I'm Still In Love With My X!

(but, do I want my X to be an optimist, a pessimist, or eliminated?)

Stuart Sutherland
SystemVerilog Trainer and Consultant
Sutherland HDL, Inc.
Portland, Oregon
stuart@sutherland-hdl.com

***Abstract*—This paper explores the advantages and hazards of X-optimism and X-pessimism, and of 2-state versus 4-state simulation. A number of papers have been written over the years on the problems of optimistic versus pessimistic X propagation in simulation. Some papers argue that Verilog/ SystemVerilog is overly optimistic, while other papers argue that SystemVerilog can be overly pessimistic. Which viewpoint is correct? Just a few years ago, some simulator companies were promising that 2-state simulations would deliver substantially faster, more efficient simulation run-times, compared to 4-state simulation. Now it seems the tables have turned, and Verilog/SystemVerilog simulators are providing modes that pessimistically propagate logic X values, with the promise that 4-state simulation will more accurately and efficiently detect obscure design bugs. Which promise is true? This paper answers these questions.**

Keywords—Verilog, SystemVerilog, RTL simulation, 2-state, 4-state, X propagation, X optimism, X pessimism, register initialization, randomization, UVM

1. Introducing My X

SystemVerilog uses a four-value logic system to represent digital logic behavior: 0, 1, Z (high-impedance) and X (unknown, uninitialized, or don't care). Values 0, 1 and Z are abstractions of the values that exist in actual silicon (abstract, because these values do not reflect voltage, current, slope, or other characteristics of physical silicon). The fourth value, X, is not an abstraction of actual silicon values. Simulators can use X to indicate a degree of uncertainty in how physical hardware would behave under specific circumstances, i.e., when simulation cannot predict whether an actual silicon value would be a 0, 1 or Z. For synthesis, logic X provides design engineers a way to specify "don't care" conditions, where the engineer is not concerned about whether actual hardware will have a 0 or a 1 value for a specific condition.

X values are useful, but can also be a challenge for design verification. Of particular concern is how X values propagate through digital logic in RTL and gate-level simulation models. A number of conference papers have been written on this topic. The title of this paper is inspired by two earlier papers on X propagation, "*The Dangers of Living with an X*" by Turpin [1] and "*Being Assertive with Your X*" by Mills [2], presented in 2003 and 2004, respectively. Both the SystemVerilog standard and SystemVerilog simulators have added many new features since those papers were written. This paper reiterates concepts and good advice from earlier papers, and adds coding guidelines that reflect the latest in the SystemVerilog language and software tool features.

Terminology: For the purposes of this paper, *X-optimism* is defined as any time simulation converts an X value on an expression or logic gate input into a 0 or a 1 on the result. *X-pessimism* is defined as any time simulation passes an X on an input to an expression or logic gate through to the result. As will be shown in this paper, sometimes X-optimism is desirable, and sometimes it is not. Conversely, in different circumstances, X-pessimism can be the right thing or the wrong thing.

Note: In this paper, the term "*value sets*" is used to refer to 2-state values (0 and 1) and 4-state values (0, 1, Z, X). The term "*data types*" is used as a general term for all net types, variable types, and user-defined types. The terms *value sets* and *data types* are not used in the same way in the official IEEE SystemVerilog standard [3], which is written primarily for companies that implement software tools such as simulators and synthesis compilers. The SystemVerilog standard uses terms such as "*types*", "*objects*" and "*kinds*", which have specific meaning for those that implement tools, but which this author feels are neither common place nor intuitive for engineers that use the SystemVerilog language.

2. How did my one (or zero) become my X?

Logic X is a simulator's way of saying that it cannot predict whether the value in actual silicon would be 0 or 1.

There are several conditions where simulation will generate a logic X:

- Uninitialized 4-state variables
- Uninitialized registers and latches
- Low power logic shutdown or power-up
- Unconnected module input ports
- Multi-driver Conflicts (Bus Contention)
- Operations with an unknown result
- Out-of-range bit-selects and array indices
- Logic gates with unknown output values
- Setup or hold timing violations
- User-assigned X values in hardware models
- Testbench X injection

2.1. Uninitialized 4-state variables

The SystemVerilog keywords that will declare or infer a 4-state variable are: **`var`**, **`reg`**, **`integer`**, **`time`**, and, depending on context, **`logic`**.

The **`var`** keyword explicitly declares a variable. It can be used by itself, or in conjunction with other keywords. In most contexts, the **`var`** keyword is optional, and is seldom used.

```
var integer i1; // same as "integer i1"
var i2;         // same as "var reg i2"
```

Example 1: The var variable type

The **`logic`** keyword is *not* a variable type or a net type. Nor is the **`bit`** keyword. **`logic`** and **`bit`** define the digital value set that a net or variable models; **`logic`** indicates a 4-state value set (0, 1, Z, X) and **`bit`** indicates a 2-state value set (0, 1). The **`reg`**, **`integer`**, **`time`** and **`var`**. variable types infer a 4-state **`logic`** value set.

The **`logic`** keyword can be used in conjunction with the **`var`**, **`reg`**, **`integer`** or **`time`** keyword or a net type keyword (such as **`wire`**) to explicitly indicate the value set of the variable or net. For example:

```
var  logic [31:0] v; // 4-state 32-bit variable
wire logic [31:0] w; // 4-state 32-bit net
```

Example 2: 4-state variable and net declarations

The **`logic`** (or **`bit`**) keyword can be used without the **`var`** or a net type keyword. In this case, either a variable or net is inferred, based on context. If **`logic`** or **`bit`** is used in conjunction with an **`output`** port, an **`assign`** keyword, or as a local declaration, then a variable is inferred. If **`logic`** is used in conjunction with an **`input`** or **`inout`** port declaration, then a net of the default net type is inferred (typically **`wire`**). An **`input`** port can also be declared with a 4-state variable type, using either the keyword triplet **`input var logic`** or the keyword pair **`input var`**.

```
module m1 (
  input  logic [7:0] i; // 4-state wire inferred
  output logic [7:0] o; // 4-state var inferred
);
  logic [7:0] t; // 4-state var inferred
  ...
endmodule
```

Example 3: Default port data types

The SystemVerilog standard [3] defines that 4-state variables begin simulation with an uninitialized value of X. This rule is one of the biggest causes of X values at the start of simulation.

2.2. Uninitialized registers and latches

"Register" and "latch" refer to models that store logic values over time. This storage behavior can be represented as either abstract RTL procedural code or as low-level User-defined primitives (UDPs). Most often, the storage of registers and latches is modeled with 4-state variables, such as the **`reg`** data type.

Note: The **`reg`** keyword does not, in and of itself, indicate a hardware register. The **`reg`** data type is simply a general purpose 4-state variable with a user-defined vector size. A **`reg`** variable can be used to model pure combinational logic, as well as hardware registers and latches.

In SystemVerilog, 4-state variables begin simulation with an uninitialized value of X. This means that register and latch outputs will have a logic X at the start of simulation. Register outputs will remain an X until the register is either reset or a known input value is clocked into the register. Latch outputs will remain an X until the latch is both enabled and the latch input is a known value. This is true for both abstract RTL simulations and gate-level simulations.

There are ways to handle uninitialized registers and latches. Section 5 discusses using 2-state simulation, Section 7 discuses using proprietary simulation options, and Section 10.2 discusses using the SystemVerilog UVM standard.

2.3. Low power logic shutdown or power-up

Simulation of low-power models can result in registers and latches that had been initialized changing back to logic X during simulation. The effect is similar to uninitialized registers and latches at the beginning of simulation, except that the X storage occurs sometime during simulation, instead of at the beginning of simulation. Once a register has gone back to storing an X, the outputs will remain at X until the register is either reset or a known input value is clocked into the register. A latch that has stored an X will remain an X until the latch is both enabled and the latch input is a known value.

This X lock-up when a design block is powered back up from a low-power mode is especially problematic when registers are only set by loading a value, instead of being reset or preset. This behavior is a 4-state simulation anomaly. Actual silicon registers or latches would power up from a low power mode with a 0 or a 1.

2.4. Unconnected module input ports

Unconnected module inputs generally represent a floating input, and result in a simulation value of Z on that input (assuming the input data type is **`wire`**, which is the default in SystemVerilog). When an input floats at high-impedance, it will often result in a logic X elsewhere within the model.

2.5. Multi-driver Conflicts (Bus Contention)

SystemVerilog net types allow multiple outputs to drive the same net. Each net type (**`wire`**, **`tri`**, **`tri0`**, **`tri1`**, **`wor`**, **`wand`** and **`trireg`**) has a built-in resolution function to resolve the combined value of the multiple drivers. If the final value that would occur in silicon cannot be predicted, the simulation value will be an X. (The SystemVerilog-2012 standard also allows engineers to specify user-defined net types and resolution functions, which might also resolve to a logic X under specific conditions).

2.6. Operations with an unknown result

All SystemVerilog RTL operators are defined to work with 4-state values for the operands. Some operators have optimistic rules and others have pessimistic rules. Section 3 (page 4) and Section 4 (page 9) discus when X values can result from optimistic operations and pessimistic operations, respectively.

2.7. Out-of-range bit-selects and array indices

A *bit-select* is used to read or write individual bits out of a vector. A *part-select* reads or writes a group of contiguous bits from a vector. An *array index* is used to access specific members or slices of an array.

When reading bits from a vector, if the index is outside the range of bits in the vector, a logic X is returned for each bit position that is out-of-range. When reading members of an array, if the index is outside the range of addresses in the array, a logic X is returned for the entire word being read. Of course, even in-range bit-selects, part-selects and array selects can result in an X value being returned, if the vector or array being selected contains X values.

Section 4.8, on X-pessimism, discusses reading vector bits and array members with unknown indices. Section 3.7, on X-optimism, discusses writing to vector bits and array members with unknown or out-of-range indices.

