

SystemVerilog Saves the Day—the Evil Twins are Defeated!

“unique” and “priority” are the new Heroes

Stuart Sutherland
Sutherland HDL, Inc.
stuart@sutherland-hdl.com

ABSTRACT

Abstract: The villains of synthesis for many a design are the “parallel_case” and “full_case” synthesis pragmas. The dastardly deeds of these infamous pragmas have been well documented in a past SNUG paper, “‘full_case parallel_case’, the Evil Twins of Verilog Synthesis”, by Clifford Cummings[6]. Despite this excellent paper, these pragmas are still misunderstood and are abused by many design engineers. SystemVerilog introduces two new synthesis heroes, “**unique**” and “**priority**”. These new keywords are intended to replace the villainous twin pragmas. This paper discusses in depth the purpose of the **unique** and **priority** decision modifiers, and how they affect simulation, synthesis, and formal verification. Comparisons between the SystemVerilog keywords and the infamous pragmas are made. These comparisons show how SystemVerilog’s **unique** and **priority** keywords provide all the advantages of the full_case/parallel_case synthesis pragmas, and eliminate the disadvantages that made these pragmas villains. Guidelines on the proper usage of the **unique** and **priority** keywords are presented, showing how these decision modifiers should be properly used.

1.0 Introduction

This focus of this paper on the proper modeling, simulation and synthesis of multiple-branch decisions. The goal of this paper is two-fold: First, to ensure that the design engineer’s assumptions about each and every multiple-branch decision in a design is correct. Second, to ensure that synthesis compilers will correctly implement what the design engineer specified.

The concepts presented in this paper are based on the following key principles:

- Software simulation works differently than hardware.
- The Synopsys DC synthesis tool generally tries to match simulation behavior.
- Engineers occasionally need to be smarter than synthesis.

1.1 Software works differently than hardware

Software and hardware do not work the same way. Software programs execute sequentially, as a serial stream of instructions, whereas hardware is very parallel in its execution. These differences are readily apparent in how a multiple-branch decision is executed, as shown in the following two examples.

Example 1 illustrates one way in which the difference between software and hardware is manifest:

```
module mux3to1a (output reg y,  
                input      a, b, c,  
                input [1:0] select );  
  
  always @* begin  
    if      (select == 2'b00) y = a;  
    else if (select == 2'b01) y = b;  
    else if (select == 2'b10) y = c;  
  end  
endmodule
```

Example 1 — 3-to-1 MUX using multiple if...else decisions (Verilog language, keywords in bold)

In software, the `select` values will be tested one at a time, in the order in which the values are listed in the code. As soon as the first match is found, no other values are tested. If the value of `select` does not match any of the branches (a value of `2'b11`, in this example), then no action is taken, and the output of the MUX remains unchanged.

Hardware, unless specifically designed otherwise, will evaluate this example very differently. In hardware, all four values of `select` will be evaluated in parallel. The decode of these four possible values will cause one branch to be selected. If the value of `select` is `2'b11`, the output of the MUX will be set to some value, but that value cannot be determined without examining the actual transistors that make up the decode logic. In order to force hardware to match software behavior, additional hardware is required, probably in the form of latch circuitry, so that the value of the output is not changed when the `select` input has a value of `2'b11`.

Example 2 illustrates another difference in how software and hardware will evaluate a multiple-branch decision. This example illustrates an interrupt decoder.

```
module interrupt_decode1 (output reg something,  
                        input [3:0] IRQ );  
  
  always @* begin  
    if      (IRQ[0])      // test if IRQ bit 0 is set  
      // process interrupt 0  
    else if (IRQ[1])      // test if IRQ bit 1 is set  
      // process interrupt 1  
    else if (IRQ[2])      // test if IRQ bit 2 is set  
      // process interrupt 2  
    else if (IRQ[3])      // test if IRQ bit 3 is set  
      // process interrupt 3  
    else  
      // process default for no interrupt  
  end  
endmodule
```

Example 2 — Interrupt decoder with 4 possible interrupts

In software, the `IRQ` interrupt bits will be tested one at a time, in the order in which the bits are listed in the code. As soon as one of the bits tests as true, no other bits are tested. If no bits are set, then no action is taken. Hardware, unless specifically designed otherwise, will evaluate this

example very differently. If multiple `IRQ` bits are set, multiple actions will be executed in parallel. To make hardware evaluate this multiple-branch decision in the same way as software, priority encoding logic would need to be added to the design.

1.2 The Synopsys DC synthesis tool generally tries to match simulation behavior

A primary role of The Synopsys DC synthesis tool is to transform abstract programming statements into a hardware implementation. DC is reasonably intelligent, and will try to generate an implementation that approximates the software simulation behavior as closely as possible. In Example 1, above, DC will add latched logic to the 3-to-1 MUX circuitry, to maintain the behavior that the output will remain unchanged for a `select` value of `2'b11`. For Example 2, DC will add priority encoded logic so that the `IRQ` bits are evaluated in the same order as they are in simulation, and so that only the first `IRQ` bit that is set will cause an action to be taken.

1.3 Engineers occasionally need to be smarter than synthesis

Sometimes it is necessary for engineers to override the default behavior of the synthesis tool. For example, it may be that, in the 3-to-1 MUX example, it is impossible for a `select` value of `2'b11` to occur. Therefore, the extra logic to latch the MUX output is not necessary, and can be removed from the hardware circuitry. In the interrupt decoder example, an engineer might know that there will never be two `IRQ` bits set at the same time, and, therefore, it is not necessary for the hardware to evaluate the `IRQ` bits in the same order as the software simulation model.

DC provides a way for an engineer to override how synthesis will interpret a decision statement. However, if an engineer is mistaken about the design assumptions, overriding the default synthesis behavior will very likely result in circuitry that does not work correctly.

2.0 Verilog decision statements syntax and semantics

[If you already know how the Verilog `if...else` and `case` statements work, then you can skip this section, and jump to section 3]

In order to ensure that decision statements correctly model intended functionality, and will synthesize to the correct hardware implementation, it is important to understand the simulation semantics of decision statements.

The Verilog standard provides four decision statements: `if...else`, `case`, `casez` and `casex`. The syntax and simulation semantics of each of these are discussed in the following subsections.

All Verilog decision statements are procedural (programming) statements, and must be specified in a procedural block context. Verilog procedural blocks are `initial` blocks, `always` blocks, `task` and `function`. SystemVerilog adds a `final` procedural block.

