SystemVerilog 2-State Simulation Performance and Verification Advantages

Clifford E. Cummings

Sunburst Design, Inc.

cliffc@sunburst-design.com

Lionel Bening

Hewlett-Packard

lionel.bening@hp.com

ABSTRACT

VCS has had a proprietary 2-state simulation mode for years. SystemVerilog adds 2-state data types that will allow engineers to take advantage of a standard 2-state simulation mode using any compliant SystemVerilog simulator.

This paper will summarize the verification advantages related to 2-state simulation and address 4-state simulation problems that are alleviated by performing 2-state simulation.

Are there any simulation performance advantages to using 2-state simulation? How do other Verilog data types impact 2-state simulation? Do 2-state vector simulations show more improvement over 2-state scalar models? These questions will be quantified and addressed in this paper.

This paper will also detail a new SystemVerilog approach to 2-state simulation and will discuss potential Verilog language modifications that could further enhance 2-state simulation.

Table of Contents

Table of Figures

Table of Examples

Table of Tables

1 Introduction

This paper will show that there are two distinct reasons why engineers should consider 2-state simulation:
1. Increased simulation speed.
2. Enhanced design-problem identification environment.

In this section, a brief history of 2-state simulation is presented. In later sections, the VCS 2-state simulation capabilities, the 2-state simulation methodology used by HP large-scale server design projects and the SystemVerilog 2-state simulation techniques are all described.

There is a general industry misconception that 2-state simulation is much faster than 4-state simulation. This paper will show cases where this is and is not true.

Unfortunately, there are deficiencies related to doing VCS **`+2state`** simulation. Some deficiencies are related to modifying the RTL code to support a full 2-state simulation environment, and other deficiencies are related to important missing 2-state capabilities in the VCS **`+2state`** mode.

Some of the 2-state simulation deficiencies can be addressed by new SystemVerilog capabilities and methodologies, but unfortunately, not even the new SystemVerilog capabilities fully address the requirements of an advanced 2-state simulation environment.

HP large-scale server design projects have assembled an advanced 2-state simulation methodology. This methodology combines a vendor linting tool with 2-state RTL linting rules, and a patented reproducible random reset initialization technology that is currently used with 4-state simulators to replicate the design flaw removal capabilities of a 2-state simulation environment. The methodology is described and a couple of the patent issues are addressed in this paper. Whenever this paper mentions “the 2-state methodology” or “2-state discipline,” it is with reference to the methodology described in this paragraph.

So why even write this paper if there are so many problems related to 2-state simulations and methodologies? Because there is great potential to improve 2-state tools and capabilities once the problems are understood.

1.1 2-State Simulation - History

Although both RTL and gate-level simulation began as thought exercises in the 1950’s, engineers began applying gate-level simulations in the form of computer programs called simulators to their designs in the 1960’s. RTL languages accompanied by simulators originated in the 1960’s and began widening application in industry by the 1970’s. In those days, each simulator had its own language, and the RTL and gate-level languages were generally separated as well.

The first gate-level simulation programs used 2-state modeling, but by the later 1960's, many gate-level simulators added an **X**-state [9]. Many RTL simulators continued using 2-state in a more abstract simulation model, and applied what later came to be known as cycle-based techniques for fast simulation.

As the VHSIC and Verilog HDLs emerged as open and standard in the early 1990's, design teams in several companies mapped their RTL 2-state cycle-based simulation language techniques to these open and standard languages. Their application of open standard HDLs gave these teams access to the broadening EDA vendor tool sets supporting these languages, while retaining the advantages of the RTL disciplines supported by their earlier use of their in-house proprietary languages and simulators.

In spite of repeated project success in applying the RTL disciplined design principles to arrive at good silicon in separate design labs, efforts to spread the word about the disciplines have been difficult, even within corporations. There are multiple reasons for the difficulties:

- Many designers have had successful experiences using an **X** in gate-level simulation and some RTL simulations and have not had experience with bad silicon resulting from **X**-optimism.
- 4-state simulation is still the most widely taught method in Verilog training classes and training in RTL 2-state disciplined application is largely ignored.
- Understanding of successful 2-state simulation environments is not widely known in the general design and verification user community.
- Reproducible randomized 2-state initialization technology is patented and not readily available at any company without licensing the patented technology from HP.

The Bening DAC paper [10] and the Bening and Foster book [12] are efforts to enlighten design projects about the high value of RTL 2-state disciplines. The papers by Cummings [1], Mills and Cummings [6], and Turpin [13] illustrate many of the pitfalls that come from a lack of a 2-state discipline.

The authors do not 100% agree on all of the proposed Verilog and SystemVerilog 2-state coding and simulation guidelines, but where the authors disagree, both viewpoints are expounded and the readers are left to choose the appropriate guidelines for their corporate design and verification environments.

2 RTL 2-State Simulation Methods

The methods described in this section were derived from more than ten years of experience on projects that were 100% committed to use of RTL **0**/**1** randomization in place of **X**. The designers applied this RTL **0**/**1** randomization at anytime and anywhere that designers earlier/elsewhere would think that **X**'s might originate. This includes power-on, primary inputs, and special function module types, such as synchronizers, tri-state/differential receivers, etc.

To those who have not had this 2-state project experience, it might seem that a randomized 2-state discipline would be onerous and difficult to live with. Experience by HP large-scale server design projects has shown that the randomized 2-state discipline sells itself to new designers within an hour of their first using it. After that, these designers never want to go back to RTL debugging designs without the 100% commitment to use of RTL **`0`**/**`1`** randomization in place of **`X`**.

To build a successful 2-state simulation environment, "some assembly is required." Tools that the HP large-scale server design projects used to build a successful 2-state simulation environment include:

- A lint rule checker, with a "no-exception" setting for 2-state-related rules
- A Boolean equivalence checking tool, to verify that the Verilog upgrades to 2-state identified by the lint checks are functionally equivalent to the original Verilog.

2.1 2-State Power-On modeling

One of the keys to the 2-state methodology is an RTL reproducible **`0`**/**`1`** random technology that projects apply to improve the likelihood of power-on reset success. It should be noted that the 2-state techniques described in this paper do not compensate for bad reset-logic design or bad multi-clock design techniques and engineers still need to follow good reset and multi-clock design techniques in addition to the 2-state techniques described in this paper[2][3].