2.8. Logic gates with unknown output values

SystemVerilog built-in primitives and User-defined primitives (UDPs) are used to model design functionality at a detailed level of abstraction. These primitives operate on 4-state values for the gate inputs. An input with a logic X or Z value can result in a logic X output value.

2.9. Setup or hold timing violations

SystemVerilog provides timing violation checks, such as `$setup`, `$hold`, and a few more. Typically, these constructs are used by model library developers for models of flip-flops, RAMs, and other devices that have specific timing requirements. These timing checks can be modeled to be either optimistic or pessimistic, should a timing violation occur. An optimistic timing check will generate a run-time violation report when a violation occurs, but leave the values of the model a known value. A pessimistic timing check will generate the run-time violation report and set one or more of the model outputs to X.

2.10. User-assigned X values in hardware models

A common source of X values in RTL simulation is user code that intentionally assigns logic X to a variable. There are two reasons for doing this: to trap error situations in a design such as a state condition that should never occur, and to indicate a "don't care" situation for synthesis. A common example of a user-assigned X is a **`case`** statement, such as this 3-to-1 multiplexor:

```
always_comb begin
  case (select)
    2'b01:   y = a;
    2'b10:   y = b;
    2'b11:   y = c;
    default: y = 'x;  // don't care about any
  endcase             // other values of select
end
```

Example 4: User-assigned X values

In this example, a `select` value of 2'b00 is not used by the design, and should never occur. The default assignment of a logic X serves as a simulation flag, should `select` ever have a value of 2'b00. The same default assignment of X serves as a don't care flag for synthesis. Synthesis tools see this X-assignment as an indication that logic minimization can be performed for any values of the case expression (`select`, in this example) that were not explicitly decoded.

2.11. Testbench X injection

A testbench will often send logic X values into the design being tested. One way this can occur is when a testbench uses 4-state variables to calculate and store stimulus values. These stimulus variables will begin simulation with a logic X, and will retain that X until the testbench assigns a known value to the variable. Often, a test might not make the first assignment to a stimulus variable until many hundreds of clock cycles into a simulation.

Some verification engineers will write a test to deliberately drive certain design inputs to an X value when the design should not be reading those specific inputs. This deliberate X injection can bring out errors in a design, should the design read that input at an incorrect time. For example, a design specification might be that the `data` input is only stored when `load_enable` is high. To verify this functionality was correctly implemented, the testbench could deliberately set the `data` input to an X while `data_enable` is low. If that X value propagates into the design, it can indicate the design has a bug.

3. An optimistic X — is that good or bad?

Optimism*: an inclination to put the most favorable construction upon actions and events or to anticipate the best possible outcome.* [4]

In simulation, X-optimism is when there is some uncertainty on an input to an expression or gate (the silicon value might be either 0 or 1), but simulation comes up with a known result instead of an X. SystemVerilog is, in general, an optimistic language. There are many conditions where an ambiguous condition exists in a model, but SystemVerilog propagates a 0 or 1 instead of a logic X. A simple example of X-optimism is an AND gate. In SystemVerilog, an X ANDed with 0 will result in 0, not X.

An optimistic X can be a good thing! X-optimism can more accurately represents silicon behavior when an ambiguous condition occurs in silicon. Consider the following example, shown in Figure 1.

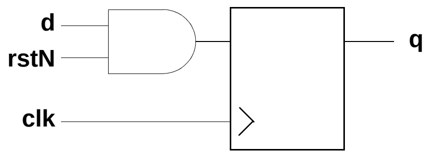

Figure 1: Flip-flop with synchronous reset

This circuit shows a flip-flop with synchronous, active-low reset. In actual silicon, the `d` input might be ambiguous at power-up, powering up as either a 0 or 1. If the `rstN` input of the AND gate is 0, however, the output of the AND gate will be 0, despite the ambiguous power-up value of `d`. This correctly resets the flip-flop at the next positive edge of `clk`.

In simulation, the ambiguous power-up value of `d` is represented as an X. If this X were to pessimistically propagate to the AND gate output, even when `rstN` is 0, the design would not correctly reset, which could cause all sorts of problems. Fortunately, SystemVerilog AND operators and AND gates are X-optimistic. If any input is 0, the result is 0. Because of X-optimism, simulation accurately models silicon behavior, and the simulated models function correctly.

An optimistic X can also be a bad thing! X-optimism can, and will, hide design problems, especially at the abstract RTL level of verification. At best, these design bugs are not caught until late in the design cycle during gate-level simulations or when other low-level analysis tools are used. At worst, design ambiguities that were hidden by X-optimistic simulation might not be discovered until the design has been implemented in actual silicon.

Several X-optimistic SystemVerilog constructs are discussed in more detail in this section.

3.1. If...else statements

SystemVerilog has an optimistic behavior when the *control condition* of an **`if`**...**`else`** statement is unknown. The rule is simple: should the *control condition* evaluate to unknown, the **`else`** branch is executed.

```
always_comb begin
  if (sel) y = a;   // if sel is 1
  else     y = b;   // if sel is 0, X or Z
end
```

Example 5: if...else statement X-optimism

This optimistic behavior can hide a problem with `sel`, the *control condition*. In actual silicon, the ambiguous value of `sel` will be 0 or 1, and **`y`** will be set to a known result. How accurately does SystemVerilog's X-optimistic behavior match the behavior in actual silicon? The answer depends in part on how the **`if`**...**`else`** is implemented in silicon.

The behavior of this simple **`if`**...**`else`** statement might be implemented a number of ways in silicon. Figure 3 and Figure 2 illustrate two possibilities, using a Multiplexor or NAND gates, respectively.

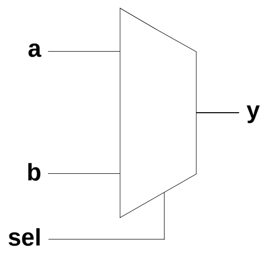

Figure 2: 2-to-1 selection — MUX gate implementation

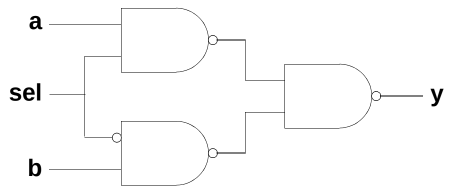

Figure 3: 2-to-1 selection — NAND gate implementation

Table 1 shows the simulation results for an X-optimistic **`if`**...**`else`** when the control expression (`sel`) is unknown, compared to the simulation behavior of MUX and NAND implementations and actual silicon behavior.

inputs			output (y)			
			simulation behavior			actual silicon behavior
sel	a	b	if...else RTL	MUX gate	NAND gates	
X	0	0	0	0	0	0
X	0	1	1	X	X	0 or 1
X	1	0	0	X	X	0 or 1
X	1	1	1	1	X	1

Table 1: if...else versus gate-level X propagation

Some important things to note from this table are:

- For all rows, the **if**...**else** statement propagates a known value instead of the X value of sel. This X-optimistic behavior could hide error conditions in the design.
- For rows 2 and 3, the X-optimistic **if**...**else** behavior only matches one of the possible values that could occur in actual silicon. *The other possible value is not propagated and therefore the design is not verified with that other possible value.*
- The MUX implementation of an **if**...**else** is the most accurate, and propagates an X when there is a potential of actual silicon having either a 0 or a 1.
- The NAND-gate implementation is overly pessimistic for when a and b are both 1 (row 4), and propagates an X value, even though the actual silicon would have a known value of 1.

Following is a more detailed example that illustrates how optimistic **if**...**else** X propagation can hide a design problem. The example is a program counter that: can be reset, can be loaded with a new count, or can increment the current count. The program counter is instantiated within a larger design, cpu, that does not need the ability to load the program counter, and leaves the loadN and new_count inputs unconnected.

```
module program_counter (
  input  logic        clock, resetN, loadN,
  input  logic [15:0] new_count,
  output logic [15:0] count
);
  always_ff @(posedge clock or negedge resetN)
    if (!resetN)      count <= 0;
    else if (!loadN) count <= new_count;
    else              count <= count + 1;
endmodule: program_counter
module cpu (...);
  ...
  program_counter pc (.clock(m_clk),
                      .resetN(m_rstN),
                      .loadN(/* not used */),
                      .new_count(/* not used */),
                      .count(next_addr) );
  ...
endmodule: cpu
```

Example 6: Program counter with unused inputs, X-optimistic rules

In actual silicon, each bit of these unconnected inputs will have ambiguous values — they will be sensed as either 0 or 1, depending on factors such as transistor technology and interconnect capacitance. If actual silicon senses loadN as 1, the counter will increment on each clock, which is the desired functionality. If silicon senses loadN as 0, the counter will load an ambiguous new_count value on each clock, and the program counter will not work correctly.

X-optimism hides this design bug! The loadN and new_count inputs will float at high-impedance (assuming the default net type of **wire**). Instead of seeing loadN as being either 0 or 1, the way silicon would, RTL simulation always takes the **else** branch, which increments the counter. This X-optimistic behavior happens to be the desired behavior for this design, but ***it is a dangerous simulation hazard!*** In RTL simulation, the design appears to work correctly, and a serious design bug could go undetected.

Later sections of this paper show several ways to detect problems with **if** conditions, so that design bugs of this nature do not remain hidden by an optimistic X.

3.2. Case statements without a default-X assignment

The control value of a **case** statement is referred to as the *case expression*. The values to which the control value is compared are referred to as *case items*.

```
always_comb begin
  case (sel)       // sel is the case expression
    1'b1: y = a;   // 1'b1 is a case item
    1'b0: y = b;   // 1'b0 is a case item value
  endcase
end
```

Example 7: case statement X-optimism

Functionally, **case** and **if**...**else** represent similar logic. However, SystemVerilog's X-optimistic behavior for a **case** statement without a **default** branch is very different than an **if**...**else** decision when the select control is unknown, as is shown in Table 2.

inputs			previous value of y	output (y)			
sel	a	b		case RTL	if...else RTL	MUX gate	silicon
X	0	0	0	0	0	0	0
X	0	1	0	0	1	X	0 or 1
X	1	0	0	0	0	X	0 or 1
X	1	1	0	0	1	1	1
X	0	0	1	1	0	0	0
X	0	1	1	1	1	X	0 or 1
X	1	0	1	1	0	X	0 or 1
X	1	1	1	1	1	1	1

Table 2: case versus if...else versus MUX X propagation

Observe in this table that a **`case`** statement without a default branch retains its previous value whenever the case expression is unknown.