2.1 Verilog `if...else` decisions

The syntax of the Verilog `if...else` statement is straightforward, and similar to the C language:

```
if ( expression ) statement_or_statement_group
else statement_or_statement_group
```

For example:

```
always @(posedge clock)
  if (!reset) begin
    r1 <= 0;
    r2 <= 0;
  end
  else begin
    r1 <= opcode;
    r2 <= operand;
  end
```

Example 3 — Verilog if...else statement

The expression can be anything in Verilog that has a logic value, including nets, variables, constants and literal values. The expression can also be the result of an operation, such as a concatenation or the “not” (!) operator. If the value of the expression evaluates to true, then the statement or group of statements following the expression are executed. To evaluate as true, at least one bit of the expression must be a logic 1, and none of the bits a logic X or Z.

If the expression evaluates to either false (all bits set to 0) or unknown (any bit set to X or Z), then the statement or statement group following the **else** keyword is executed.

The **else** branch of an **if...else** decision is optional. If no **else** branch is specified, and the expression is not true, then no statements are executed. This behavior has an important implication for hardware implementation, which is discussed in Section 3 of this paper.

Multiple-branch decisions can be formed using a series of **if...else** decisions, as was illustrated earlier, in Examples 1 and 2. Verilog simulation semantics define that a series of **if...else** decisions are evaluated sequentially. When an expression in the sequence evaluates as true, that branch is executed, and no further expressions are evaluated. This sequential behavior creates a priority to the order in which the expressions are listed, with the first branch having the highest priority, and the last branch the lowest priority.

2.2 Verilog case statements

A **case** statement provides a more concise way to specify multiple-branch decisions than using a series of **if...else** statements. The basic syntax of a Verilog **case** statement is:

```
case ( case_expression )
  case_item1 : statement_or_statement_group;
  case_item2 : statement_or_statement_group;
  case_item3 : statement_or_statement_group;
  default   : statement_or_statement_group;
endcase
```

An example of using a **case** statement is:

```
always @* begin
  case (state)
    4'h1:   begin // test if in state 1
              // do state 1 activity
            end
  endcase
end
```

```

        end
4'h2:   begin    // test if in state 2
        // do state 2 activity
        end
4'h3:   begin    // test if in state 3
        // do state 3 activity
        end
default: begin    // any other state value
        // do other states activity
        end
    endcase
end

```

Example 4 — FSM state decoder

The **case** keyword is followed by an expression, which is referred to as the “*case expression*”. This expression can be anything that has a logic value, including a net, variable, constant, or literal value. The case expression can also be the result of an operation, such as a concatenation.

The case expression is followed by any number of “*case items*”. Each case item is also an expression, which can be anything that has a logic value, including the result of an operation.

The value of the case expression is compared to each case item. If the value of the case expression and the value of the case item match, then that branch is executed. The comparison is made bit-by-bit, for all four possible Verilog values, 0, 1, X and Z.

A case statement can also include a *default branch*, specified using the **default** keyword. The default branch is executed if no case items matched the case expression. The default branch is analogous to the final else branch in an **if...else** series of decisions.

Verilog semantic rules define that the case items must be evaluated in the order in which the items are listed. Verilog also defines that at most only one case item branch is executed. If no case items match the case expression, and if no default branch has been specified, then no branch is executed. Any variables set within the case statement retain their previous values.

2.3 Verilog casez and casex statements

Verilog provides two variations of the **case** statement, **casez** and **casex**. These variations allow “*don’t care*” bits to be specified in either the case expression or the case items. Synthesis only supports don’t care bits in case items. When a bit is flagged as “don’t care”, then the value of that bit is ignored when comparing the case expression to the case item.

With a **casez** statement, any case item bits that are specified with the characters *z*, *Z* or *?* are treated as don’t care bits. With a **casex** statement, any case item bits that are specified with the characters *x*, *X*, *z*, *Z* or *?* are treated as don’t care bits.

Example 2, listed earlier in this paper, showed an interrupt decoder example modeled using **if...else** decisions. This same functionality can be coded using a **casez** (or **casex**) statement, as follows:

```

module interrupt_decode2 (output reg something,
                        input [3:0] IRQ );

```

```

always @* begin
  // set outputs to defaults for when there is no interrupt
  casez (IRQ) // use don't care bits when evaluating IRQ
    4'b???1: begin // test if IRQ bit 0 is set, ignore other bits
      // process interrupt 0
    end
    4'b??1?: begin // test if IRQ bit 1 is set, ignore other bits
      // process interrupt 1
    end
    4'b?1??: begin // test if IRQ bit 2 is set, ignore other bits
      // process interrupt 2
    end
    4'b1???: begin // test if IRQ bit 3 is set, ignore other bits
      // process interrupt 3
    end
  endcase
end
endmodule

```

Example 5 — Interrupt decoder modeled with casez and don't care bits

There are several modeling guidelines that should be followed when using the `casez` and `casex` decision statements. These guidelines are beyond the scope of this paper, but are covered in a paper presented at a previous SNUG conference, “*RTL Coding Styles That Yield Simulation and Synthesis Mismatches*”[7].

3.0 Synthesizing decision statements

The DC synthesis tool looks for two key factors when translating Verilog decision statements into hardware implementation.

- Is the decision statement fully specified?
- Can the decision control expressions be evaluated in parallel, or does the decision evaluation order need to be maintained?

3.1 Fully specified decisions versus latched logic

A fully specified decision statement defines a branch of execution for all possible values of the controlling expression. For `if...else` decisions, this means that every `if` branch must have a matching `else` branch. For `case` statements, either all possible values of the case expression must have a matching case item value, or there must be a default branch to cover any unspecified values. Synthesis will also consider a decision statement to be fully specified if all variables that assigned values within the decision block are assigned a default value before the decision statement.

DC only considers 2-state values when determining if all possible case expression values have been specified. Logic values of X and Z are not considered.

The following example of a 4-to-1 MUX illustrates a fully specified case statement.

```

module mux4to1a (output reg y,
                 input a, b, c, d,
                 input [1:0] select );

  always @* begin
    case (select)
      2'b00: y = a;
      2'b01: y = b;
      2'b10: y = c;
      2'b11: y = d;
    endcase
  end
endmodule

```

Example 6 — 4-to-1 MUX using a case statement that is fully specified

In the example above, DC will consider the `case` statement to be fully specified, even though there is no default branch. The `select` input is 2 bits wide, so it can have 4 possible values that comprise of zeros and ones. All four of these possible values are specified as case item expressions, making the case statement fully specified for synthesis purposes.