At first glance, the $2^{200,000}$ or more state bit combinations in the chips designed today may appear to overwhelm the few millions (2^{20}) of simulation random start-up states that are typically simulated on HP large-scale server design projects; however, project experience with simulations using random startup states shows that this class of design errors is discovered in the earliest simulations.

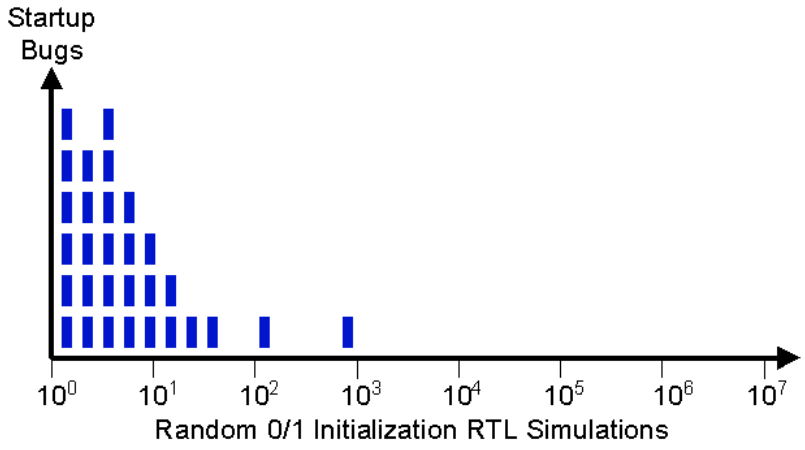


Figure 1 - Random initialization startup bug experience

Figure 1 illustrates how most of this class of problems are discovered within the first hundred simulations. Subsequent design errors of this type are discovered in the following hundreds of simulations, and then do not show up at all in the remaining millions of simulations.

Note that in Figure 1, the horizontal axis counts simulations on a log scale. If the axis were linear, all of the design errors would be squashed against the vertical axis on the left.

Simulated logic designs run tens of millions of times slower than actual hardware. A million hours of logic simulation runs represent about five seconds of actual hardware operation; however, simulation models are far superior to actual logic hardware for verifying power-on reset functionality.

HP large-scale server design projects run millions of simulations on a server farm consisting of hundreds of CPU's to test:

- using millions of start-up states
- applying reproducible [11] and high quality random distribution.

The reproducible simulation technology supported by the Bening and Chaney patent [11] extends seed-based duplication of startup states through a design change (except for the state bits added or deleted by the change). After designers change logic to fix a problem that were discovered in one of the simulations based on time-of-day seeds, they can take a seed associated with the problem, and repeat the same state conditions that caused the failure before their fix.

Using actual hardware for power-on reset verification is typically not as efficient simulated power-on verification, because:

- power-on tests with real hardware can typically only be conducted thousands of times instead of millions of simulated power-on resets.
- repeatability of power-on failures either on the same chip or from chip-to-chip, is problematic.
- power-on testing with prototypes typically uses a small set of devices with similar process characteristics, so there likely will be poor-quality random distribution. Simulations will randomly initialize state variables, which also simulates manufactured devices with random process variations.

2.2 Power-On 2-State Randomization using $set_values

To simulate the random **`0`**/**`1`** conditions after power-on in 2-state RTL modeling, projects set up the simulation environment to initialize all storage elements to random **`0`**/**`1`** values, including memory arrays, flip-flops, and latches.

One approach to randomizing storage elements is to adapt a PLI system task called **$set_values**. Source code for **$set_values** system task and its supporting sub-functions has been part of the PLI [7] (and now VPI [8]) example set delivered with vendor simulation products going back to Verilog-XL.

Engineers using **$set_values** specify the scope and the type of initialization that they want, "random", or a specified value string. When called at power-on time in a simulation, the **$set_values** user task traverses the design from the specified scope and downward.

```
$set_values("random", top0_i0.CORE.gclkg)

$set_values("0", top0_i0.CORE.gclkg)
```

In each "visit" to a memory array, **reg**, or UDP, **$set_values** pokes in a call-order-sequence-based random, or specified value.

Although **$set_values** is excellent as a example to illustrate PLI usage for a useful purpose, projects using **$set_values** need to be aware of its limitations and understand the adaptations required for its use in production.

2.2.1 $set_values Races

One of the limitations of **$set_values** is that it pokes values into all elements at one time cycle. The events resulting from these pokes into interdependent logic may race with one another and propagate into other logic. With Verilog simulators, the outcome from simultaneous event ordering is unstable. Examples of logic races resulting from the initializing pokes include:

- arrays and the address registers that reference the arrays.

- clock divider, clock enable flip-flops, and the registers controlled by these flip-flops. The **$set_values** **0**/**1** initialization of a read address may read out and propagate **X**'s from the array before the array has completed its "at the same time" initialization.

The outcome of these races is that **X**'s may propagate and persist, even after the **$set_values** system task has poked its **0**/**1**'s into all of the logic.

To check whether **X**'s persist after a **$set_values** task call, some project teams set up their test environment to wait a minimum time step and call a **$check_values** function. The **$check_values** function traverses all of the **reg**s and UDPs, and instead of poking, queries to see whether any **X**'s are left after the **$set_values** traversal, and pokes in **0**/**1**'s to replace the **X** residue. The initialization repeats the **$set_values**/**$check_values** cycle until the **$check_values** reports no more **X**'s. Based on experience, project reports have shown that three cycles have been enough to eliminate all of the **X**'s.

2.2.2 $set_values and VCS +2state

As written, **$set_values** is oblivious to whether it is poking into storage elements or combinational logic **reg**s. If everything starts at an **X**, its **0**/**1** pokes will produce events. In combinational logic, these events will propagate, and cause the combinational logic terms to "relax" into self-consistent **0**/**1** values within each logic cone.

If everything starts at **0** as in **+2state**, and we poke in **0**/**1**'s, combinational logic within a cone will have **0**/**1**'s with inconsistent values, and cause tests to fail at the start of simulation.

This can be overcome by extending **$set_values** to recognize storage elements, for example using an **_r** flip-flop suffix naming convention. In this way, **$set_values** focuses its **0**/**1** poking on storage elements, and leaves combinational logic **reg**s alone.