A **`case`** statement with a **`default`** assignment of a known value is also optimistic, but in a different way. Consider the following example:

```
always_comb begin
  case (sel)
    1'b1:    y = a;
    default: y = b;
  endcase
end
```

Example 8: case statement with default assignment of a known value X-optimism

If any bit in the *case expression* is an X or Z, the value of the **`default`** *case item* is assigned to `y`, instead of keeping the previous value. Table 3 show this difference.

inputs			previous value of y	output (y)		
sel	a	b		case without default	case with default	silicon
X	0	0	0	0	0	0
X	0	1	0	0	1	0 or 1
X	1	0	0	0	0	0 or 1
X	1	1	0	0	1	1
X	0	0	1	1	0	0
X	0	1	1	1	1	0 or 1
X	1	0	1	1	0	0 or 1
X	1	1	1	1	1	1

Table 3: case with default versus case without default

As can be seen in this table, **`case`** statements with or without a default assignment are X-optimistic, and will hide problems in the *case expression*. With either coding style, the X-optimism does not accurately reflect the ambiguity that exists on the results in actual silicon, should a selection control be ambiguous.

Section 3.2 on pessimistic modeling styles will discuss what happens when a **`case`** statement **`default`** branch assigns an X as the decoded result. Sections 7 and 9 present other ways to reduce or eliminate this X-optimism problem with **`case`** statements.

3.3. Casex, casez and case...inside statements

SystemVerilog's **`casex`**, **`casez`** and **`casez`**...**`inside`** statements allow specific bits to be masked out — i.e., ignored — from being compared for each case branch. Collectively, these three constructs are sometimes referred to as *wildcard case statements*.

With **`casez`**, any bit in the *case expression* or *case item* that is set to Z will be ignored. With **`casex`**, any bit in the *case expression* or *case item* that is either X or Z will be ignored. (In literal numbers, a ? can be used in place of the letter Z.)

```
always_comb begin
  casex (sel)      // sel is 3 bits wide
    3'b1??: y = a;  // matches 100, 101, 110, 111
    3'b00?: y = b;  // matches 000, 001
    3'b01?: y = c;  // matches 010, 011
    default: $error("sel had unexpected value");
  endcase
end
```

Example 9: casex statement X-optimism

By using "don't care" values, the 3 *case items* above decode all 8 possible 2-state values of `sel`. Less obvious is that these *case items* also decode all possible unknown values of `sel`, because the don't care bits in *case items* ignore all values in those bit positions, including X and Z values. Furthermore, any X or Z bits in the *case expression* are also considered to be don't care bits, and are ignored in any comparisons. The values of `sel` will decode as:

- `3'b1??` matches sel values of:
  `100`, `101`, `110`, `111`,
  `10X`, `11X`, `1X0`, `1X1`, `1XX`,
  `10Z`, `11Z`, `1Z0`, `1Z1`, `1ZZ`,
  `1XZ`, `1ZX`,
  `X00`, `X01`, `X10`, `X11`,
  `XXX`, `XZZ`, `XZX`, `XXZ`,
  `ZZZ`, `ZZX`, `ZXZ`, `ZXX`
- `3'b00?` matches sel values of:
  `000`, `001`, `00X`, `00Z`,
  `0X0`, `0X1`, `0XX`, `0XZ`,
  `0Z0`, `0Z1`, `0ZZ`, `0ZX`
- `3'b01?` matches sel values of:
  `010`, `011`, `01X`, `01Z`

- **`default`** does not match any values, because all possible 4-state values have already been decoded by the previous case items.

Using **`casez`** instead of **`casex`** changes the X-optimism. With **`casez`**, only Z values (also represented with a ?) in the *case expression* or *case items* are treated as don't care values.

```
always_comb begin
  casez (sel)     // sel is 3 bits wide
    3'b1??: y = a;  // matches 100, 101, 110, 111
    3'b00?: y = b;  // matches 000, 001
    3'b01?: y = c;  // matches 010, 011
    default: $error("sel had unexpected value");
  endcase
end
```

Example 10: casez statement X-optimism

With casez, the values each *case item* represents are:

- `3'b1??` matches `sel` values of:
  `100`, `101`, `110`, `111`,
  `10X`, `11X`, `1X0`, `1X1`, `1XX`,
  `10Z`, `11Z`, `1Z0`, `1Z1`, `1ZZ`,
  `1XZ`, `1ZX`,
  `ZZZ`, `ZZX`, `ZXZ`, `ZXX`
- `3'b00?` matches `sel` values of:
  `000`, `001`, `00X`, `00Z`,
  `0Z0`, `0Z1`, `0ZZ`, `0ZX`
- `3'b01?` matches `sel` values of:
  `010`, `011`, `01X`, `01Z`
- **`default`** matches `sel` values of:
  `X00`, `X01`, `X10`, `X11`,
  `XXX`, `XZZ`, `XZX`, `XXZ`,
  `0X0`, `0X1`, `0XX`, `0XZ`,

Using **`casez`**, some, but not all, of the possibilities of `sel` having a bit with an X or Z value fall through to the **`default`** statement. Since y is not assigned a value in the default branch, the value of y would not be changed, and would retain its previous value.

The **`case`**...**`inside`** statement is also X-optimistic, but less so than **`casex`** or **`casez`**. With **`case`**...**`inside`**, only the bits in *case items* can have mask (don't care) bits. Any X or Z bits in the *case expression* are treated as literal values.

```
always_comb begin
  case (sel) inside
    3'b1??: y = a;
    3'b00?: y = b;
    3'b01?: y = c;
    default: $error("sel had unexpected value");
  endcase
end
```

Example 11: case...inside statement X-optimism

Using **`case`**...**`inside`**, the values each *case item* represents are:

- `3'b1??` matches `sel` values of:
  `100`, `101`, `110`, `111`,
  `10X`, `11X`, `1X0`, `1X1`, `1XX`,
  `10Z`, `11Z`, `1Z0`, `1Z1`, `1ZZ`,
  `1XZ`, `1ZX`
- `3'b00?` matches `sel` values of:
  `000`, `001`, `00X`, `00Z`
- `3'b01?` matches `sel` values of:
  `010`, `011`, `01X`, `01Z`
- `default` matches `sel` values of:
  `0X0`, `0X1`, `0XX`, `0XZ`,
  `0Z0`, `0Z1`, `0ZZ`, `0ZX`,
  `X00`, `X01`, `X10`, `X11`,
  `XXX`, `XZZ`, `XZX`, `XXZ`,
  `ZZZ`, `ZZX`, `ZXZ`, `ZXX`

All forms of wildcard case statements are X-optimistic, but in different ways. The **`case`**...**`inside`** does the best job of modeling actual silicon optimism, but can still differ from true silicon behavior, and can hide problems with a case expression. Sections 3.3, 7 and 9 discuss ways to reduce or eliminate this X-optimism problem with wildcard case statements.

3.4. Bitwise, unary reduction, and logical operators

Many, but not all, of SystemVerilog's RTL programming operators are X-optimistic. An X or Z bit in an operand might not propagate to an unknown result. For example, 0 ANDed with any value, including an X or Z, will result in 0, and 1 ORed with any value will result in 1. This optimistic behavior can accurately represent the silicon behavior of an actual AND or OR gate, but it can also hide the fact that there was a problem on the inputs to the RTL operation.

The optimistic operators are:

- Bitwise: AND (**`&`**), OR (**`|`**)
- Unary: AND (**`&`**), NAND (**`~&`**), OR (**`|`**), NOR (**`~|`**)
- Logical: AND (**`&&`**), OR (**`||`**), Implication (**`->`**), and Equivalence (**`<->`**)

The logical AND and OR operators evaluate each operand to determine if the operand is true or false. These operators have two levels of X-optimism that can hide X or Z values:

- An operand is considered to be true if any bit is a 1, and false if all bits are 0. For example, the value `4'b010x` will evaluate as true, hiding the X in the least-significant bit..
- Logical operators "short circuit", meaning that if the result of the operation can be determined after evaluating the first operand, the second operand is not evaluated.

The following example illustrates a few ways in which X-optimistic RTL operators could hide a problem in a design.

```
logic [3:0] a = 4'b0010;
logic [3:0] b = 4'b000x;
logic [1:0] opcode;

always_comb begin
  case (opcode) inside
    2'b00: y = a & b;
    2'b01: y = a | b;
    2'b10: y = &b;
    2'b10: y = a || b;
  endcase
end
```

Example 12: Bitwise, unary and logical operator X-optimism

For the values of `a` and `b` shown above:

- `a & b` results in `4'b0000` — the X in `b` is hidden, but the operation result accurately represents silicon behavior.
- `a | b` results in `4'b001x` — the X in `b` is propagated, accurately indicating there will be ambiguity in silicon behavior.
- `&b` results in `1'b0` — the X in `b` is hidden, but the operation result accurately represents silicon behavior.
- `a || b` results in `1'b1` — the X in `b` is hidden, but the operation result accurately represents silicon behavior.

Note that the X-optimism of these operators accurately models silicon behavior. As noted at the beginning of Section 3, there are times that this optimism is desirable and necessary, in order for RTL simulation to work correctly, but the optimism can also obscure design problems.

Not all SystemVerilog operators are optimistic. Sections 4.5 and 4.6 list several operators that are X-pessimistic.

3.5. And, nand, or, nor, logic primitives

SystemVerilog’s **`and`**, **`nand`**, **`or`** and **`nor`** gate-level primitives are used for low-level, timing-detailed modeling. These constructs are often used in ASIC, FPGA and custom model libraries. These primitives follow the same truth tables as their RTL operator counterparts, and have the same X-optimistic behavior.