The following example illustrates a `case` decision that is *not* fully specified:

```

module mux3to1b (output reg y,
                 input a, b, c,
                 input [1:0] select );

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

```

Example 7 — 3-to-1 MUX using a case statement that is not fully specified

This example illustrates a 3-to-1 MUX. As such, there are only three possible branches, all of which are specified. However, the 2-bit `select` input can have 4 possible values. DC will consider the case statement to not be fully specified, because a value of `2'b11` for `select` is not specified. Should this value occur, then no branch will be executed. Any variables that are assigned in the `case` statement (`y`, in this example) will remain unchanged. This implies that the values of these variables must be preserved in hardware through some type of storage. To maintain this software simulation behavior, DC will add latched logic functionality in the hardware implementation.

Note that a decision statement that is not fully-specified is one reason that synthesis will add latched logic to a design, but it is not the only reason. Another common cause of latches being added by synthesis is when not all variables are assigned a value each time combinational logic blocks are executed (e.g., if each branch of a decision statement assigns to different variables).

3.2 Parallel evaluation versus priority encoded evaluation

A parallel decision statement is one in which, for each decision control value, at most only one decision branch can be true. This means the decision expressions can be evaluated in any order, rather than a specific order.

Verilog simulation semantics define that a series of `if...else` decisions and `case` statements are evaluated sequentially. When an expression in the sequence evaluates as true, that branch is executed, and no further expressions are evaluated. This sequential behavior creates a priority to the expressions. To mimic this software simulation behavior in hardware, DC will add priority encoded logic to the hardware implementation.

Priority encoding adds both extra logic gates and longer timing paths to the logic. The extra gates and timing for priority encoded logic can be a problem in an area-critical or timing-critical design. Therefore, DC will try to optimize away this logic. If DC can determine that each case item is mutually exclusive, then the order in which the expressions are evaluated does not matter. The priority encoded logic is not required, and can be optimized out of the circuit. Some synthesis compilers will also do this type of optimization with `if...else` decision series.

Consider the following example:

```
module mux4to1b (output reg y,
                input      a, b, c, d,
                input [1:0] select );

    always @* begin
        case (select)
            2'b00: y = a;
            2'b01: y = b;
            2'b10: y = c;
            2'b11: y = d;
        endcase
    end
endmodule
```

Example 8 — 4-to-1 MUX using a case statement

This example is considered to be a parallel case statement by synthesis. In simulation, the value of `select` will be evaluated in the order of the case items, and only the first matching case item branch will be executed. However, the value of each of the expressions for the case items is unique from the value of all other case item expressions. Therefore, at most, only one branch will be executed in simulation, no matter in what order the case items are listed. DC will recognize this, and optimize away the priority encoding logic in the implemented hardware.

The next example shows an interrupt decoder modeled using a `casez` statement with don't care bits set in each case item expression (e.g. `4'b???1`).

```
module interrupt_decode3 (output reg something,
                        input [3:0] IRQ );
```

```

always @* begin
  // set outputs to defaults for when there is no interrupt
  casez (IRQ)          // use don't care bits when evaluating IRQ
    4'b???1: begin    // test if IRQ bit 0 is set, ignore other bits
      // process interrupt 0
    end
    4'b??1?: begin   // test if IRQ bit 1 is set, ignore other bits
      // process interrupt 1
    end
    4'b?1??: begin   // test if IRQ bit 2 is set, ignore other bits
      // process interrupt 2
    end
    4'b1???: begin   // test if IRQ bit 3 is set, ignore other bits
      // process interrupt 3
    end
  endcase
end
endmodule

```

Example 9 — Interrupt decoder modeled with priority given to lower numbered interrupts

Verilog semantics for case statements specify that only the branch for the first matching case item will be executed. This means that, in simulation, if two or more interrupt request bits are set at the same time, the lowest bit set will be the interrupt that is processed. There is a clear priority to the order in which the interrupt bits are evaluated. DC will recognize that it is possible for more than one case item expression to be true at the same time (the case item values are not unique, or mutually exclusive). Therefore, synthesis will add priority encoded logic in the design implementation. The functionality of the design implementation will match the behavior of the software simulation of the case statement.

4.0 Synthesis `full_case` and `parallel_case` pragmas—the villains of synthesis

In most situations, DC will do the right thing. Synthesis will correctly add priority encoded logic and/or latched logic to the design implementation where priority logic or latched logic behavior is represented in Verilog software simulation. When simulation does not depend on priority encoding or value storage, synthesis will correctly optimize away the extra logic.

There are times, usually rare, when the design engineer knows (or at least assumes) something about the design that the synthesis tool cannot see by just examining the decision statement. For example, the engineer may know (or assume) that, for a 3-to-1 MUX, the design will never generate an illegal select value.

DC provides two ~~villains~~ “pragmas” that allow design engineers to force synthesis to leave out the decision statement latched logic and/or priority encoded logic in the design implementation. These are the synthesis “`full_case`” pragma and the synthesis “`parallel_case`” pragma. These synthesis pragmas are specified in the form of a comment immediately following the `case` (or `casez` or `casex`) keyword and the case expression. Either the Verilog `//` single-line comment or the `/*...*/` multi-line comment can be used. The comment must begin with “synopsys”. For example:

```

case (select) // synopsys full_case
    ...
endcase

case (select) /* synopsys parallel_case */
    ...
endcase

```

Normally, comments in Verilog code are ignored, but DC looks for comments that begin with “synopsys”, and treats these comments as special directives to the synthesis process. The IEEE 1364.1 Verilog RTL synthesis standard[3] provides another way to specify synthesis directives, using the Verilog attribute construct. The examples in this paper, however, use the older and more traditional comment form of synthesis pragmas.

NOTE: DC only supports the use of these pragmas with **case**, **casez** and **casex** statements. DC does not provide a way to modify the synthesis of **if...else** decision statements.

4.1 The **full_case** ~~villain~~ pragma

The **full_case** pragma instructs DC to assume that a **case** statement (or **casez** or **casex** statement) is fully specified, as described in section 3.1. This means that synthesis will ignore any values for the case expression that do not match a case item.

Example 7, presented in section 3.1, showed a 3-to-1 MUX using a **case** statement. DC would see this **case** statement as not being fully specified, and would add latched logic to the design implementation to allow for the unspecified **select** value of **2'b11**. DC is correct to add this extra logic. In simulation, a **select** value of **2'b11** would mean that no branch is executed, and, therefore, the output of the MUX would retain its previous value. The additional latched logic in the implementation ensures that the post-synthesis design functionality will match the pre-synthesis simulation behavior, at least for the MUX logic.