2.2.3 $set_values Repeatability

To a large degree, with some simple extensions, **$set_values** can support seed-based repeatability. To do this, all simulation runs log the time-of-day based seed. If a test shows an error, an engineer turns on debugging and repeats the test by entering the seed associated with the failing test. This seed supercedes the time-of-day based seed that **$set_values** ordinarily uses, with largely the same **0**/**1** startup values poked into the test that failed.

```
...

simv +test=early_tag
***  +rseed=61248222, based on time-of-day
***  Test 'early_tag' PASSED

simv +test=early_tag
***  +rseed=97249740, based on time-of-day
***  Test 'early_tag' PASSED

simv +test=early_tag
***  +rseed=61300869, based on time-of-day
***  Test 'early_tag' FAILED
```

(a) Test passes then fails in regression runs, depending on seed

```
simv +test=early_tag +rseed=61300869
***  +rseed=61300869, based on command line
***  Test 'early_tag.run' FAILED
```

(b) Engineer repeats failed run, entering the failing test and seed

Example 1 - Repeating failed regression test run for debugging

With inherent races associated with state independencies, **$set_values** cannot be guaranteed to reproduce the exact same state in the move from the regression simulation run that detected an error, and a debug simulation run with debugging turned on.

By extending **$set_values** and its usage to recognize flip-flops and some of the special clock controlling flip-flop variants, a test environment can call **$set_values** at separate times and have it successively initialize classes of storage elements in a manner that eliminates races in interdependent logic interactions, as shown in Example 2.

```
initial begin
  #0.004;
  $set_values_mem("random", top );
end

initial begin
  #0.005;
  $set_values_ff("random", top );
end

initial begin
  #0.006;
  $set_values_latch("random", top );
end

initial begin
  #0.007;
  $set_values_ffcken("random", top );
end
```

Example 2 - Separating $set_values times to eliminate races

2.3 Library-Based Power-On 2-State Randomization

To overcome the entanglements associated with power-on 2-state randomizations using **$set_values** (and for other reasons), many engineers believe that library-based instantiated storage elements (advocated by Bening and Foster [12] and others) provides a simple method for encapsulating random initialization.

These library-based modules encapsulate all of the detail about the race-free timing of initialization, and capture the nature of the storage element where the designers instantiate the storage element modules. Example 3 shows initialization timings for various storage element types

```
module dff (q, d, clk);
  parameter W=1;
  output [W-1:0] q;
  input  [W-1:0] d;
  input          clk;
  reg    [W-1:0] q;

  always @(posedge clk)
    q <= d;
```

```
 `ifdef INITSTATE
    initial
      `TIMEINIT $InitReg(q);
 `endif
endmodule
```

(a) RTL D flip-flop module dff

```
...
  `ifdef INITSTATE
    initial begin
      `TIMEINITMEM
      // InitReg is smart enough to init the whole
      // array with only 1 call
      $InitReg(dataarray[0]);
    end

    initial begin
      `TIMEINIT
      $InitReg(waddr_r);
      $InitReg(raddr_r);
      $InitReg(wdata_r);
    end
  `endif
endmodule
```

(b) Memory array address/data register initialization

```
module dff_cken (q, d, clk);
  < dff model as in (a)>
  `ifdef INITSTATE
    initial
      `TIMEINITCKEN $InitReg(q);
  `endif
endmodule
```

(c) RTL D flip-flop module dff_cken clock control register

Example 3 - RTL Library modules with encapsulated initialization timings

By having random initialization built into library models, the library-based method eliminates the multiple traversals of the design associated with **`$set_values`**, where it has to locate memory arrays, ordinary flip-flops, latches, and clock-control flip-flops.

2.4 External inputs

Projects adhering to a 2-state simulation discipline in their chip design have to deal with the external world, where testbenches and other chip designs in the system model put **`Z`**'s or **`X`**'s on the inputs of a design, or have basic bugs in their signal connections, states, or timing.

2.4.1 Tri-state Bus Receivers.

Where the input may be tri-state **`Z`** (or **`X`**), designs simulated with a 2-state methodology need to **`0`**/**`1`** randomize the receiver output.

2.4.2 Differential Receivers.

As with tri-state receivers with **`Z`** states at their inputs, differential receivers with non-differential values (or **`X`**'s) need to **`0`**/**`1`** randomize their outputs.

2.4.3 Testbenches.

In setting up the test environment, verification engineers may have loose ends in the timing or completeness of the test input. Common testbench oversights include applying input stimulus prior to the hardware power-on initialization, and starting the clock before setting **`0`**/**`1`** data values on the inputs.

A sign of trouble in projects using power-on **`0`**/**`1`** randomization is when testbenches apply **`0`**/**`1`** inputs at time zero. In addition to likely time zero races that time zero inputs create on their own, there is loss of power-on test integrity where the time zero values that propagate through edge-triggered logic either:

- get wiped out by power-on **`0`**/**`1`** randomization pokes, or
- supercede power-on **`0`**/**`1`** randomization pokes

2.4.4 HDL Intellectual Property (IP).

Projects nowadays frequently reuse design blocks, either from prior projects within their own company, or as IP from external sources. IP may be from design teams that did not follow **`0`**/**`1`** disciplines. It may produce **`X`**'s at its outputs in a manner that conflicts with the **`0`**/**`1`** protocols at the inputs to the designs adhering to a 2-state design discipline.

One way that projects have dealt with this is to add buffers that produce randomized **`0`**/**`1`** outputs whenever there is a **`X`** at their inputs that receive signals from the IP.

Another way projects have used is to upgrade the IP to use the 2-state discipline. To upgrade the Verilog IP, the engineers first apply lint rules that diagnose departures from a 2-state convention, and then edit the IP Verilog as suggested by the lint rule diagnostic messages. The edits may include:

- Change implicit flip-flops and memories to instantiated library modules, where library modules include **`$InitReg()`** random initialization calls.
- Convert **`case`**/**`casex`**/**`casez`** statements to fully specified **`case`** or **`casex`**.

- Eliminate designer logic **X** tests/assignments.

- Add assertion **X**-tests.

Projects using the upgrade path apply lint rules to ensure that all of the IP HDL 2-state adherence is 100% complete, and then apply Boolean equivalence to prove that the upgrade preserves the original IP function.

2.5 Disciplined modeling

For more than a decade, the HP designs produced by large-scale server projects following a 100% 2-state discipline have shown full awareness of the warnings about the Evil Twins [1], the dangers of living with an **X** [13], and more. With this full awareness, designers completely specify the Boolean **0**/**1** output for all **0**/**1** inputs, even "don't care" input conditions. This practice ensures that the behavior in the power-on state prior to reset maps to realities of the reset process.