3.6. User-defined primitives

SystemVerilog provides ASIC, FPGA and custom library developers a means to create custom user-defined primitives (UDPs). UDPs are defined using a 4-state truth table, allowing library developers to define specific behavior for X and Z input values. It is common for developers to “reduce pessimism” by defining known output values for when an input is X or Z, or when there is a transition to or from an X or Z. As with other X-optimistic constructs, UDPs with reduced pessimism might accurately model actual silicon behavior, but can hide inputs that have an X or Z value.

3.7. Array index with X or Z bits for write operations

SystemVerilog is X-optimistic when making an assignment to an array with an ambiguous array index. If the index has any bits that are X or Z, the write operation is ignored, and no location in the array is modified.

```
logic [7:0] RAM [0:255];
logic [7:0] data = 8'b01010101
logic [7:0] addr = 4'b0000000x;

always_latch
  if (write && enable) RAM[addr] = data;
```

Example 13: Array index ambiguity X-optimism

In this example, only the least-significant bit of `addr` is unknown. A pessimistic approach would have been to write an unknown value into the `RAM` locations that might have been affected by this unknown address bit (addresses 0 and 1 in this example). SystemVerilog’s X-optimistic rule, however, acts as if no write operation had occurred. This completely hides the fact that the address has a problem, and does not accurately model silicon behavior.

3.8. Net data types

Net types are used to connect design blocks together. In the original Verilog language, net data types were also required to be used internally within a module for all input ports. In SystemVerilog, module input ports can be either a net or a variable, but the default is still a net type.

Net types have driver resolution functions, which control how simulation resolves multiple drivers on the same net. Multi-driver resolution is important for specific design situations, such as shared data and address busses that can be controlled by more than one device output. When single-source logic is intended, however, the resolution function of a net can optimistically hide design problems by propagating a resolved value instead of an X.

The most commonly used net type in SystemVerilog is the **`wire`** type. The multi-driver resolution for **`wire`** is that driven values of a stronger strength take precedence over driven values of a weaker strength (logic 0 and logic 1 each have 8 strength levels). If, for example, two sources drive the same **`wire`** net, and one value is a weak-0 and the other a strong-1, the **`wire`** resolves to the strong-1 value. If two values of equal strength but opposing logic values are driven, the wire to resolves to a logic X.

Consider the following module port declarations:

```
module program_counter (
  input              clock, resetN, loadN,
  input  logic [15:0] new_count,
  output logic [15:0] count
);
  ...
endmodule: program_counter
```

Example 14: Program counter with default wire net types

In Example 14, `clock`, `resetN` and `loadN` are input ports, but no data type has been defined. These signals will all default to **`wire`** nets. The signal `new_count` is declared as **`input logic`**, and will also default to **`wire`** (**`logic`** only defines that `new_count` can have 4-state values, but does not define the data type of `new_count`). Conversely, `count` is declared as **`output logic`**. Module output ports default to a variable of type **`reg`**, unless explicitly declared a different data type. (***Note:*** The default data type rules changed between the SystemVerilog-2005 and SystemVerilog-2009 standards for when **`logic`** is used as part of port declaration.)

Design bugs can easily occur when a mistake is made and a **`wire`** net, that was intended to only have one driver, is unintentionally driven by two sources. Since **`wire`** types support and resolve multiple drivers, simulation will only propagate an X if the two values are of the same strength and opposing values. Any other combination will resolve to a known value, and hide the fact that there were unintentional multiple drivers.

3.9. Posedge and negedge edge sensitivity

In SystemVerilog, the **`posedge`** keyword represents any transition that might be sensed as a positive going change in silicon. Value changes of 0->1, 0->Z, Z->1, 0->X, and X->1 are all positive edge transitions. A **`negedge`** transition includes the value changes of 1->0, 1->Z, Z->0, 1->X, and X->0.

The following example illustrates a simple RTL register with an asynchronous active-low reset.

```
always_ff @(posedge clk or negedge rstN)
  if (!rstN) q <= 0;
  else       q <= d;
```

Example 15: Edge sensitivity X-optimism

Table 4 shows SystemVerilog's X-optimistic RTL behavior and actual silicon behavior if `clk` transitions from 0 to X (indicating that in silicon, the new value of `clk` might be either 0 or 1, but simulation is not certain which one). This table assumes `rstN` is high (inactive), and only shows the effects of a transition on the clock input. For this table, all signals are 1-bit wide.

inputs		**old q**	**output (q)**	
clk	**d**	**q**	**RTL**	**silicon**
0->X	0	0	0	0
0->X	0	1	0	0 or 1
0->X	1	0	1	0 or 1
0->X	1	1	1	1

Table 4: Ambiguous clock edge X-optimism

As shown in this table, SystemVerilog's X-optimism rules for transitions will behave as if a clock edge occurred every time there is an ambiguous possibility of a positive edge on the clock. The ambiguous clock is hidden, instead of propagating the ambiguity onto the `q` output in the form of an X value.

The behavior of an ambiguous asynchronous reset is more subtle. Actual silicon would either reset or hold its currently stored value. SystemVerilog RTL semantics behave quite differently. If the asynchronous `rstN` transitions from 1->Z, Z->0, 1->X, or X->0, the following results will occur:

inputs		**old q**	**output (q)**	
rstN	**d**	**q**	**RTL**	**silicon**
1->X	0	0	0	0
1->X	0	1	0	0 or 1
1->X	1	0	1	0
1->X	1	1	1	0 or 1

Table 5: Ambiguous reset edge X optimism

This table shows that the ambiguous transition from 1 to X on the reset acts as if a positive edge of the clock occurred. This X-optimism not only hides that there was a problem with the reset, it does not at all behave like actual silicon!

4. A PESSIMISTIC X — IS THAT ANY BETTER?

Pessimism*: an inclination to emphasize adverse aspects, conditions, and possibilities or to expect the worst possible outcome.* [4]

In simulation, X-pessimism occurs when simulation yields an X where there is no uncertainty in actual silicon behavior. A common misconception is that SystemVerilog RTL code is always X-optimistic, and gate-level code is always X-pessimistic. This is not true. Some RTL operators and programming statements are optimistic, but others are pessimistic. Likewise, some gate-level primitives and UDPs are optimistic and some are pessimistic.

While X-optimism often accurately represents actual silicon behavior, the optimism can hide X values by propagating a known result. X-pessimism, on the other hand, guarantees that all ambiguities (one or more bits that are X or Z) will propagate to downstream code, helping to ensure that the problem will be detected, so that it can be debugged and corrected. X-pessimism will not hide design bugs, but there are at least three difficulties that can arise from X-pessimism.

One difficulty of X-pessimism is the point where verification first observes the X might be far downstream from the original source of the problem. An engineer might have to tediously trace an X value back through many lines of code, and over many clock cycles, to find where and when the X originated.

Another difficulty is that X-pessimism can propagate X results, where actual silicon would work without a problem. This section will show several examples where an X value should not have been propagated, but X-pessimism does so anyway. A great deal of engineering time can be lost debugging the cause of a pessimistic X, only to find out that there is no actual design problem.

A third difficulty with X-pessimism is the potential of simulation locking up in an unknown condition, where actual silicon, though perhaps ambiguous about having 0 or 1 values, will function correctly and not lock up. Figure 4 illustrates a common X lock-up situation, a clock divider (divide-by-two, in this example).

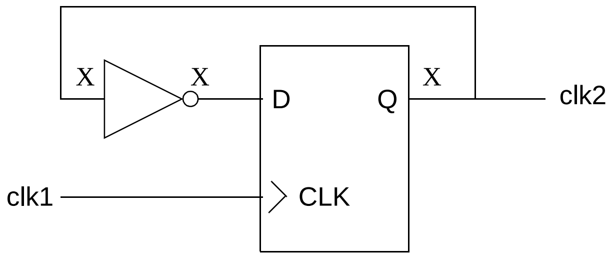

Figure 4: Clock divider with pessimistic X lock-up

In actual silicon, the internal storage of this flip-flop might power up as either a 0 or a 1. Whichever value it is, `clk2` will change value every second positive edge of `clk1`, and give the desired behavior of a divide-by-two. In simulation, however, the ambiguity of starting as either a 0 or 1 is represented as an X. The pessimistic inverter will propagate this X to the `D` input. Each positive edge of `clk1` will propagate this X onto `Q`, which once again feeds back to the input of the inverter. The result is that `clk2` is stuck at an X.

The failure of `clk2` to toggle between 0 and 1 will likely lock up downstream registers that are controlled by `clk2`. The X-pessimistic simulation will be locked up in an X state, where actual silicon would not have a problem. This X-pessimism exists at both the RTL level and at the gate level. The invert operator and the **`not`** inverter primitive are both X-pessimistic. An RTL assignment statement, such as `Q <= D`, and the typical gate-level flip-flop will both propagate an X when `D` is an X.

Several overly pessimistic SystemVerilog constructs that can cause simulation problems are discussed in this section.

4.1. If...else statements with X assignments

Section 3.1 showed how SystemVerilog **`if`**...**`else`** statements are, by default, X-optimistic, and can propagate known values, even though a decision condition has ambiguity (one or more bits at X or Z). It was also shown that this X-optimistic behavior did not always accurately represent silicon behavior.

It is possible to code decision statements to be more X-pessimistic. Consider the following example:

```
always_comb begin
  if (sel)         y = a;
  else
  // synthesis translate_off
    if (!sel)
  // synthesis translate_on
    y = b;
  // synthesis translate_off
  else             y = 'x;
  // synthesis translate_on
end
```

Example 16: if...else statement with X-pessimism and synthesis pragmas

Assuming that `sel` is only 1-bit wide, the **`if`** `(sel)` will evaluate as true if, and only if, `sel` is 1. The first **`else`** branch is taken if `sel` is 0, X or Z (X-optimistic). This first **`else`** branch then tests for **`if`** `(!sel)`, which will evaluate as true if, and only if, `sel` is 0. If `sel` is X or Z, the last **`else`** branch will be taken. This last branch assigns `y` to X, thus propagating the ambiguity of `sel`. This **`if`**...**`else`** statement is now X-pessimistic, propagating X values rather than known values when there is a problem with the select condition.