However, it might be that the design would never allow a **select** value of **2'b11** to occur, and therefore the additional latched logic is not actually needed. If the design engineer knows that the unspecified values can never occur, then the engineer can inform DC of this fact by specifying that the case statement is to be treated as a **full_case**. The following example adds the **full_case** pragma to the 3-to-1 MUX example previously described in Example 7.

```

module mux3to1c (output reg y,
                 input a, b, c,
                 input [1:0] select );

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

```

Example 10 — 3-to-1 MUX using the **full_case synthesis ~~villain~~ pragma**

With the `full_case` pragma specified, synthesis will ignore any unspecified values for `select` (`2'b11`) in this example. No latches will be added to the design implementation for this MUX logic. This will reduce both the gate count and timing for the MUX circuitry.

The question every design and verification engineer should be asking right now is, “*What will the design do, should a `select` value of `2'b11` occur?*”. This question is answered in section 4.4.

4.2 The `parallel_case` ~~villain~~ pragma

The `parallel_case` pragma instructs DC to assume that a `case`, `casez`, or a `casex` statement can be evaluated in parallel, as described in section 3.2. In a parallel case statement, the value of each case item expression is unique from the value of all other case item expressions. This means that synthesis does not need to maintain the order in which the case items are listed in the case statement.

Example 9 in section 3.2 illustrated an interrupt decoder modeled using a `casez` statement and don’t care bits in the case item expression. If two interrupt bits are set at the same time, this example explicitly gives priority to the lower numbered interrupt. DC correctly recognizes this, and implements the design with priority encoded logic. It may be, however, that the design engineer knows that, in the overall design, it is not possible for more than one interrupt to ever occur at the same time (perhaps because the interrupts only come from a single source). In this situation, the design engineer needs to tell the synthesis compiler this additional information, because it is not apparent in the code for the interrupt decoder. This can be done using the `parallel_case` pragma, as follows.

```

module interrupt_decode4 (output reg something,
                        input [3:0] IRQ );

always @* begin
    // set outputs to defaults for when there is no interrupt
    casez (IRQ)          // synopsys parallel_case
        4'b???1: begin    // test if IRQ bit 0 is set, ignore other bits
            // process interrupt 0
        end
        4'b??1?: begin    // test if IRQ bit 1 is set, ignore other bits
            // process interrupt 1
        end
        4'b?1??: begin    // test if IRQ bit 2 is set, ignore other bits
            // process interrupt 2
        end
        4'b1???: begin    // test if IRQ bit 3 is set, ignore other bits
            // process interrupt 3
        end
    endcase
end
endmodule

```

Example 11 — Interrupt decoder using the `parallel_case` synthesis ~~villain~~ pragma

The `parallel_case` pragma instructs the synthesis compiler to ignore the possibility of overlap between the case item values, and treat each case item as if it is unique from all other case items.

DC will do exactly what it has been told to do. Since the design engineer, in his infinite wisdom (or foolish assumption) has told DC that each case item is unique, DC will leave out the priority encoding logic from the design implementation. This will result in a smaller and faster design.

Once again, the astute design or verification engineer should be asking the question “*What will happen if two or more interrupt bits do get set at the same time?*”. When the `parallel_case` pragma is used, having multiple interrupt bits set could be a serious problem in the logic that is implemented by the synthesis. This is discussed in more detail in section 4.4.

4.3 Using `full_case` and `parallel_case` together

The `full_case` and `parallel_case` directives can be used together. There are times when both are needed, in order to obtain the most optimal design implementation. The following example models a 1-hot state machine decoder. In a 1-hot state machine, each bit of the state vector represents a different state. Only a single bit of the state vector is set at a time. One common way to model this decode logic in Verilog is to use a “*reverse case statement*”. With this style, a literal value of single bit of 1 is specified as the case expression. This single bit of 1 is then compared to each bit of the state variable in the case items. This example specifies both the `full_case` and `parallel_case` pragmas.

```
module FSM (...);
  reg [3:0] state; // 4-bit wide state vector

  always @* begin
    case (1'b1) // synopsys full_case parallel_case
      state[0]: begin // test if in state 1 (bit 0 set)
                  // set state 1 output values
                end
      state[1]: begin // test if in state 2 (bit 1 set)
                  // set state 2 output values
                end
      state[2]: begin // test if in state 3 (bit 2 set)
                  // set state 3 output values
                end
      state[3]: begin // test if in state 4 (bit 3 set)
                  // set state 4 output values
                end
    end
  end
endmodule
```

Example 12 — FSM state decoder using the `full_case` and `parallel_case` pragmas

Without the `full_case` and `parallel_case` pragmas, this design, DC would not optimize the implementation of this `case` statement. The synthesis compiler would determine that the case statement is not fully specified; a state value of 0 would result in no case item matching, which means no statements would be executed. Since this is not a fully-specified case statement, synthesis would add in latched logic to the design implementation. Synthesis would also determine that the case items are not mutually exclusive; there is a possibility that two or more case item expressions could be true at the same time. Since the case statement semantics give priority to the lowest numbered bit that is set, synthesis will add priority encoded logic to the design implementation.

In this example, the design engineer needs to be smarter than the synthesis tool, and override the normal synthesis rules. If this is truly a one-hot state machine, then all possible state values have been covered by the case item expression. Therefore, the design engineer needs to tell the synthesis compiler that the case statement is a `full_case`. In addition, the state machine will only be in one state at a time, so there will never be two bits set (hot) at the same time. Therefore, the design engineer also needs to tell the synthesis compiler that this case statement is a `parallel_case`, indicating that each case item is unique.

4.4 Why `full_case` and `parallel_case` are dastardly villains

Sections 4.1, 4.2 and 4.3 have shown that there are times when it is correct to force DC to implement a case statement as if it were fully specified, or that all case items are unique, or both. The `full_case` and `parallel_case` pragmas are sometimes necessary, in order for synthesis to produce the most optimized design implementation. Since these synthesis pragmas are useful, why are they villains?

The reason is that the synthesis pragmas force synthesis to change (or at least ignore) the semantics of Verilog case statements. This means that the software simulation of the case statement might differ from the functionality of the logic that synthesis creates. Example 10 in Section 4.1 illustrated a 3-to-1 MUX modeled with an incomplete case statement (there was no case item for a `select` value of `2'b11`). The villainous `full_case` pragma told synthesis to ignore that value, and not generate any logic in the design to allow for the possibility of both `select` bits being set at the same time.

So what will happen if both `select` bits are set at the same time? In simulation, the answer is well defined. The semantics of the `case` statement are that, if the value of the case expression (`select` in the 3-to-1 MUX example) does not match any case item, and there is no default case, then no statements are executed. The output of the MUX will remain unchanged, retaining its previous value. However, the behavior for when both `select` bits are set in the logic implemented by synthesis is not defined. Something will happen—hardware is always doing something as long as power is applied—but what will happen will be determined by what type of logic gates are used to decode the `select` bits, and how those gates are interconnected. This implementation can be, and likely will be, different for different hardware devices. For example, the logic used to decode the `select` bits in an FPGA might be different than the logic used in an ASIC.