These projects (this is not an exaggeration) deliberately avoid bringing in consultants to do Verilog training for their new designers. There is well-grounded concern that external consultants will teach Verilog without a commitment to the 2-state **0**/**1** discipline, and talk about generating **X**'s, and co-mingling **X** trapping with state machine and mux **casez** statements.

2.5.1 Use casex or casez?

HP large-scale server design projects committed to the 2-state discipline use **casex** instead of **casez** . In response to **0**/**1** inputs, **casex** and **casez** are Boolean equivalent. Bening has pointed out that the **casez** statement allows designers to craft special responses to **X** inputs, which both authors agree is both counterproductive and a poor coding style. Cummings has noted that this same argument applies to plain old **case** statements.

Both authors agree that testing for **X**'s in the **case** items is absurd and error prone. The solution to this problem seems to be reasonably simple, linting tools should report errors if **X**'s are found in ANY **case** items (**case**/**casez**/**casex**).

Cummings knows of severe design problems that have been masked by **casex** during 4-state simulation, so he continues to insist that recommending the use of **casex** to be a very dangerous practice in 4-state simulation, even if its use is safe in 2-state simulation [6].

2.5.2 Make X-assignments or avoid X-assignments?

The HP 2-state discipline prohibits the use of **X**-assignments altogether, even in state machine designs. The 2-state discipline dictates that where speed is important, designers apply the one-hot FSM coding technique shown on page 142 of Bening's book [12] to overcome potential synthesis inefficiencies that could occur due to the lack of "don't-care" **X**-assignments for

unused FSM states. Bening notes that where speed is not as important, “gates are cheaper than bugs.”

Using the 2-state discipline, Engineers can apply lint checks to ensure that designers fully specify their **`case`**/**`casex`**s. There are no “don’t care” assignments in the 2-state Verilog code. The 2-state discipline designers use other ways to generate optimum gates where needed, and where not needed, the overriding assumption is that gates are cheaper than bugs hiding in “you should have cared” branches.

Bening points out that just as stray **`X`**’s can wipe out 2-state assumptions, stray **`X`**-assignments can wander into **`if-else`**s or **`case`**s with **`case-default`**s. Unlike 2-state checking through linting and formal checks, there is no help for designers to check the correctness or completeness of their **`X`**-crafting, or where it goes. This is a fair point that Cummings acknowledges that he will have to further consider when formulating **`X`**-assignment guidelines and prohibitions.

Although the 2-state discipline prohibits **`X`**-assignments, Cummings has found the practice of making **`X`**-assignments to be very useful, especially in Finite State Machine (FSM) design. Consider the code fragment in Example 4:

```
always @* begin
  next = 3'bx;
  case (state)
    IDLE: ...
    READ: ...
    ...
    default: begin
               next = 3'bx;
               <outputs> = 'bx;
             end
  encase
end
```

Example 4 - Commingled state machine and X assignment

In this example, Cummings claims that the **`X`**-assignments serve two purposes:

(1) If there is a missing a **`next`**-assignment inside of the **`case`** statement, the error will be obvious and easily trapped. The **`X`**-assignment basically acts like an assertion, “I assert that if I am missing any state assignments, the FSM behavior will be unknown.” When this type of bug shows up in training labs, it immediately points to the problem and is among the easiest to debug. If the **`X`**-assignment is omitted, frequently another valid (but wrong) **`state`** value is assigned and it can be difficult and time-consuming to find the bug.

(2) **`X`**-assignments are treated as “don’t cares” by the synthesis tool, so better synthesis results are usually achieved making **`X`**-assignments (again, Bening has a coding style in his book that may be equally efficient, but benchmarking between the coding styles was not performed for this paper).

If an engineer adds **`next = state`** or **`next = IDLE`** at the top of the combinational **`always`** block, the state machine will always be in a valid state, the bug will not be detected by a lint tool, and it may be many clock cycles before the problem manifests itself. Cummings claims that making **`X`**-assignments is like an immediate assertion. As soon as a missing state assignment occurs, the design will suffer an immediate catastrophic failure that is easily identified in a waveform display as the **`next`** variable takes on all **`X`**'s (the **`next`** variable in the waveform display "bleeds red").

The concept behind **`X`**-assignments is that the **`X`**-assignment should always be replaced by a valid update-assignment within the same time-step and that if the valid update-assignment is missing, the **`X`**-assignment sticks out like a red flag in a 4-state simulation. A true 2-state simulator would convert all **`X`**-assignments to **`0`**'s, **`1`**'s or randoms, so they would likely cause similar failures in 2-state simulations. In the absence of an **`X`**-assignment, engineers often rely on the last assigned value to be correct, which may not be a wise assumption. If a old-valid assignment strays into other logic, it may be multiple clock cycles before the bug causes an error and a waveform display will not immediately reveal the point of initial failure. Cummings claims this type of bug can be time consuming to trace.

Cummings has had lots of experience in training classes with bugs that are easily found by using **`X`**-assignments. For FSM designs, missing states are very quickly identified and easily debugged if an initial **`X`**-assignment is made. All other **`next`**-assignment errors are typically much more time-consuming to find during execution of labs.

Bening notes that Cummings' use of **`0`**/**`1`**-randomized **`X`**-assignments is an interesting technology, which may be useful after some more study. Bening's first thought on this concept is that this coding style puts a bigger burden on the "random state space" to be explored by the notoriously slow speed of logic simulators.

As a side note, recent experimentation conducted by Cummings with 2003 and 2004 versions of Synopsys' Design Compiler seem to show the best synthesis optimizations are achieved only after adding **`case`** default **`X`**-assignments. This did not seem to be true with older versions of Synopsys DC and more experimentation is required to verify this observation.

2.5.3 Separation of Concerns

Projects applying the 2-state discipline separate their 2-state functional correctness verification concerns from stray **`X`** concerns.

Where engineers have concerns about stray **`X`**'s, an **`assert_no_X`** (or a **`$check_values`** sweep) might be in order. The **`casex`** addresses the Boolean function. The 2-state methodology linting rules out use of the **`casez`** and whatever **`X`**/**`Z`**-trapping baggage might have accompanied it. Cummings believes a good linting tool would report errors whenever a **`case`**-item tries to test for **`X`**'s and thereby would also identify the poor **`X`**/**`Z`**-trapping coding style.