Note that the additional code to make the **`if`**...**`else`** decision be X-pessimistic might not yield optimal synthesis results. Therefore, the additional checking must be hidden from synthesis, using either conditional compilation (**`` `ifdef ``** commands) or synthesis "pragmas". A pragma is a tool-specific command hidden within a comment or attribute. The synthesis pragma is ignored by simulation, but tells synthesis compilers to skip over any code that should not be synthesized.

4.2. Conditional operator

Coding an X-pessimistic **`if`**...**`else`** statement might not be the best choice for all circumstances. The X-pessimism will not hide a problem with the select condition the way an X-optimistic statement would, as described in 3.1. However, the pessimistic model will also propagate X values at times where there is no ambiguity in hardware. This can occur when the select condition is unknown, but the values assigned in both branches are the same. The value propagated in hardware would be that value, with no ambiguity.

Turpin, in his paper "*The Dangers of Living with an X*"[1], recommends using the conditional operator (**? :**) instead of **if**...**else** statements in combinational logic. The conditional operator is a mix of X-optimistic and X-pessimistic behavior. The syntax is:

condition **?** *expression1* **:** *expression2*

- If the *condition* evaluates as true (any bit is a 1), the operator returns the value of *expression1*.
- If the *condition* evaluates as false (all bits are 0), the operator returns the value of *expression2*.
- If the *condition* evaluates to unknown, the operator does a bit-by-bit comparison of the values of *expression1* and *expression2*. For each bit position, if that bit is 0 in both expressions, then a 0 is returned for that bit. If both bits are 1, a 1 is returned. If the corresponding bits in each expression are different, or Z, or X, then an X is returned for that bit.

The following example and table compare the X-optimistic behavior of **if**...**else**, with a pessimistic **if**...**else**, the mixed-optimism conditional operator, and actual silicon. The table is based on all signals being 1-bit wide.

```
always_comb begin      // X-optimistic if...else
  if (sel) y1 = a;
  else     y1 = b;
end

always_comb begin      // X-pessimistic if...else
  if (sel)        y2 = a;
  else if (!sel)  y2 = b;
  else            y2 = 'x;
end

always_comb begin      // mixed pessimism ?:
  y3 = sel? a: b;
end
```

inputs			output (y1, y2, y3)			
sel	**a**	**b**	**optimistic if...else**	**pessimistic if...else**	**?:**	**silicon**
X	0	0	0	X	0	0
X	0	1	1	X	X	0 or 1
X	1	0	0	X	X	0 or 1
X	1	1	1	X	1	1

Table 6: Conditional operator X propagation compared to optimistic if...else and pessimistic if...else

As can be seen in this table, the conditional operator represents a mix of X-optimism and X-pessimism, and more accurately represents the ambiguities of actual silicon behavior, given an uncertain selection condition. For this reason, Turpin [1] recommends using the conditional operator instead **if**...**else** in combinational logic.

This author does not agree with Turpin's coding guideline for two reasons. First, complex decode logic often involves multiple levels of decisions. Coding with **if**...**else** and **case** statements can help make complex logic more readable, easier to debug, and easier to reuse. Coding the same logic with nested levels of conditional operators obfuscates code and adds a risk of coding errors. Furthermore, synthesis compilers might not permit or properly translate nested conditional operators.

A second reason the conditional operator should not always be used in place of **if**...**else** is when the condition is based on a signal or expression that is more than one bit wide. The condition is evaluated as a true/false expression. In a multi-bit value, if any bit is 1, the condition is considered to be true, even if some other bits are X or Z. The conditional operator will optimistically return the value of expression 1, rather than propagate an X.

Sections 7 and 9 show ways to keep the benefits of **if**...**else** and **case** statements, and also have the benefit of the conditional operator's balance of X-optimism and X-pessimism.

4.3. Case statements with X assignments

A **case** statement can also be coded to be X-pessimistic, as shown in the next example:

```
always_comb begin
  case (sel)
    2'b00:   y = a;
    2'b01:   y = b;
    2'b10:   y = c;
    2'b11:   y = d;
    default: y = 'x;
  endcase
end
```

Example 17: case statement with X-pessimism

If `sel` (the *case expression*) should have any bit at X or Z, none of the explicit *case item* values will match. Without a **default** *case item*, no branch of the **case** statement would be executed, and `y` would retain its old value (X-optimistic, but not accurate silicon behavior). By adding a **default** *case item* that assigns `y` to X, this **case** statement becomes X-pessimistic. If `sel` should have any bit at X or Z, `y` will be assigned X, propagating the ambiguity of the *case expression*.

This coding style is supported by synthesis, and so synthesis pragmas are needed, as was the case with an X-pessimistic **if**...**else**. Engineers should be aware, however, that this coding style can result in synthesis performing logic minimizations that might or might not be desirable in a design. It should also be noted that this coding style can reduce the risk of unintentional latches during synthesis, but it does not guarantee latches will not be inferred.

This pessimistic coding technique does not work as well with **`casex`**, **`casez`** and **`case`**...**`inside`** wildcard case statements. As already shown in 3.3, any don't care bits specified in *case items*, and possibly in the *case expression*, will mask out X or Z values. This masking will always make wildcard case statements at least partially X-optimistic, which can hide design problems, and not accurately represent silicon behavior.

4.4. Edge-sensitive X pessimism

Edge transitions can also be coded in an X-pessimistic style. As described in 3.9, value changes to and from X or Z are treated as transitions, which results in X-optimistic behavior that does not accurately represent possible ambiguities in silicon behavior. The following example shows how this optimism can be removed:

```
always_ff @(posedge clk or negedge rstN)
  // synthesis translate off
  if ($isunknown(rstN) )
    q = 'x;
  // synthesis translate on
  if (!rstN) q <= 0;
  else
  // synthesis translate off
    if (rstN & clk)
  // synthesis translate on
    q <= d;
  // synthesis translate off
  else            q = 'x;
  // synthesis translate on
```

Example 18: Edge sensitivity X-optimism

Note that the extra checking to eliminate the X-optimism is not synthesizable, and needs to be hidden from synthesis compilers. This coding style does prevent the problems of X-optimism for edge sensitivity, but the coding style is awkward and non-intuitive. Section 9 shows another approach to this problem that is preferred by the author.

4.5. Bitwise, unary reduction, and logical operators

While many SystemVerilog operators are X-optimistic (see 3.4), several operators are X-pessimistic. An X or Z bit in an operand will always propagate to an unknown result, even when there would be no ambiguity in the actual silicon result. The pessimistic operators are:

- Bitwise: INVERT (**~**), XOR (**^**), and XNOR (**~^**)
- Unary: XOR (**^**), and XNOR (**~^**)
- Logical: NOT (**!**)

Example 19 illustrates a 5-bit linear-feedback shift register that uses the logical exclusive-OR operator for the feedback taps. The initial value of the LFSR is seeded using a synchronous (multiplexed) active-low reset. In this example, the most-significant bit of the seed value is shown as Z (perhaps due to an interconnect error or some other design bug).

```
logic [4:0] lfsr;
logic [4:0] seed = 5'bz1010; // problem with seed!

always @(posedge clk)
  if (!rstN
    lfsr <= seed;  // seed has a bug with msb
  else begin
    lfsr    <= {lfsr[0], lfsr[4:1]};  // rotate
    lfsr[2] <= lfsr[3] ^ lfsr[0];     // xor tap
    lfsr[3] <= lfsr[4] ^ lfsr[0];     // xor tap
  end
```

Example 19: Bitwise operator with X-pessimism

Simulation cannot predict which value would be seen in silicon for the MSB of seed, but does it really matter? In actual silicon, this floating input would be seen as either a 0 or 1, and the LFSR would work without a problem, though perhaps with a different seed value than intended. A fully X-optimistic model would propagate known values through the LFSR, and hide the ambiguity that exists in silicon. The logical XOR, however, is pessimistic, and the problem with the seed value will show up as X values on the outputs of the LFSR. This X-pessimism does not accurately represent silicon behavior, and can result in X values propagating to downstream logic that can be difficult and time consuming to debug.

Example 20 shows a place where an X-pessimistic operator is desirable. The example is a verification code snippet that takes advantage of — and relies on — the X-pessimism of the exclusive-or operator:

```
logic [3:0] d = 4'b001x;

if (^d === 1'bx)    // check for any unknown bit
  $display("d has one or more X or Z bits");
```

Example 20: Unary-reduction operator with X-pessimism

In this example, if any bit of `d` has a value of X or Z, the unary exclusive-OR operator will return an X, allowing the verification code to detect a problem with `d`.

4.6. Equality, relational, and arithmetic operators

SystemVerilog's equality, relational and arithmetic operators are X-pessimistic. An ambiguity (any bit with X or Z) in an operand will propagate as a result of X. SystemVerilog's X-pessimism for equality, relational, and arithmetic operators sometimes propagates an X where no hardware ambiguity exists. A simple example of this pessimism is a greater-than or less-than comparator.

```
logic [3:0] a = 4'b1100;
logic [3:0] b = 4'b001x;
logic       gt;

always_comb begin
  gt = (a > b);     // compare a to b
end
```

Example 21: Logical operators with X-pessimism

The return from the expression `(a > b)` for the values shown in this example will be 1'bx. In this simple code snippet, it is obvious that the value of `a` is greater than the value of `b`, regardless of the actual value of the least-significant bit of `b`. Actual silicon would not have an ambiguous result.