Example 11 in Section 4.2 illustrated an interrupt decoder using a `casez` statement. The design ***assumes*** that there will never be two interrupt requests at the same time, and specifies, using the dastardly `parallel_case` pragma, that synthesis should treat each case expression as if it were unique. But, ***what if the assumption is wrong, and multiple interrupts do occur at the same time?*** Once again, simulation semantics are definitive. The `casez` statement is evaluated in the order in which the case items are listed. The implementation of the design, however, does not include this priority encoding. Something will happen if more than one interrupt occurs at the same time, but exactly what will happen is dependent on the gate level implementation. The pre-synthesis simulation of the interrupt controller will most likely not match the functionality of the design implementation.

This difference in behavior between pre-and post-synthesis is dangerous! In his evil twins paper[6], Cummings cites a case where this difference went undetected in an actual design. An

engineer had blindly added `full_case` and `parallel_case` directives to all case statements in the design, in order to achieve a smaller, faster gate-level implementation of the design. Somewhere in the design, however, was a case statement that did depend on the priority in which the case items were evaluated. In simulation, the design appeared to work correctly—simulation always simulates case statements with priority. Synthesis did exactly what it was told to do by the dastardly `parallel_case` pragma, and left out the priority encoded logic. As a result, the final ASIC did not work correctly, and had to be re-spun, costing the company hundreds of thousands of dollars in actual costs, plus untold costs in project delays.

The synthesis `full_case` and `parallel_case` pragmas can indeed be dastardly villains! But, what makes them villains is that they change the semantics of case statements for synthesis, but they do not change the semantics for simulation. This can create a difference in the simulation of a case statement and the gate-level implementation of that same case statement. The burden is placed on the design engineer to correctly use the `full_case` and `parallel_case` synthesis directives. If misused, it is an all too real possibility that a serious design flaw can go undetected.

As Cummings puts it, “*full_case and parallel_case directives are most dangerous when they work!*”, and “*Educate (or fire) any employee or consultant that routinely adds ‘full_case parallel_case’ to all case statements in the Verilog code*”[6].

5.0 SystemVerilog unique and priority decisions—the designer’s heroes

SystemVerilog adds two modifiers for Verilog decision statements, in the form of two new keywords, `unique` and `priority`. Syntactically, these modifiers are placed just before an `if`, `case`, `casez` or `casex` keyword. For example:

```
unique if ( expression ) statement_or_statement_group
else                statement_or_statement_group

priority case ( case_expression )
  case_item1 : statement_or_statement_group;
  case_item2 : statement_or_statement_group;
  default   : statement_or_statement_group;
endcase
```

In a series of `if...else` decisions, the `unique` or `priority` modifier is only specified for the first `if` statement. The modifier affects all subsequent `if` statements within the sequence.

The `unique` and `priority` decision modifiers provide all of the benefits of the `full_case` and `parallel_case` synthesis pragmas, and, at the same time, remove all of the dangers associated with these pragmas.

5.1 The heroic unique decision modifier

The `unique` decision modifier instructs *all* software tools that each selection item in a series of decisions is unique from any other selection item in that series. That is, every selection value is mutually exclusive. At first glance, it may seem that `unique` does the same thing as the `parallel_case` pragma. There are, however, several important differences between the two.

- **unique** is a keyword in the Verilog source code, not a pragma hidden inside a comment.
- **unique** affects all tools (simulation, formal, etc.), not just synthesis.
- **unique** defines semantic rules that help avoid mismatches between simulation and synthesis; `parallel_case` does not define any semantics, it tells DC to ignore the case statement semantics.
- **unique** will generate warning messages if it is misused in simulation; the `parallel_case` is completely ignored by simulation. It is just a comment.
- **unique** can be used with both `case` statements and `if...else` decision sequences; the `parallel_case` pragma can only be used with `case` statements.

The IEEE P1800 SystemVerilog standard[1] defines semantics for the **unique** decision modifier. All compliant SystemVerilog tools must use the same semantics, including simulators, synthesis compilers, and formal verifiers. The rules are simple, but critical for successful hardware design.

- For `if...else` decision sequences:
 - A warning shall be generated if all of the `if` conditions are false, and there is no final `else` branch. In other words, a warning shall occur if the `if...else` sequence is entered, and no branch is executed.
 - A warning shall be generated if two or more of the `if` conditions are, or can be, true at the same time. That is, a warning will occur if the `if` conditions are not mutually exclusive, or *unique*.
- For `case`, `casez`, or `casex` statements:
 - A warning shall be generated if the case expression does not match any of the case item expressions, and there is no default case. In other words, a warning shall occur if the case statement is entered, and no branch is taken.
 - A warning shall be generated if two or more of the case item expressions are, or can be, true at the same time. That is, a warning will occur if the case item expressions are not mutually exclusive, or *unique*.

Example 13 lists a 3-to-1 MUX with a 2-bit wide `select` line modeled using an `if...else` decision sequence. Example 14 shows the same 3-to-1 MUX, but modeled using a `case` statement. The two examples are functionally equivalent.

```

module mux3to1d (output reg y,
                 input a, b, c,
                 input [1:0] select );

  always @* begin
    unique if (select == 2'b00) y = a;
    else if (select == 2'b01) y = b;
    else if (select == 2'b10) y = c;
  end
endmodule

```

Example 13 — 3-to-1 MUX with unique if...else decisions

```

module mux3to1e (output reg y,
                 input      a, b, c,
                 input [1:0] select );

always @* begin
    unique case (select)
        2'b00: y = a;
        2'b01: y = b;
        2'b10: y = c;
    endcase
end
endmodule

```

Example 14 — 3-to-1 MUX with a unique case statement

With both of these examples, a `select` value of `2'b11` is not defined. As was discussed in Section 4.4, if the villainous `full_case` synthesis pragma were used, this undefined value of `select` would have potentially caused a mismatch between the behavior of pre-synthesis simulation and the gate-level implementation of the design. It would be possible for the a design to appear to function correctly in simulation, and then fail when implemented in hardware. More than one engineer has been burned by the simulation mismatch that can occur when using the `parallel_case` pragma incorrectly. This is why the `full_case` synthesis pragma can be a villain, destroying a design.