Designers on projects applying the 2-state discipline quickly come to believe in it, and apply their awareness of its value in their Verilog designs. For example, it is a widely used practice to propagate an **X** during potentially metastable periods in synchronizers. Applying the 2-state discipline, designers propagate **0**/**1** random values instead during the synchronizer potentially metastable period.

3 2-State Simulation

3.1 Synopsys VCS +2state

The Synopsys VCS **+2state** simulation mode generally converts **Z**'s to **0** and **X**'s to **1**. One important exception is at time-0 when uninitialized variables that assume an **X**-value in 4-state simulation, actually are set to **0** in **+2state** simulations [18].

VCS allows a user to specify that certain variables will always be 4-state variables by including a **/* 4state */** pragma in the variable declaration. For example:

```
reg /* 4state */ [7:0] q;
```

In this example, the **q**-variable will always be a 4-state variable.

VCS also has a configuration file capability to indicate that entire files should be treated as either 4-state or 2-state simulation blocks (see the VCS User Guide for more details) [18].

VCS **+2state** simulation mode also automatically converts many nets and variables to 4-state types in order to reduce potential simulation mismatches between 4-state and 2-state simulations.

A couple of the more notable conversions include the use of a **case** (**casex**/**casez**) statement that does comparisons to **X** and **Z** values. For example:

```
always @*
  case (a)
    1'b0: y = 2'b00;
    1'b1: y = 2'b01;
    1'bz: y = 2'b10;
    1'bx: y = 2'b11;
  endcase
```

Example 5 - Bad coding style that causes automatic conversion to 4-state variables

In this example, the **a** variable is converted to a 4-state variable to ensure that 4-state and 2-state simulations will yield the same assignment to the **y**-variable. As noted earlier both authors agree that testing **X**'s and **Z**'s in **case** items is a terrible coding style, so this conversion should never occur in a well-coded model.

Similarly, the case-equality (**===**) and case-inequality (**!==**) Verilog comparison operators may force a variable to become a 4-state variable to preserve similar 2-state and 4-state simulation results.

While some block-level simulations on some projects have successfully applied the VCS **+2state** for simulations, other projects doing chip and system-level simulations that included tri-state buses found that the need to accurately specify all of the **/* 4value */** signal declarations was too much extra work. Any required **/* 4value */** declaration that a designer accidentally omitted required another debugging cycle.

As noted earlier (see section 2.2.2), projects that applied the oblivious **$set_values** that poked **0**/**1**'s into sequential and combinational variables (typically, Verilog **reg**-variables) found that using the **+2state** caused their simulations to fail. The problem is that **$set_values** will try to assign random values to combinational **reg**-variables (bad) and try to assign random values to clocked **reg**-variables (good). As a result, combinational paths have inconsistent values.

3.2 Modeling 2-State HiZ?

Trying to model **HiZ** nets with 2-state representation does not always work. Why?

At first glance the problem seems to be simple, anytime a net goes to **HiZ**, model it as a **0** or a **1**, after all, a real logic gate input will interpret the **HiZ** input as either a **0** or **1**; although, different inputs on different gates might interpret the **HiZ** differently with some input thresholds recognizing the **HiZ** as a logic **1** and other input thresholds interpreting the logic input as a **0**. Still, it seems like modeling a net at **HiZ** as either a **1** or **0** would be reasonable.

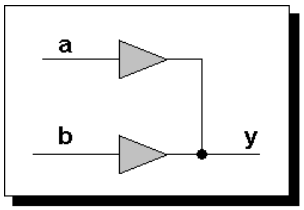
The problem with **HiZ** nets is that almost 100% of nets that allow **HiZ** drivers have two or more drivers, otherwise there would be no need for driving **HiZ**. When there are multiple drivers, the design objective is to ensure that only one driver is enabled at any time, and in most cases the design objective is to ensure that one driver IS enabled at all times, or to connect "weak keepers" or "bus sustainers (busus)" devices to the net to make sure the net never floats to a **HiZ** state (**HiZ** is very bad for CMOS devices because CMOS devices consume the most power during switching and **HiZ** is in the switching voltage range, which will cause the greatest power dissipation).

If the problem were as simple as looking at the resolved **HiZ** value and changing it to a **1** or **0**, then **HiZ** 2-state net modeling would be practical, but the reality is that each **HiZ** contributor would be changed to **0** or **1** and then be combined with an enabled and valid **0** or **1** to generate potential **X**-values which are also converted to **0**'s or **1**'s.

VCS 2-state mode converts all **Z**'s to **0** and all **X**'s to **1** (except at time-0 when even uninitialized **X**-variables are converted to **0** [18]).

The entries in the following table show the 4-state values and, where applicable, the corresponding 2-state resolution. For example, when the 4-state **a** and **b** drivers are both **Z**, the resolved 4-state **y**-output is also **Z**. For these same entries, the resolved 2-state **a** and **b** drivers are both **0**(**Z**) (zero after conversion from **Z**) and the corresponding 2-state **y**-output is simply **0**, because the **a** and **b** inputs were both **0** after conversion from **Z** and no trace of the original **Z**-values exist; hence, a solid **0** on the 2-state **y**-output.

a_4	a_2	b_4	b_2	y_4	y_2
0	0	0	0	0	0
0	0	1	1	X	1(X)
0	0	Z	0(Z)	0	0
0	0	X	1(X)	X	1(X)
1	1	0	0	X	1(X)
1	1	1	1	1	1
1	1	Z	0(Z)	1	1(X)
1	1	X	1(X)	X	1
Z	0(Z)	0	0	0	0
Z	0(Z)	1	1	1	1(X)
Z	0(Z)	Z	0(Z)	Z	0
Z	0(Z)	X	1(X)	X	1(X)
X	1(X)	0	0	X	1(X)
X	1(X)	1	1	X	1
X	1(X)	Z	0(Z)	X	1(X)
X	1(X)	X	1(X)	X	1



a_4 - 4-state **a** driver / b_4 - 4-state **b** driver / y_4 - 4-state **y** driver
a_2 - 2-state **a** driver / b_2 - 2-state **b** driver / y_2 - 2-state **y** driver

Figure 2 - 4-state and 2-state resolution using Z->0 and X->1 conversions

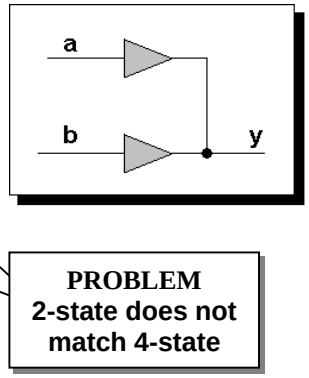
From the above table, it can be seen that resolved nets would be equal to the corresponding 4-state resolved value after making the **Z**->**0** and **X**->**1** conversions.