Arithmetic operations are also X-pessimistic, and will propagate an X if there is any ambiguity of the input values.

```
logic [3:0] a = 4'b0000;
logic [3:0] b = 4'b001z;
logic [3:0] sum;

always_comb begin
  sum = a + b;
end
```

Example 22: Arithmetic operator with X-pessimism

With arithmetic operators, all bits of the operation result are X, which can be overly pessimistic. In this example, `sum` will have a value of `4'bxxxx`. In silicon, only the least-significant bit is affected by the ambiguous bit in `b`. The silicon result would be either `4'b0010` or `4'b0011`. A more accurate representation of the silicon ambiguity would be: `4'b001x`.

Arithmetic operations are X-pessimistic, even when the result in silicon would not have any ambiguity at all.

```
logic [3:0] b = 4'b001x;
logic [4:0] product;

always_comb begin
  product = b * 0;  // multiply b with 0
end
```

Example 23: Overly pessimistic arithmetic operation

In this example, `product` will have an overly pessimistic value of `4'bxxxx`, but in silicon (and in normal arithmetic) zero times anything, even an ambiguous value, would result in 0.

4.7. User-defined primitives

ASIC, FPGA and custom library developers can create custom primitives (UDPs) to represent library-specific components. UDPs are defined using a 4-state truth table, allowing library developers to define specific behavior for X and Z input values. In addition to specifying an output value for each combination of 4-state logic values, the truth tables can also define an output value for transitions between logic values (e.g. what happens on a posedge of clock).

Since each input can have 4 values and 12 transitions to and from those values, these truth tables can be quite large. By default, UDPs are pessimistic — any undefined input value combination that is not explicitly defined in the table will default to a result of X. Library developers often take advantage of this default to reduce the number of lines that need to be defined in the truth table. It is not uncommon for a UDP to only define output values for all possible 2-state combinations and transitions. Any X or Z values on an input, or transitions to and from X or Z, will default to propagating an X on the UDP output.

An inadvertent omission from the UDP truth table will also propagate an X value. This pessimism might be great for finding bugs in the library, but is often a source of frustration for engineers using a library from a 3rd party vendor.

4.8. Bit-select, part-select, array index on right-hand side of assignments

SystemVerilog defines that if the index value of a bit-select, part-select or array index is unknown (any bit is X or Z), the return from the operation will be X. If this X occurs on the right-hand side of an assignment statement, it will propagate to the left-hand side, even if there would be no ambiguity in actual silicon behavior. Consider the following:

```
logic [7:0] data = 8'b10001000;
logic [2:0] i = 4'b00x0;
logic       out;

always_comb begin
  out = data[i];  // variable bit select of data
end
```

Example 24: Ambiguous bit select with X-pessimism

The ambiguity of the value of `i` means that `out` will be X. This pessimistic rule means that problems with an index will propagate to the result of the operation. Since the values of `data` and `i` could change during simulation, this pessimism will be sure to propagate an X whenever an ambiguous value of `i` might occur.

This X-pessimistic rule does not accurately represent silicon behavior, however. There are times when an ambiguity in the index can still result in a known value. With the values shown in Example 24, the ambiguous value of `i` would either select bit 0 or 2. In either case, `out` would receive the deterministic value of 0 in actual silicon.

4.9. Shift operations

SystemVerilog has several shift operators, all of which are X-pessimistic if the shift factor is ambiguous (any bit is X or Z).

```
logic [7:0] data = 8'b10001000;
logic [2:0] i = 4'b00x0;
logic [7:0] out ;

always_comb begin
  out = data << i;  // shift of data
end
```

Example 25: Ambiguous shift operation with X-pessimism

The result of this shift operation is `8'bxxxxxxxx`. As with other pessimistic operations, this will be sure to propagate an X result whenever the exact number of times to shift is uncertain. Setting all bits of the result to X, however, can be overly pessimistic, and not represent actual silicon behavior, where only some bits of the result might be ambiguous, instead of all bits. Given the values in Example 25, data is either shifted 0 times or 2 times. The two possible results are `8'b10001000` and `8'b00100000`. If only the ambiguous bits of these two results were set to X, the X-optimistic value of `out` would be `8'bX0X0X000` instead of an overly pessimistic `8'bxxxxxxxx`.

4.10. X-pessimism summary

Sections 3 and 4 have shown that, while there are times X-optimism and X-pessimism can be desirable in specific situations, neither is ideal for every situation. Subsequent sections in this paper will explore solving this problem by:

- Eliminating X values using 2-state simulation or 2-state data types.
- Breaking SystemVerilog rules in order to find a compromise between X-optimism and X-pessimism.
- Trapping X values rather than propagating Xs.

5. ELIMINATING MY X BY USING 2-STATE SIMULATION

There have been arguments made that it is better to just eliminate logic X rather than to deal with the hazards and difficulties of X-optimism and X-pessimism (see [1], [5], [6]). Some SystemVerilog simulators offer a 2-state simulation mode, typically enabled using an invocation option such as -2state or +2state.

Using 2-state simulation can offer several advantages:

- Eliminates uninitialized register and X propagation problems (the clock divider X lock-up problem shown in Section 4 would not occur in a 2-state simulation).
- Eliminates certain potential mismatches between RTL simulation and how synthesis interprets that code, because synthesis only considers 2-state values in most RTL modeling constructs.
- RTL and gate-level simulation behaves more like actual silicon, since silicon always has a 0 or 1, and never an X.
- Reduces the simulation virtual memory footprint; Encoding 4-date values for each bit, along with strength values for net types, requires much more memory than just storing simple 2-state values.
- Improves simulation run-time performance, since 4-state encoding, decoding, and operations do not need to be performed.

On the other hand, there are several hazards to consider when only 2-state values are simulated.

First, a functional bug in the RTL or gate-level code might go undetected. Logic X is a simulator's way of indicating that it cannot accurately predict what actual silicon would do under certain conditions. When X values occur in simulation, it is an indication that there might be a design problem. Without X values, verification and detection of possible design ambiguities can be more difficult.

A second hazard of 2-state simulation values is that, since there is no X value, simulators must choose either a 0 or a 1 when situations occur where the simulator cannot accurately predict actual silicon behavior. The value that is chosen only represents one of the conditions that might occur in silicon. This means the design is verified for that one value, and leaves any other possible values untested. ***That is dangerous!*** Some simulators handle this hazard by simulating both values in parallel and merging the results of the parallel threads. This concept is discussed in more detail in section 7.

A third hazard is that all design registers, clock dividers, and input ports begin simulation with a value of 0 or 1 instead of X. Silicon would also power up with values of 0 or 1, but are they the same values that were simulated? Cummings and Bening [6] suggest that the most effective 2-state verification is performed by running hundreds of simulations with each register bit beginning with a random 2-state value. Cummings and Bening [6] also note that, at the time the paper was written, a preferred way for handling seeding and repeatability of randomized 2-state register initialization was patented by Hewlett-Packard, and might not be available for public use.

A fourth hazard is that verification cannot check for design problems using a logic X or Z. The following two verification snippets will not work with 2-state simulations:

```
assert (ena == 0 && data === 'Z)
else $error("Data bus failed to tri-state");

assert (^data !== 'X)
else $error("Detected contention on data bus");
```

Example 26: Verification hazard with 2-state simulation

A fifth hazard of 2-state simulation to consider is the use of X assignments within RTL code. The following example illustrates a common modeling style used in combinational logic case statements:

```
case ( {sel1,sel2} )
  2'b01:   result = a + b;
  2'b10:   result = a - b;
  2'b11:   result = a * b;
  default: result = 'X;
endcase
```

Example 27: Assigning 4-state values in 2-state simulation

Synthesis compilers treat assignment of a logic X value as a don't care assignment, meaning the design engineer does not care if silicon sees a logic 0 or a logic 1 for each bit of the assignment. In a 2-state simulation, the simulator must convert each bit of the X assignment value to either a 0 or a 1. The specific value would be determined by the simulator, since 2-state simulation is a feature of the simulator and not the language. There is a high probability that the values used in simulation and the values that occur in actual silicon will not be the same. In theory, this should not matter, since by assigning a logic X, the engineer has indicated that the actual value is a "don't care". The hazard is that, without X propagation, this theory is left unproven in 2-state simulation.

6. Eliminating some of my X with 2-state data types

The original Verilog language only provided 4-state data types. The only way to achieve the benefits of 2-state simulation was to use proprietary options provided by simulators, as discussed in the previous section. These proprietary 2-state algorithms do not work the same way with each simulator. 2-state simulation modes also make it difficult to mix 2-state simulation in one part of a design and 4-state simulation in other parts of the design.

SystemVerilog improves on the original Verilog language by providing a standard way to handle 2-state simulations. Several SystemVerilog variable types only store 2-state values: **`bit`**, **`byte`**, **`shortint`**, **`int`**, and **`longint`**. SystemVerilog-2012 adds the ability to have user-defined 2-state net types, as well.

Using these 2-state data types has two important advantages of simulator-specific 2-state simulation modes:

- All simulators follow the same semantic rules for what value to use in ambiguous conditions (such as power-up).
- It is easy to mix 2-state and 4-state within a design, which allows engineers to select the appropriate type for each design or verification block.

The uninitialized value of 2-state variables is 0. This can help prevent blocks of design logic from getting stuck in a logic X state at the start of simulation, as discussed in 2.2 earlier in this paper. The clock-divider circuit that was described at the beginning of Section 4 will work fine if the flip-flop storage is modeled as a 2-state type.

Having all variables begin with a logic 0 does not accurately mimic silicon behavior, however, where each bit of each register can power-up to either 0 or 1. When all variables start with a value of 0, only one extreme and unlikely hardware condition is verified. Bening [5] suggests that simulation should begin with random values for all bits in all registers, and that hundreds of simulations with different seed values should be run, in order to ensure that silicon will function correctly at power-up under many different conditions.

The ability to declare nets and variables that use either 2-state or 4-state value sets makes it possible to freely mix 2-state and 4-state within a simulation. Engineers can choose the benefits of 2-state performance in appropriate places within a design or testbench, and choose the benefits of 4-state simulation where greater accuracy is required.