The SystemVerilog `unique` decision modifier is a design engineer’s hero. In the examples above, if, during simulation, both `select` bits were ever true when the `if` or `case` decision is entered, a run-time warning message would occur. This run-time warning is a flagrant red flag that the design is not working the way that the design engineer assumed it would. The warning generated by VCS from simulating Example 13 is:

```

RT Warning: No condition matches in 'unique if' statement.
            "example_13.sv", line 7, at time 30

```

Before ever getting to synthesis, the design and/or verification engineer will know that there is a design problem that needs to be corrected. (The design problem might be that a 4-to-1 MUX is needed, or it might be that the unexpected value on `select` should not have occurred, indicating a problem elsewhere in the design.)

The heroic `unique` decision modifier has saved the design from a potential flaw!

Following is a copy of Example 11, from Section 4.2 on the `parallel_case` pragma. The example has been modified to use the SystemVerilog `unique` decision modifier instead of the `parallel_case` pragma.

```

module interrupt_decode5 (output reg something,
                         input [3:0] IRQ );

always @* begin
    // set outputs to defaults for when there is no interrupt
    unique casez (IRQ)
        4'b???1: begin // test if IRQ bit 0 is set, ignore other bits
                    // process interrupt 0
                end
    end
end

```

```

    4'b???1?: begin      // test if IRQ bit 1 is set, ignore other bits
                // process interrupt 1
    end
    4'b?1???: begin      // test if IRQ bit 2 is set, ignore other bits
                // process interrupt 2
    end
    4'b1????: begin      // test if IRQ bit 3 is set, ignore other bits
                // process interrupt 3
    end
endcase
end
endmodule

```

Example 15 — Interrupt decoder using unique case

As was discussed in Sections 4.2 and 4.4, using the `parallel_case` pragma would force synthesis to assume that there would never be two interrupt bits set at the same time. The pragma would effectively optimize the design implementation of the interrupt decoder, but at the risk of creating a design implementation that did not work the same as simulation, should two or more interrupts ever occur at the same time. The `parallel_case` pragma becomes a villain because, in simulation of the case statement, there is no obvious indication should two or more interrupts occur at the same time. The interrupts would be handled in simulation, with priority given to the lowest number interrupt. The assumption that two interrupts should never occur at the same time was wrong, but this could go undetected until after the design was synthesized and implemented in hardware.

With the `unique` case decision modifier, VCS reports the following warnings if unexpected interrupt values are encountered during simulation:

```

IRQ=0001
IRQ=0010
IRQ=0100
IRQ=1000
RT Warning: No condition matches in 'unique case' statement.
    "example_15.sv", line 8, at time 40
IRQ=0000
IRQ=0001
RT Warning: More than one conditions match in 'unique case' statement.
    "example_15.sv", line 8, at time 60
IRQ=1010

```

The unique decision modifier is once again the designer’s hero! It will not allow values the designer assumed would never occur to go undetected. Design and verification engineers will know before synthesizing the design that the assumption of never having two interrupts at the same time was incorrect.

5.2 The heroic priority decision modifier

The `priority` decision modifier instructs *all* software tools that each selection item in a series of decisions must be evaluated in the order in which they are listed. In addition, the IEEE P1800 SystemVerilog standard[1] defines specific semantics for the `priority` decision modifier that all

compliant SystemVerilog tools must abide by. These semantic rules apply to simulation, synthesis, formal verification, lint-checkers, and other software tools that read in SystemVerilog source code. The rules are simple, but very important for successful hardware design.

- For **if...else** decision sequences, a warning shall be generated if all of the **if** conditions are false, and there is no final **else** branch. In other words, a warning shall occur if the **if...else** sequence is entered, and no branch is executed.
- For **case**, **casez**, or **casex** statements, a warning shall be generated if the case expression does not match any of the case item expressions, and there is no default case. In other words, a warning shall occur if the case statement is entered, and no branch is taken.

Example 16, which follows, modifies the interrupt decoder example to use a **priority** case, instead of a **unique** case statement. This change is appropriate when two or more interrupts can occur at the same time, but must be processed in a specific order.

```
module interrupt_decode6 (output reg something,
                        input [3:0] IRQ );

    always @* begin
        // set outputs to defaults for when there is no interrupt
        priority casez (IRQ)
            4'b????1: begin // test if IRQ bit 0 is set, ignore other bits
                // process interrupt 0
            end
            4'b???1?: begin // test if IRQ bit 1 is set, ignore other bits
                // process interrupt 1
            end
            4'b?1???: begin // test if IRQ bit 2 is set, ignore other bits
                // process interrupt 2
            end
            4'b1????: begin // test if IRQ bit 3 is set, ignore other bits
                // process interrupt 3
            end
        endcase
    end
endmodule
```

Example 16 — Multiple interrupt decoder using priority case

By specifying that this case statement is a **priority** case, all SystemVerilog tools will maintain the order of evaluation that is listed in the case statement code. Software simulation would do this anyway—software evaluates decisions sequentially. However, the **priority** keyword clearly documents that the designer intends to have a specific priority to the evaluations. This removes any ambiguity for hardware accelerators, hardware emulators or synthesis compilers, or other tools that might try to optimize the decision statement. The order of evaluation must be maintained by all tools.

For simulation, the **priority** decision modifier also adds run-time checking that at least one branch of a multiple branch decision is executed each time the decision sequence is entered. This helps ensure that all possible conditions that occur during simulation are covered by the decision sequence.

VCS reports the following run-time warning when no case item matches the value of `IRQ`:

```
IRQ=0001
IRQ=0010
IRQ=0100
IRQ=1000
RT Warning: No condition matches in 'priority case' statement.
    "example_16.sv", line 6, at time 40
IRQ=0000
IRQ=0001
IRQ=1010
```

Note that the `IRQ` value of `4'b1010` did not result in a warning message with a **priority case**, as it did with a **unique case**. The **priority case** allows case expressions values to match more than one case item value, and ensures that only the first matching branch is executed.

The following example modifies the interrupt decoder example so that it only decodes one interrupt at a time. The model gives priority to interrupts with a lower bit number in the `IRQ` vector.

```
module interrupt_decode7 (output reg something,
                        input [3:0] IRQ );

    always @* begin
        // set outputs to defaults for when there is no interrupt
        priority case (IRQ)
            4'b0001: begin // test if IRQ bit 0 is set, ignore other bits
                // process interrupt 0
            end
            4'b0010: begin // test if IRQ bit 1 is set, ignore other bits
                // process interrupt 1
            end
            4'b0100: begin // test if IRQ bit 2 is set, ignore other bits
                // process interrupt 2
            end
            4'b1000: begin // test if IRQ bit 3 is set, ignore other bits
                // process interrupt 3
            end
        endcase
    end
endmodule
```

Example 17 — Single interrupt decoder using priority case

In this example, simulation will work correctly with or without the **priority** decision modifier. Lower numbered interrupts will have priority over higher number interrupts. Without the **priority** modifier, however, DC would synthesize this example incorrectly. A synthesis compiler would see that the case item expressions are mutually exclusive; there is no overlap in the expressions. Therefore, DC would optimize away the priority encoded logic, and create a gate-level implementation that evaluates the four interrupt values in parallel. Adding the **priority** decision modifier will prevent this from happening. The **priority** modifier requires that all tools maintain the order of evaluation specified in the decision sequence.