What if **Z** and **X** were both converted to **0**'s for 2-state simulation? This actually proves to be problematic when trying to match 4-state simulations to 2-state simulations. As shown in Figure 3, there are two places where converting both **Z** and **X** to **0**'s will cause corresponding 2-state and 4-state simulations to have a mismatch.

Consider the Figure 3 case where **a** drives **1** (in both 4-state and 2-state simulations) and where **b** drives **Z** (**Z** for 4-state and **0**(**Z**) for state). After resolving the 4-state drivers, the 4-state value will be **1**, while the 2-state resolution will be **X**, which will change to a **0** (**X->0**). Now there is a 4-state/2-state simulation mismatch. The same type of mismatch happens later in the table.

It should be noted that SystemVerilog 2-state variables convert all **X**'s and **Z**'s to **0** (see Section 6). This conversion is fine for 2-state simulation except for tri-state drivers.

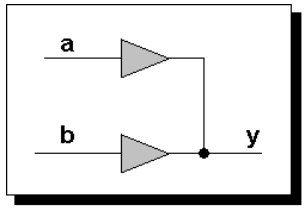
a_4	a_2	b_4	b_2	y_4	y_2
0	0	0	0	0	0
0	0	1	1	X	0(X)
0	0	Z	0(Z)	0	0
0	0	X	0(X)	X	0
1	1	0	0	X	0(X)
1	1	1	1	1	1
1	1	Z	0(Z)	1	0(X)
1	1	X	0(X)	X	0(X)
Z	0(Z)	0	0	0	0
Z	0(Z)	1	1	1	0(X)
Z	0(Z)	Z	0(Z)	Z	0
Z	0(Z)	X	0(X)	X	0
X	0(X)	0	0	X	0
X	0(X)	1	1	X	0(X)
X	0(X)	Z	0(Z)	X	0
X	0(X)	X	0(X)	X	0



a_4 - 4-state **a** driver / b_4 - 4-state **b** driver / y_4 - 4-state **y** driver
a_2 - 2-state **a** driver / b_2 - 2-state **b** driver / y_2 - 2-state **y** driver

Figure 3 - 4-state and 2-state resolution using Z->0 and X->0 conversions

a_4	a_2	b_4	b_2	y_4	y_2
0	0	0	0	0	0
0	0	1	1	X	0(X)
0	0	Z	1(Z)	0	0(X)
0	0	X	0(X)	X	0
1	1	0	0	X	0(X)
1	1	1	1	1	1
1	1	Z	1(Z)	1	1
1	1	X	0(X)	X	0(X)
Z	1(Z)	0	0	0	0(X)
Z	1(Z)	1	1	1	1
Z	1(Z)	Z	1(Z)	Z	1
Z	1(Z)	X	0(X)	X	0(X)
X	0(X)	0	0	X	0
X	0(X)	1	1	X	0(X)
X	0(X)	Z	1(Z)	X	0(X)
X	0(X)	X	0(X)	X	0



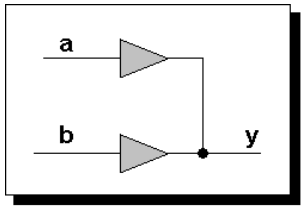
a_4 - 4-state **a** driver / b_4 - 4-state **b** driver / y_4 - 4-state **y** driver
a_2 - 2-state **a** driver / b_2 - 2-state **b** driver / y_2 - 2-state **y** driver

Figure 4 - 4-state and 2-state resolution using Z->1 and X->0 conversions

What if **Z->1** and **X->0** as shown in Figure 4. As can be seen in the table, all 4-state simulations for multiple drivers will match the 2-state simulations for the same drivers. Using the **Z->1** and **X->0** conversion would be a safe representation for 2-state simulations. No simulation mismatches.

And finally, consider the 2-state conversion case where **Z->1** and **X->1** as shown in Figure 5 below.

a_4	a_2	b_4	b_2	y_4	y_2
0	0	0	0	0	0
0	0	1	1	X	1(X)
0	0	Z	1(Z)	0	1(X)
0	0	X	1(X)	X	1(X)
1	1	0	0	X	1(X)
1	1	1	1	1	1
1	1	Z	1(Z)	1	1(X)
1	1	X	1(X)	X	1
Z	1(Z)	0	0	0	1(X)
Z	1(Z)	1	1	1	1
Z	1(Z)	Z	1(Z)	Z	1
Z	1(Z)	X	1(X)	X	1
X	1(X)	0	0	X	1(X)
X	1(X)	1	1	X	1
X	1(X)	Z	1(Z)	X	1
X	1(X)	X	1(X)	X	1



PROBLEM
2-state does not match 4-state

a_4 - 4-state **a** driver / b_4 - 4-state **b** driver / y_4 - 4-state **y** driver
a_2 - 2-state **a** driver / b_2 - 2-state **b** driver / y_2 - 2-state **y** driver

Figure 5 - 4-state and 2-state resolution using Z->1 and X->1 conversions

Examine the Figure 5 case where **a** drives **0** (same in both 4-state and 2-state simulations) and where **b** drives **Z** (**Z** for 4-state and **1**(**Z**) for state). After resolving the 4-state drivers, the 4-state value will be **0**, while the 2-state resolution will be **X**, which will change to a **1** (**X->1**). Now again there is a 4-state/2-state simulation mismatch. The same type of mismatch also happens later in the table.

From these four tables, it can be seen that modeling **HiZ** using 2-state representations only works if the either the (**Z->0** / **X->1**) or the (**Z->1** / **X->0**) conversions are used. VCS does use the first conversion for 2-state simulations but it does not use these 2-state conversions for tri-stateable drivers.

4 Reproducible Power-On 0/1 Random

From its early days of project application at HP, it was realized that the reproducible random simulation method described in Bening and Chaney [11] was a technology that should really be supported by vendors.

In 2002, some EDA vendors expressed interest in licensing the random-state repeatability patent [11]. In spite of HP's efforts to publicize the technology and start wheels in motion for the licensing to EDA simulation vendors, to our knowledge, the licensing still has not happened.

