. If the case statement includes a case default, the "full_case" directive will be ignored.

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 driven onto 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       y;
  input  [1:0] sel;
  input        a, b, c;
  reg          y;

  always @(a or b or c or sel)
      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 8 - Case statement report for a non-full case statement with "full_case" directive

4.0 What is a "parallel" case statement?

A "parallel" case statement is a case statement in which it is only possible to match a 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."

4.1 Synopsys case statement reports - "parallel_case"

For each case statement that is read by Synopsys tools, 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 9) - The case statement was not recognized to be "parallel" by Synopsys.

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

Figure 9 - parallel / no - Case statement not "parallel"

- Parallel / auto (Figure 10) - Synopsys tools have 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 10 - parallel / auto - Case statement is "parallel"

- Parallel / user (Figure 11) - 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 11 - parallel / user - "// synopsys parallel_case" added to the case header

4.2 Non-parallel case statements

Example 4 shows a casez statement that is not parallel because if the 3-bit irq bus is 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.

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

  always @(irq) 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 12 - Case statement report for Example 4

4.3 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 happens to infer priority encoder logic when synthesized.

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

  always @(irq) 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 13 - Case statement report for Example 5

4.4 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.

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

  always @(irq) 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 14 - Case statement report for Example 6

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

4.5 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 as 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 (int2, int1, int0, irq);
  output       int2, int1, int0;
  input  [2:0] irq;
  reg          int2, int1, int0;

  always @(irq) 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

4.6 Verilog & VHDL case statements

VHDL case statements are required to have no overlap in of any case items, are therefore "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, the corresponding 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 it 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.

4.7 Coding priority encoders

Non-parallel case statements infer priority encoders. 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 a 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.

Guideline: Examine all synthesis tool case-statement reports [5].

Guideline: Change the case statement code, as outlined in the above coding guidelines, whenever the synthesis tool reports that the case statement is not parallel (whenever the synthesis tool reports "no" for "parallel_case") [5].

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.

5.0 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: There are exceptions to the above guideline but you better know what you're doing if you plan to add "full_case parallel_case" directives to your Verilog code.

Guideline: Educate (or fire) any employee or consultant that 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: only use full_case parallel_case to optimize onehot FSM designs.

Other exceptions might exist and will be acknowledged by the author as they are discovered.

6.0 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 only removes latches from a case statement for missing case items. 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 to this type of case statement will not eliminate latches [6].

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, 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. 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.

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

  always @(addr)
    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

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

  always @(addr) 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 16 - Case statement report for Example 9

7.0 Synopsys warnings

When Verilog files are read by design_analyzer or dc_shell, 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 3.5).

Example 10 shows a non-full case statement with "full_case" directive. Figure 17 shows the warning that is reported when the "full_case" directive is used with a non-full case statement.

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

  always @(d or en)
    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 17 - Synopsys "full_case" warning

The warning in Figure 17 is really saying, "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 design_analyzer or dc_shell, Synopsys issues warnings when the "parallel_case" directive is used with a case statement that was not "parallel." (see Synopsys "parallel_case" description in section 4.4).

Example 11 shows a non-parallel case statement with "parallel_case" directive. Figure 18 shows the warning that is reported when the "parallel_case" directive is used with a non-parallel case statement.

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

  always @(a or b or c or d) begin
    {y,z} = 2'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 18 - Synopsys "parallel_case" warning

The warning in Figure 18 is really saying, "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.

8.0 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. No latch is 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. 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. 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].

```
// no full_case
// Decoder built from four 3-input and gates
//   and two inverters
module code4a (y, a, en);
  output [3:0] y;
  input  [1:0] a;
  input        en;
  reg    [3:0] y;

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

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 19 - Case statement report for Example 12

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

  always @(a or en) begin
    y = 4'h0;
    case ({en,a}) // synopsys full_case
      3'b1_00: y[a] = 1'b1;
      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 20 - Case statement report for Example 13

9.0 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.

10.0 Summary of guidelines and conclusions

Summary of guidelines and conclusions

Guideline: Exercise caution when coding synthesizable models using the Verilog casez statement [2].

Guideline: Do not use casex for synthesizable code [2].

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

Guideline: There are exceptions to the above guideline but you better know what you're doing if you plan to add "full_case parallel_case" directives to your Verilog code.

Guideline: Code all intentional priority encoders using if-else-if statements. It is easier for a 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: Examine all synthesis tool case-statement reports [5].

Guideline: Change the case statement code, as outlined in the above coding guidelines, whenever the synthesis tool reports that the case statement is not parallel (whenever the synthesis tool reports "no" for "parallel_case") [5].

Guideline: only use full_case parallel_case to optimize onehot FSM designs.

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.

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

Conclusion: "full_case" and "parallel_case" directives are most dangerous when they work! It is better to code a full and parallel case statement than it is to use directives to make up for poor coding practices.

References

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

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

[3] IEEE P1076.6 Draft Standard For VHDL Register Transfer Level Synthesis, section 5

[4] IEEE P1364.1 Draft Standard For Verilog Register Transfer Level Synthesis, section 4

[5] Steve Golson, personal communication

[6] "HDL Compiler for Verilog Reference Manual," section 9. Synopsys Online Documentation v1999.05

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 18 years of ASIC, FPGA and system design experience and eight years of Verilog, synthesis and methodology training experience.

Mr. Cummings, a member of the IEEE 1364 Verilog Standards Group (VSG) since 1994, currently chairs the VSG Behavioral Task Force, which is charged with proposing enhancements to the Verilog language. Mr. Cummings is also a member of the IEEE Verilog Synthesis Interoperability Working Group.

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

E-mail Address: cliffc@sunburst-design.com
This paper can be downloaded from the web site: www.sunburst-design.com/papers

(Data accurate as of October 9th, 2000)