SystemVerilog defines a standard rule for mapping 4-state values to 2-state values. The rule is simple. When a 4-state value is assigned to a 2-state net or variable, any bits that are X or Z are converted to 0. This simplistic rule eliminates X values, but does not accurately mimic silicon behavior where each ambiguous bit might be either a 0 or a 1, rather than always 0.

Your X just might be your best friend!

Section 5 of this paper discussed several hazards with using 2-state simulation modes. All of those hazards also apply to using 2-state data types. X is the simulator's way of saying there is some sort of ambiguity in a design. As much as all engineers hate to see an X show up during simulation, an X indicates there is a potential design problem that needs to be investigated. The following example illustrates how 2-state types can hide a serious design error.

```
module program_counter (       // 2-state types
  input  bit        clock, resetN, loadN,
  input  bit [15:0] new_count,
  output bit [15:0] count
);
  always_ff @(posedge clock or negedge resetN)
    if (!resetN)     count <= 0;
    else if (!loadN) count <= new_count;
    else             count <= count + 1;
endmodule: program_counter

module cpu (                   // 4-state types
  wire        m_clk, m_rstN,
  wire [15:0] next_addr
);
  ...
  program_counter pc (.clock(m_clk),
                      .resetN(m_rstN),
                      .loadN(/* not used */),
                      .new_count(/* not used */),
                      .count(next_addr) );
  ...
endmodule: cpu
```

Example 28: Program counter with unused inputs, 2-state data types

The program counter in this example is loadable, using an active-low `loadN` control. The CPU model has an instance of the program counter, but does not use the loadable `new_count` input or `loadN` control. Since they are not used, these inputs are left unconnected, which is probably an inadvertent design bug! With 2-state data types, however, the unconnected inputs will have a constant value of 0, which means the statement

```
    if (!loadN) count <= new_count;
```

will always evaluate as true, and the program counter will be locked in the load state, rather than incrementing on each clock edge.

In this small example, this bug would be easy to find. Imagine, though, a similar bug in a huge ASIC or FPGA design. Simple mistakes that are hidden by not having a logic X show up in simulation can become very difficult to find. Worse, the symptom of having a logic 0, instead of a logic X, might make a design bug appear to be working at the RTL level, and not show up until gate-level simulations are run. (And what if your team doesn't do gate-level simulations?)

After having a 2-state data type hide a design error or cause bizarre simulation results in a large, complex design, you too might feel, as the author does, that *"I'm still in love with my X!"*

7. BREAKING THE RULES — SIMULATOR-SPECIFIC X-PROPAGATION OPTIONS

The previous sections in this paper have shown that SystemVerilog can sometimes be overly optimistic, and at other times overly pessimistic in how logic X is propagated, and that 2-state simulations and data types can hide design problems by completely eliminating Xs. Can a balance between these two extremes be found by breaking the IEEE 1800 SystemVerilog standard X propagation rules and simulating with a different algorithm?

Some simulators provide proprietary invocation options to begin simulation with random variable values, instead of with X values. Using simulator-specific options can accomplish Bening's [5] recommended approach of randomly initializing all registers using a different seed. Since these options are not part of the SystemVerilog language, however, the capability is not available on every simulator and does not work the same way on simulators that have the feature.

Some SystemVerilog simulators offer a way to reduce X-optimism in RTL simulation by using a more pessimistic, non-standard algorithm. For example, the Synopsys VCS "**-xprop**" [11] simulation option causes VCS to use simulator-specific X propagation rules for **if**...**else** and **case** decision statements and **posedge** or **negedge** edge sensitivity. This non-standard approach tries to find a balance between X-optimism and X-pessimism.

See Evans, Yam and Forward [12] and Greene, Salz and Booth [13] for more information on—and experience with—using proprietary X-propagation rules to change SystemVerilog's X-optimism and X-pessimism behavior.

One concern with proprietary X propagation rules is that their purpose is to ensure that design bugs will propagate downstream from the cause of the problem, so that the bug will be detected instead of hidden. This then requires tracing the cause of an X back through many lines of code, branching statements, and clock cycles to find the original cause of the problem. Though most simulators provide powerful debug tools for tracing back X values, the process can still be tedious and time consuming.

Another concern is the risk of false failures, by making simulation more X-pessimistic. Finding a balance between X-optimism and being overly pessimistic can be good, but, like the **? :** conditional operator, will not always perfectly match silicon behavior (see 4.2). There might still be times when this balance of X-optimism and X-pessimism can result in false failures. At best, these false failures can consume significant project man-hours to determine that there is no actual design problem. Worse — and very possible — these false failures could potentially cause problems with simulation locking up in X states, as described in 2.2.

8. CHANGING THE RULES — A SYSTEMVERILOG ENHANCEMENT WISH LIST

There have been proposals to modify SystemVerilog's X-optimism and X-pessimism rules in some future version of SystemVerilog. If readers of this paper feel these proposed enhancements would be important for their projects, they should put pressure on their EDA vendors to push these enhancements through in the next version of SystemVerilog.

One of the X-optimism issues presented in this paper is that wildcard "don't care" bits in **casex**, **casez** and **case**...**inside** statements mask out all 4 possible 4-state values, causing unknown bits in *case expressions* to be treated as don't care values.

Turpin [1] proposed adding the ability to specify 2-state wildcard "don't care" values using an asterisk (instead of X, Z or ?), as follows:

```
always_comb begin
  case (sel) inside
    3'b1**: y = a;  // matches 100, 101, 110, 111
    3'b00*: y = b;  // matches 000, 001
    3'b01*: y = c;  // matches 010, 011
    default: y = 'x;
  endcase
end
```

Example 29: Proposed case...inside with 2-state don't cares

In normal SystemVerilog X-optimistic semantics, if either of the lower 2 bits of `sel` were X or Z, those bits could potentially be masked out by the 4-state don't cares in the *case items*, causing `y` to be assigned a known value instead of propagating an X. The proposed 2-state don't care bits (represented by an asterisk) would not mask out X or Z values, and result in the **`default`** branch propagating an X whenever there is a problem with the *case expression*.

Cummings [15] proposed adding new procedural blocks that are X-pessimistic instead of X-optimistic. The proposed keywords are **`initialx`**, **`alwaysx`**, **`always_combx`**, **`always_latchx`** and **`always_ffx`**. Cummings proposes that any time a decision control expression or loop control expression evaluates to X or Z, simulation should do three things:

- Assign X values to all variables assigned within the scope of the decision statement or loop.
- Ignore all system tasks and functions within the scope of the decision statement or loop.
- Optionally report a warning or error message that the tested expression evaluated to an X or Z.

An example usage is:

```
always_ffx @(posedge clk or negedge rstN)
  if (!rstN) q <= 0;
  else       q <= d;
```

Example 30: Proposed procedural block with X-pessimism

Under SystemVerilog's normal X-optimistic rules, if `rstN` evaluated as X or Z, then `q` would be set to the value of `d`, hiding the ambiguous reset problem. Using the proposed X-pessimistic rules for **`always_ffx`**, if `rstN` evaluated as X or Z, then `q` would be set to X, propagating the ambiguous reset problem.

The author of this paper does not fully concur with the semantics Cummings has proposed. The author likes the concept of special RTL procedures with more accurate X-propagation behavior, but feels the proposed semantics are overly pessimistic, and could result in causing false X values or X-lockup problems — the same issues noted earlier in this paper regarding excessive X pessimism. The author would prefer to see semantics that are similar to the T-merge algorithm used by the proprietary VCS -xprop simulation option.

9. Detecting and stopping my X at the door

Let's face it, when an X shows up, trouble is sure to follow! Rather than having X problems propagate through countless lines of code, decision branches, and clock cycles, it would be much better to detect an X the moment it occurs. ***Detecting when an X first appears solves the problems of both X-optimism and X-pessimism!***

X-optimism results in X values propagating as 0 or 1 values to downstream logic, potentially hiding design problems. X-pessimism results in all X values propagating to downstream logic, potentially causing simulation problems such as X-lockup, that would not exist in actual silicon. In either case, design problems might not be detected until far down stream in both logic and clock cycles from the original cause of the bug. Engineers must then spend a great deal of valuable engineering time debugging the cause of the problem.

SystemVerilog immediate assertions can be used to detect X values at the point the value occurs, rather than detecting the X value after it has (maybe) propagated downstream to other logic in the design. The way to do this is to use assertions to monitor all input ports of a module, as well as selection control values on conditional operations.

An additional advantage of using assertions to monitor for X values is that assertions can be disabled when X values are expected, such as before and during reset or during a low power shut down mode. Disabling and re-enabling of assertions can be done at any time during simulation, and can be on a global scale, on specific design blocks, or on specific assertions.

The syntax for an immediate assertion is:

> **`assert (`** *expression* **`)`** [*pass_statement*]
> [**`else`** *fail_statement*] **`;`**

An immediate **`assert`** statement is similar to an **`if`** statement, except that both the *pass_statement* and the **`else`** clause are optional.

The pass or fail statements can be any procedural statements, such as printing messages or incrementing counters. Typically, the pass statement is not used, and the fail statement is used to indicate that an X value has been detected, as shown in the following code example for a simple combinational **`if`**...**`else`** statement:

```
always_comb begin
  assert (!$isknown(sel))
  else $error("%m, sel = X");

  if (sel) y = a;
  else     y = b;
end
```

Example 31: if...else with X-trap assertion

This is the same **`if`**...**`else`** example that has presented in previous sections, but with an added assertion to validate the value of `sel` each time it is evaluated.

Without the assertion, this simple **`if`**...**`else`** decision has several potential X hazards, as was discussed in 3.1 and 4.1. Adding an immediate assertion to verify **`if`** conditions is simple to do, and avoids all of these hazards. A problem with the **`if`** condition is detected when and where the problem occurs, rather than hoping that propagating an X will make it visible sometime, somewhere. Assert statements are ignored by synthesis, so no code has to be hidden from synthesis compilers.