5 Simulation Performance

When most engineers explore the concept of 2-state simulation, an initial reason for considering this technique is the perception of increased simulation performance. This paper has discussed important coding and design techniques using 2-state simulation to avoid problems related to **`X`**-optimism and **`X`**-pessimism, but what about simulation performance?

First, let's look at some of the theoretical performance enhancements that might be possible using 2-state versus 4-state simulation.

5.1 Data representation

4-state data requires at least 2 bits to represent a logic value. In addition to 2 bits for logic values, some Verilog variables may also carry strength information for the logic value. The Verilog HDL has 8-levels of built-in strength, ranging from the highest strength, level-7 for the supply strength, down to level-**`0`** for the **`HiZ`** strength. If unspecified, the default strength is level-6 or strong strength.

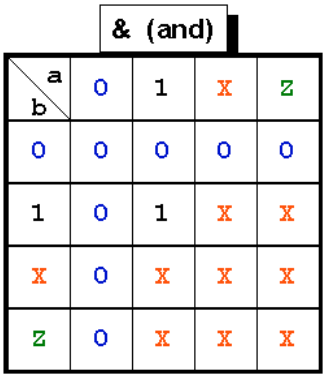
The signal strength can also be different for **`0`** and **`1`** values, which means that 3 bits each (to represent 8 strength levels) may be required to represent the logic-**`1`** strength and the logic-**`0`** strength.

With 2 bits for the logic value (**`0`**, **`1`**, **`X`**, **`Z`**) and 3-bits for the logic-**`1`** strength and 3 bits for the logic-**`0`** strength, Verilog may require 8-bits of compute storage to represent each individual simulated logic bit.

By representing each logic bit as a **`0`** or **`1`** with no strength information, there is a theoretical one-to-one mapping of simulation bits to compute storage bits.

5.2 Native machine operations

When using 4-state logic, simulators require some type of lookup or calculation method to perform Verilog operations. Consider a simple 2-input **`and`** operation.



The simulator must calculate or lookup the resultant and operation for every 4-state bit combination.

Most if not all computers have built-in **and**-ing capability where two bits can be **and**-ed together using native computer instructions, so no calculation or lookup operation is needed.

5.3 Bit, byte & word boundaries

A scalar 2-state **and** operation may not be much more efficient than a 4-state **and**-lookup operation, but there may be significant benefits if a simulator can perform **and**-operations on multi-bit vectors.

Theoretically, a simulator may be able to do vector Boolean 2-state operations much quicker than equivalent 4-state lookup for each bit in a pair of vectors. The subsequent question is whether computers can perform 8-bit, 16-bit, 32-bit and 64-bit operations very quickly because of native word operation-capabilities of computers.

5.4 Memory array representation

One of the reported benefits of using 2-state array representations is that it would reduce the amount of compute memory required to simulate arrays, such as those that are declared to model RAMs and ROMs[5]. The compute memory savings can be significant for certain designs and may allow the simulator to work more efficiently during simulation.

This capability was not tested for this paper.

6 SystemVerilog 2-state RTL simulation

SystemVerilog has built-in 2-state data types. One of the most useful 2-state data types is the **`bit`** type, which can be sized to meet the needs of RTL design variables. SystemVerilog 2-state types currently have one limitation, **`X`**-state and **`Z`**-state both are converted to a logic **`0`**. This limitation is fine for non-tri-state variable representation, but it means that SystemVerilog 2-state types still do not have the flexibility of the HP reproducible random initialization.

SystemVerilog also has a new VHDL-like unresolved 4-state type called **logic**. Unlike Verilog **wire**s, where multiple drivers can be assigned and Verilog performs an automatic resolution function, a compliant SystemVerilog simulator will report an assignment error if more than one source attempts to drive a **logic** data type variable. This feature helps to identify mistakes related to accidentally making multiple driver assignments to the same variable.

The **logic** variable also serves as a variable type in SystemVerilog so if an engineer changes an RTL description from a continuous assignment to a procedural assignment, no declaration change is required (unlike Verilog, which frequently requires that the declaration change from and implicit or explicit **wire** declaration to a **reg** declaration).

The **bit** type is also an unresolved type that allows either one driver assignment or one or more procedural assignments.

Using SystemVerilog's new **typedef** capability, the new **bit** and **logic** data types make it possible to declare re-configurable 4-state/2-state data types that will simulate the same on any SystemVerilog simulator.

Consider the following data types declaration example.

```
`ifdef STATE2
  typedef bit   bit_t;
`else
  typedef logic bit_t;
`endif
```

Example 6 - SystemVerilog 2-state / 4-state types definitions file

This declaration example defines a new user defined data type called **bit_t**. When using this definitions file, by default, the simulation will be run using the SystemVerilog 4-state **logic** data type, but to switch to a SystemVerilog 2-state **bit** data type, one simply adds **+define+STATE2** to the simulation command line.

The user defined **bit_t** data type is used in the following simple 1-bit inverter example. This simple SystemVerilog example file (plus other SystemVerilog models, shown in the Appendix, section 15) are used in benchmark simulations described in section 7.

```
module sv_scalar (
  output bit_t y,
  input  bit_t a
  );

  always @(a)
    y = ~a;
endmodule
```

Example 7 - SystemVerilog 1-bit inverter model

6.1 Designs without tri-state drivers

There are some designs that prohibit the use of tri-state buses and bi-directional buses. For this type of design, there is no need for 4-state resolved data types because there is no need to model **`Z`** and multiple drivers on a common net. For this type of design, the SystemVerilog **`logic`** (4-state) and **`bit`** (2-state) data types can be used as universal data types.

6.2 Designs with tri-state drivers

Some designs do use tri-state and bi-directional buses. For these designs, the question becomes what should be the 2-state strategy for these buses?

There are two strategies related to tri-stateable nets in RTL design. The first strategy is to have all tri-state nets simulate using 4-state semantics. In SystemVerilog, 4-state tri-state nets will continued to be modeled using the automatically resolved Verilog **`wire`** type.

The second strategy is to model tri-state nets as a 2-state type using conversions similar to those shown in either Figure 2 or Figure 4. At present, SystemVerilog has not been defined to model 2-state net types and more study of the 2-state net ramifications are required before making a 2-state net enhancement proposal for SystemVerilog.