In addition, the `priority` decision modifier requires that tools generate a warning message if the case statement is entered and no branch is taken. This example is coded with the assumption that there will never be two interrupt bits set at the same time. Should that occur, no case item expression would match the `IRQ` vector, and, therefore, no interrupt would be serviced. Without the `priority` modifier, a problem such as this could go undetected, or be difficult to detect (and debug). *Once again, the `priority` decision modifier is a hero.* A run-time warning message will be generated if, during simulation, two or more interrupts do occur at the same time.

6.0 Comparing `unique/priority` to `full_case/parallel` case

DC synthesis results can be controlled using either the villainous `full_case` and `parallel_case` synthesis pragmas or the heroic `unique` and `priority` decision modifiers. This section looks at the similarities and differences between controlling synthesis using pragmas versus using the decision modifiers.

6.1 Synthesis of the `unique` decision modifier—the hero defeats the villains

For DC, the `unique` decision modifier is the same as specifying both `full_case` and `parallel_case`. The semantic rules of `unique` have assured (assuming that run-time simulation warnings are not ignored) that all case items used by the design have been specified. The `unique` modifier has also assured (assuming run-time simulation warnings are not ignored) that there will never be two case items that are true at the same time. Thus, synthesis can *safely* fully optimize a `unique` case statement, removing any latched logic that might have been caused by a case statement that is not fully specified, and removing any priority encoded logic.

The operative word in the last sentence above is *safely*. This is where `unique` becomes a hero, defeating `full_case` and `parallel_case` villains. With the `full_case` and `parallel_case` pragmas, synthesis would perform the latch and priority optimizations, regardless of whether the optimizations were safe or not. When using the pragmas, DC may generate a warning message that a case statement does not have mutually exclusive case item values, or that the case statement is not fully specified. However, the designer has already assumed that the design will not pass values to the decision statement that would cause problems. Therefore, the warnings from the dastardly `full_case` and `parallel_case` pragmas are often ignored by design engineers. And, since verification engineers seldom run synthesis, the verification engineer may not even be aware that DC may have generated warning messages.

With the `unique` modifier, proper simulation can verify that the design will indeed function correctly with the synthesis optimizations. If the design passes values to the decision statement that are different than the designer's assumptions, the `unique` decision modifier flags the problem, and saves the day. (And saves the designer engineer from embarrassment!)

When comparing the synthesis pragmas to the `unique` decision modifier, it is also important to note that the `full_case` and `parallel_case` pragmas can only be used with `case`, `casez` and `casex` statements. The SystemVerilog `unique` decision modifier can also be used with `if...else` decision sequences, thus allowing synthesis compilers to safely optimize this form of multiple-branch decisions, as well.

6.2 Synthesis of priority decision modifier—the hero wins again

There is no synthesis pragma that compares to the SystemVerilog `priority` decision modifier. The `priority` keyword both documents and enforces the intended order of multiple branch decision statements. The importance of this difference was illustrated and described in Example 17, in Section 5.2, above.

The `priority` decision modifier is a hero, because, in simulation, `priority if` or `priority case` will cause a warning message to be generated if a decision sequence is entered, and no branch is taken. Thus, synthesis can *safely* assume that the decision sequence is fully specified (assuming, again, that run-time warning messages during simulation are not ignored).

A second reason the `priority` decision modifier is a hero is that it will not allow DC to mistakenly optimize away the priority encoding of the case item order, when the designer wants to maintain that priority. There is no equivalent synthesis pragma to prevent priority optimization.

Prior to the introduction of the SystemVerilog `priority` decision modifier, some engineers and consultants recommended that, when a decision sequence had to be maintained, an `if...else` decision sequence should be used instead of a case statement. The reasoning behind this is that DC does not attempt to optimize away the priority encoding of an `if...else` decision sequence. This author disagrees with this recommendation! While the `if...else` decision sequence may synthesize with priority encoded logic when using DC, there are other synthesis compilers that attempt to optimize an `if...else` decision sequence in the same way in which case statements are optimized. Nor is there any guarantee that future versions of DC won't apply priority logic optimizations to `if...else` decision sequences.

The SystemVerilog `priority` decision modifier can also be used with `if...else` decision sequences. This ensures that a synthesis compiler will not optimize away the priority of the decision sequence. It also helps ensure that the decision sequence is complete, by generating run-time warnings during simulation, if no branch is executed.

7.0 Alternatives to the unique and priority decision modifiers

There are no alternatives to the new SystemVerilog `unique` and `priority` decision modifiers that provide all of the same capabilities.

For synthesis, the only alternative is to use the `full_case` and/or `parallel_case` synthesis pragmas. Hopefully those who have read this far into this paper now recognize the perils that go with using these villains (see also references [6] and [7]).

For software simulation, the run-time checking of decision statements can be accomplished in at least two alternatives to `unique` and `priority`:

- Extra Verilog code can be added to each decision statement to trap values that were not expected. A common style is to use the default branch of case statements to trap case expression values that do not match any case item expressions, and to propagate X values indicating that a problem has occurred. Trapping values that match two or more case expression items is more difficult, but can be done.

- Assertion statements can be added to each decision statement. This approach also requires adding multiple lines of code to a decision statement, but may be more succinct than using the regular Verilog language. An advantage of assertions is the ability to control the generation of errors and warning, based on severity, simulation time, or other factors.

While useful, these simulation alternatives to `unique` and `priority` have important shortcomings. The alternatives only affect specific tools, and not all tools that will read in the same Verilog code. Also, these simulation alternatives require adding a number of lines of code to the design models, which may have to be hidden from other tools, such as synthesis compilers, using conditional compilation.

One of the great advantages of the SystemVerilog `unique` and `priority` decision modifiers is how simple they are to use. Another significant advantage is that the keywords are intended to work with any type of software tool, and work in a consistent, meaningful way.

There are no true alternatives that offer comparable capabilities to the `unique` and `priority` decision modifiers.

8.0 Recommendations

Design engineers should adopt the following simple guidelines in RTL models intended to be both simulated and synthesized:

Guideline 1: Use the `unique` decision modifier if each branch of the decision statement is controlled by a different value, and two or more branches should never be true at the same time.