The author recommends that **`if`** statements that are conditioned on a module input port have an immediate assertion to validate the **`if`** condition. A text-substitution macro could be defined to simplify using this assertion in many places.

```
`define assert_condition (cond) \
  assert (^cond === 1'bx) \
  else $error("%m, ifcond = X")

always_comb begin
  `assert_condition(sel)
  if (sel) y = a;
  else     y = b;
end

always_comb begin
  `assert_condition({a,b,c,d})
  `assert_condition(sel)
  case (sel)
    2'b00 : out = a;
    2'b01 : out = b;
    2'b01 : out = c;
    2'b01 : out = d;
  endcase
end
```

Example 32: Using an X-trap assertion macro

SystemVerilog assertions are ignored by synthesis compilers, and therefore can be placed directly in RTL code without having to hide them from synthesis using conditional compilation or pragmas. It is also possible to place the assertions in a separate file and bind them to the design module using SystemVerilog's binding mechanism.

10. MINIMIZING PROBLEMS WITH MY X

This section presents a few coding guidelines that help to appropriately use and benefit from SystemVerilog's X-optimism and X-pessimism, and minimize the potential hazards associated with hiding or propagating an X.

10.1. 2-state versus 4-state guidelines

Your X can be your best friend. X values indicate that there is some sort of ambiguity in the design. Eliminating X values using 2-state data types does not eliminate the design ambiguity. Sutherland HDL recommends using 4-state data types in all places, with two exceptions:

- The iterator variable in **`for`**-loops is declared as an **`int`** 2-state variable.
- Verification stimulus variables that will (or might) have randomly generated values are declared as 2-state types.

This coding guideline uses 2-state types only for variables that will never be built in silicon, and therefore do not need to reflect an ambiguous condition that might exist in silicon.

There is one other place where 2-state types might be appropriate, which is the storage of large memory arrays. Using 2-state types for large RAM models can substantially reduce the virtual memory needed to simulate the memory. This savings comes at a risk, however. Should the design fail to correctly write or read from a memory location, there will be no X values to indicate there was a problem. To help minimize that risk, it is simple to model RAM storage, so that it can be configured to simulate as either 2-state storage (using the **`bit`** type) or 4-state storage (using the **`logic`** type).

10.2. Register initialization guidelines

Section 2.2 discussed the problems associated with design variables, especially those used to model hardware registers, beginning simulation with X values. Section 5 discussed using proprietary simulation options to initialize register variables with random values. If that feature is available, it should be used!

Another way to randomly initialize registers is using the UVM Register Abstraction Layer (RAL). UVM is a standard, and is well supported in major SystemVerilog simulators. A UVM testbench and RAL are not trivial to set up, but can provide a consistent way to randomly initialize registers. The advantage of using UVM to initialize registers is that it will work with all major simulators.

10.3. X-assignment guidelines

Using X assignments to make **`if`**...**`else`** and **`case`** statements more pessimistic should not be used. They add overhead to simulation, and can simulate differently than the logic that is generated from synthesis. The pessimistic X propagation can lead to false failures that can take time to debug and determine that there not be a problem in actual silicon. In lieu of using pessimistic coding styles to propagate X values, problems should be trapped at the select condition, as shown in Section 9, and discussed in the following guideline.

10.4. Trapping X guidelines

All RTL models intended for synthesis should have SystemVerilog assertions detect X values on **`if`**...**`else`** and **`case`** select conditions. Other critical signals can also have X-detect assertions on them. Design engineers should be responsible for adding these assertions. Section 9 showed how easy it is to add X-detecting assertions.

11. CONCLUSIONS

This paper has discussed the benefits and hazards of X values in simulation. Sometimes SystemVerilog is optimistic about how X values affect design functionality, and sometimes SystemVerilog is pessimistic.

X-optimism has been defined in this paper as any time simulation converts an X value on the input to an operation or logic gate into a 0 or 1 on the output. Some key points that have been discussed include:

- X-optimism can accurately represent real silicon behavior when an ambiguous condition occurs. For example, if one input to an AND gate is uncertain, but the other input is 0, the output of the gate will be 0. SystemVerilog X-optimistic AND operator and AND primitive behave the same way.
- X-optimism is essential for some simulation conditions, such as the synchronous reset circuit shown in Section 3.
- SystemVerilog can be overly optimistic, meaning an X propagates as a 0 or 1 in simulation when actual silicon is still ambiguous. Over optimism can lead to only one of the possible silicon values being verified.
- In all circumstances, X-optimism has the risk of hiding design bugs. A ambiguous condition that causes an X deep in the design might not propagate as an X to a point in the design that is being observed by verification. The value that does propagate might appear to be a good value.

X-pessimism is defined in this paper as any time simulation passes an X on an input through to the output. X-pessimism can be desirable or undesirable.

- X-pessimism will not hide design bugs the way X-optimism might. An ambiguous condition deep within a design will propagate as an X value to points that verification is observing.
- X-pessimism can lead to false failures, where actual silicon will function correctly, such as if one input to an AND gate is an X, but the other input is 0. A false X might need to be traced back through many levels of logic and clock cycles before determining that there is not an actual problem.
- X-pessimism can lead to simulation locking up with X values, where actual simulation will function correctly, even if the logic values in silicon are ambiguous. The clock divider shown in Section 4 is an example of this.

It might be tempting to use 2-state data types or 2-state simulation modes to eliminate the hazards of an X. Although there are some advantages to 2-state simulation, those advantages do not outweigh the benefits of 4-state simulation. 2-state simulation will hide all design ambiguities, and often not simulate with the same values that actual silicon would have. 2-state data types should only be used for generating random stimulus values. Design code should use 4-state types.

The best way to handle X problems is to detect the X as close to its original source as possible. This paper has shown how SystemVerilog assertions can be used to easily detect and isolate design bugs that result in an X. With early detection, it is not necessary to rely on X propagation to detect design problems.

All engineers should be in love with their X! X values indicate that there might be some ambiguity in an actual silicon implementation of intended functionality.

12. ABOUT THE AUTHOR

Stuart Sutherland is a well-known Verilog and SystemVerilog expert, with more than 24 years of experience using these languages for design and verification. His company, Sutherland HDL, specializes in training engineers to become true wizards using SystemVerilog. Stuart is active in the IEEE SystemVerilog standards process, and has been a technical editor for every version of the IEEE Verilog and SystemVerilog Language Reference Manuals since the IEEE standards work began in 1993. Prior to founding Sutherland HDL, Mr. Sutherland worked as an engineer on high-speed graphics systems used in military flight simulators. In 1988, he became a corporate applications engineer for Gateway Design Automation, the founding company of Verilog, and has been deeply involved in the use of Verilog and SystemVerilog ever since. Mr. Sutherland has authored several books and conference papers on Verilog and SystemVerilog. He holds a Bachelors Degree in Computer Science with an emphasis in Electronic Engineering Technology and a Masters Degree in Education with an emphasis on eLearning. You can contact Mr. Sutherland at stuart@sutherland-hdl.com.

13. ACKNOWLEDGMENTS

The author appreciates the contributions Don Mills and Shalom Bresticker have made to this paper. Don provided several of the examples and coding recommendations in this paper, and provided valuable suggestions regarding the paper content. Shalom made an in-depth technical review of the paper draft and provided detailed comments on how to improve the content of the paper.

The author also expresses gratitude to his one-and-only wife of more than 30 years (she will never be an “X”), who, despite the title of the paper, painstakingly reviewed the paper for grammar, punctuation, and sentence structure.

14. REFERENCES

[1] Turpin, “The dangers of living with an X,” Synopsys Users Group Conference (SNUG) Boston, 2003.

[2] Mills, “Being assertive with your X (SystemVerilog assertions for dummies),” Synopsys Users Group Conference (SNUG) San Jose, 2004.

[3] “P1800-2012/D6 Draft Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language (re-ballot draft)”, IEEE, Pascataway, New Jersey. Copyright 2012. ISBN: (not yet assigned).

[4] Merriam-Webster online dictionary, http://www.merriam-webster.com/, accessed 11/20/2012.

[5] Bening, “A two-state methodology for RTL logic simulation,” Design Automation Conference (DAC) 1999.

[6] Cummings and Bening, “SystemVerilog 2-state simulation performance and verification advantages,” Synopsys Users Group Conference (SNUG) Boston, 2004.

[7] Piper and Vimjam, “X-propagation woes: masking bugs at RTL and unnecessary debug at the netlist,” Design and Verification Conference (DVcon) 2012.

[8] Weber and Pecor, “All My X values Come From Texas…Not!,” Synopsys Users Group Conference (SNUG) Boston, 2004.

[9] Turpin, “Solving Verilog X-issues by sequentially comparing a design with itself,” Synopsys Users Group Conference (SNUG) Boston, 2005.

[10]Chou, Chang and Kuo, “Handling don’t-care conditions in high-level synthesis and application for reducing initialized registers,” Design Automation Conference (DAC) 2009.

[11]Greene, “Getting X Propagation Under Control”, a tutorial presented by Synopsys, Synopsys Users Group Conference (SNUG) San Jose, 2012.

[12]Evans, Yam and Forward, “X-Propagation: An Alternative to Gate Level Simulation”, Synopsys Users Group Conference (SNUG) San Jose, 2012.

[13]Greene, Salz and Booth, “X-Optimism Elimination during RTL Verification”, Synopsys Users Group Conference (SNUG) San Jose, 2012.

[14]Browy and K. Chang, “SimXACT delivers precise gate-level simulation accuracy when unknowns exist,” White paper, http://www.avery-design.com/files/docs/SimXACT_WP.pdf, accessed 11/12/2012.

[15]Cummings, “SystemVerilog 2012 new proposals for design engineers,” presentation at SystemVerilog Standard Working Group meeting, 2010, http://www.eda.org/sv-ieee1800/Meetings/2010/February/Presentations/Cliff%20Cummings%20Presentation.pdf, accessed 11/12/2012.

[16]Mills, “Yet another latch and gotchas paper” Synopsys Users Group Conference (SNUG) San Jose, 2012.