Guideline 2: Use the `priority` decision modifier if it is possible for more than one branch control to be true at the same time, but there should never be a situation where no branch of a decision is executed.

Guideline 3: Do not specify a decision modifier when a decision sequence does not need to be fully specified. Typically, this is when modeling a latch, or a sequential device like a flip-flop, or when a default assignment to all combinatorial outputs has been made before the decision sequence.

Guideline 4: Never, ever, at any time, use the dastardly synthesis `full_case` and `parallel_case` synthesis pragmas. The SystemVerilog `unique` and `priority` decision modifiers make these pragmas obsolete.

Some of the reasons for these guidelines are:

- The `unique` and `priority` decision modifiers document the design engineer's expectations about the decision statement.
- The decision modifiers specify how all design tools should interpret the decision statement, ensuring consistency between EDA tools such as software simulators, hardware emulators, hardware accelerators, synthesis compilers, formal verifiers, and lint checkers.
- The decision modifiers force the design engineer to think about what values need to be evaluated by the decision statement. Run-time simulation warnings will be generated if a value occurs for which the engineer did not plan.

- The decision modifiers force the design engineer to think about how the decision statement is expected to be implemented in logic gates (parallel evaluation, priority encoded evaluation, latched outputs, etc.).
- The decision modifiers provide run-time simulation warnings if values are encountered for which the decision statement does not allow. Design errors can be identified much earlier in the design cycle, before synthesizing the design.

Adding these decision modifiers is not essential in testbench code, but judicious usage may be useful, because the simulation warnings if a decision statement does not take any branch.

9.0 Conclusions

The synthesis `full_case` and `parallel_case` pragmas are villains that can destroy a design (not to mention an engineers reputation). This paper presented a number of reasons why these synthesis pragmas could cause design problems. Nonetheless, there are occasions when the functionality of one or both of these pragmas is necessary, in order to obtain optimal design implementation from DC.

At long last, these synthesis villains have been defeated! The SystemVerilog **unique** and **priority** decision modifiers replace the dastardly synthesis pragmas, and offer greater safety. These keywords can prevent dangerous simulation versus synthesis mismatches. The **unique** and **priority** decision modifiers are essential for good, efficient hardware design. All current and future design projects should take advantage of these powerful decision modifiers.

There are so many new and powerful features in SystemVerilog, that the significance of the **unique** and **priority** keywords can easily be overlooked. *Don't make that mistake!* The **unique** and **priority** decision modifiers are heroes that can prevent difficult to find design errors! Using these new keywords is an essential part of ensuring that large complex designs will function correctly, both before and after synthesis.

Some of the reasons that these new keywords are the designer's heroes are:

- The **unique** and **priority** keywords affect all Verilog design tools, and not just synthesis.
- These new keywords enhance the usage of software simulation, synthesis, hardware acceleration, hardware emulation, formal verification, lint checking and any other tool that must execute Verilog `if...else` decisions and `case`, `casez` or `casex` statements.
- The **unique** and **priority** keywords clearly and simply document the design engineers expectations for how decision statements will execute.
- The **unique** keyword generates a run-time simulation warning if two branches of a decision state could potentially execute at the same time. These simulation warnings help ensure that the designer's assumptions about each decision statement are indeed correct.
- The **priority** keyword both documents and enforces the intended order of multiple branch decision statements, and does so for all types of Verilog design tools. The `full_case` and `parallel_case` synthesis pragmas do not offer this capability, even for synthesis.
- Both the **unique** and **priority** keywords generate run-time simulation warnings if a decision statement does not execute any branch when it is entered.

- The `unique` and `priority` keywords can be used with both `if...else` decisions and `case`, `casez` or `casex` statements, whereas the synthesis pragmas can only be used with case statements. This gives designs more flexibility in modeling, and more control over the synthesis process.

These are just some of the important reasons for using the new SystemVerilog `unique` and `priority` decision modifiers. The real beauty of these new keywords is their simplicity. It takes no real effort whatsoever to add these keywords to decision statements in a design. Yet the benefits for using these simple keywords are enormous. Just one run-time simulation warning that a decision statement did not execute as expected can save hours of verification and debug time. Worse yet, without these simple keywords, major design flaws can appear to simulate correctly, and fail in actual hardware.

Begin using these new decision modifiers now—they are heroes that can defeat potential design flaws caused by synthesis pragmas! The Synopsys Verilog design tools support the `unique` and `priority` keywords. Tools from many other companies also support the constructs. There are few, if any, reasons not to take advantage of the powerful, and essential, `unique` and `priority` decision modifiers.

10.0 References

- [1] “*IEEE P1800-2005 standard for the SystemVerilog Hardware Description and Verification Language, ballot draft (D4)*”, IEEE, Piscataway, New Jersey, 2001. ISBN TBD. The P1800 standard was not yet published at the time this paper was written, but the P1800 standard is based on the Accellera SystemVerilog 3.1a standard, which it is available at www.accellera.org.
- [2] “*IEEE Std. 1364-2001 standard for the Verilog Hardware Description Language*”, IEEE, Piscataway, New Jersey, 2001. ISBN 0-7381-2827-9.
- [3] “*IEEE Std. 1364.1-2002 standard for Verilog Register Transfer Level for Synthesis*”, IEEE, Piscataway, New Jersey, 2001. ISBN 0-7381-3501-1.
- [4] “*Synopsys SystemVerilog Synthesis User Guide*”, Version V-2004.06, June 2004. Synopsys, Mountain View, CA.
- [5] Stuart Sutherland, Simon Davidmann and Peter Flake, *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling*. Published by Springer, Boston, MA, 2004, ISBN: 0-4020-7530-8.
- [6] Clifford Cummings, “*‘full_case parallel_case’, the Evil Twins of Verilog Synthesis*”, paper presented at SNUG Boston, 1999.
- [7] Don Mills and Clifford Cummings, “*RTL Coding Styles That Yield Simulation and Synthesis Mismatches*”, paper presented at SNUG San Jose, 1999.

11.0 About the author

Mr. Stuart Sutherland is a member of the IEEE P1800 working group that is defining SystemVerilog, and is the technical editor of the SystemVerilog Reference Manual. He has been involved with the definition of the SystemVerilog standard since work began in 2001. He is also a member of the IEEE 1364 Verilog standards working group. Mr. Sutherland is an independent Verilog consultant, specializing in providing comprehensive expert training on the Verilog HDL, SystemVerilog and PLI. Mr. Sutherland can be reached by e-mail at stuart@sutherland-hdl.com.