For now, it may be useful to declare a tri-stateable net type for SystemVerilog designs that could be easily modified if SystemVerilog is enhanced with 2-state net types. The following declaration would create a **`tri_t`** that would simulate like a **`wire`** for both 2-state and 4-state simulations.

```
`ifdef STATE2
  typedef wire  tri_t;
`else
  typedef wire  tri_t;
`endif
```

Example 8 - SystemVerilog tri_t type declaration for potential 2-state / 4-state tri-stateable net types

In the future, the 2-state portion of this declaration could be modified to accommodate a potential new SystemVerilog 2-state net type.

Unlike Verilog, where the **`wire`** and **`tri`** net types are identical, this SystemVerilog methodology now separates tri-stateable nets (**`tri_t`** - with automatic resolution) from non-tri-state nets (**`bit_t`** - with multi-driver checking and error reporting).

6.3 Clocks and resets

Clock and reset are frequently inputs to a design, For these designs it is reasonable to model these nets using pure 2-state types

```
typedef bit clk_t;

module myasic (
  <port list declaration>,
  ...,
  input clk_t clk, rst_n);

  ...
endmodule
```

6.4 SystemVerilog optimization in VCS

As will be seen in section 7, there is no significant current simulation performance improvement when using the SystemVerilog 2-state types versus the SystemVerilog 4-state types. Considering that the SystemVerilog language was just recently standardized, it is a fair assumption that the VCS SystemVerilog developers are concentrating first on implementing all of the new and powerful SystemVerilog features and that 2-state optimizations may occur with a later release. A quick email to a Synopsys SystemVerilog expert confirmed this theory [5].

Bottom line, SystemVerilog 2-state simulations will not increase SystemVerilog simulation performance at this time but 2-state performance enhancements may be coming in a later release of VCS.

One interesting side-note that is discussed in greater detail in the summary of the benchmarks of section 7 is that the simple 32-bit SystemVerilog inverter models simulated almost %1,300 faster than equivalent Verilog procedural block inverters. This result was completely unexpected.

For the simple **and**-gate benchmarks that are also discussed in section 7, the Verilog models ran 30-50% faster than the equivalent SystemVerilog models.

The performance numbers between SystemVerilog and Verilog data types will probably change drastically over the next few years as more optimization effort is put into more mature SystemVerilog simulators. It would be a mistake to quote the performance figures from this paper even 6 months from now.

It should also be noted that SystemVerilog's greatest advantage over Verilog is not its current performance, but its current features, many which have already been implemented in VCS.

6.5 Reproducible and random variable initialization

SystemVerilog does not currently have reproducible random initialization of variables. SystemVerilog does not even currently have random 2-state initialization of variables. Both of these capabilities would be required to match the 2-state discipline simulation environment.

6.6 Register output modeling & randomization

One of the problems experienced with the 2-state methodology was the random initialization of variable types (typically **reg**-variables), some of which were actual clocked logic outputs and some which represented combinational logic generated from a procedural block. One way engineers addressed this problem was to use library-based flip-flop and register/latch array models.

A similar concept may prove useful as a SystemVerilog data types strategy. Users may choose to declare three different types: **clk_t** (permanent 2-state types used for clk and resets), **bit_t** (compile-time selectable 2-state/4-state type for non-registered outputs) and **reg_t** (to be used for actual registered outputs). There may be a way to find and repeatably randomize all of the registered outputs in a design, leaving the combinational and interconnect **bit_t** variables to settle to a predictable initial state.

```
`ifdef STATE2
  typedef bit   bit_t; // for 2-state interconnect
  typedef bit   reg_t; // for 2-state registered outputs
  typedef wire  tri_t; // for potential 2-state resolved net
types
`else
  typedef logic bit_t; // for 4-state interconnect
  typedef logic reg_t; // for 4-state registered outputs
  typedef wire  tri_t; // for 4-state resolved net types
`endif

typedef   bit   clk_t; // for clk and reset nets
```

Example 9 - SystemVerilog 2-state / 4-state data types for registered outputs, nets, clocks and resets

6.7 SystemVerilog 2-state summary

SystemVerilog features and methodologies may help address some of the issues and limitations related Verilog-based 2-state simulation, but in its current form [17] it still lacks a few important features for a robust 2-state simulation methodology, such as reproducible random state initialization, ability to define the runtime resolution of 2-state **X** and **Z** to be **0**, **1** or random, and resolved 2-state net types.

If a design does not require tri-state nets, then the only missing capabilities are reproducible random state initialization and the ability to define the runtime resolution of 2-state **X** and **Z** to be **0**, **1** or random.

7 VCS Verilog & SystemVerilog 2-state benchmarks

To test some of the optimization theories discussed in section 5, a few benchmark VCS simulations were run on an in-house IBM Thinkpad T30 laptop with VMware workstation - Red Hat Linux 7.3 OS and VCS version 7.1-R16. Benchmarks were also run on an in-house SUN

Ultra-80 with Solaris 8 OS and VCS version 7.1-R12. Relative simulation speeds for the different models on the same platform are the benchmarks that we were most interested in.

The benchmark files selected were intentionally trivial. The authors tried to choose best-case operations that would demonstrate the anticipated optimal performance gains that could be achieved by using 2-state variables with machine-optimized operations.

The two benchmark operations selected were a series of simple inverters of different sizes and 2-input **and**-gates of different sizes. By running these simple benchmarks with millions and billions of stimulus vectors, it was hoped that start-up and non-optimized operations would be filtered, leaving pure optimized performance.

The results were most interesting and non-intuitive! (All of the benchmark model and testbench files are shown in the Appendix, section 15.1).

For real designs with millions of gates and diverse operations, your results will be very different.

For reasons that will be discussed in section 8, HP's large scale server design projects have not used the VCS **`+2state`** option on a full-sized model in about three years. In March 2001, the last time **`+2state`** benchmarks were performed (using VCS 6.0) on a ~2.4M gate-equivalent RTL chip model, the recorded simulation performance was:

- Without **`+2state`** - 2045.96 CPU seconds
- With **`+2state`** - 1928.97 CPU seconds

About a 6% performance improvement.

7.1 The inverter benchmark simulations

The first set of benchmarks used simple inversion operations. By running millions of toggling stimulus input vectors to simple inverter designs, it was hoped that the compiled designs would use very efficient built-in native inversion operations; hence, show a high-percentage difference between 2-state and 4-state simulations.

SystemVerilog configurable 4-state/2-state procedural-block inverter models (1-bit, 8-bit and 32-bit) were compared to Verilog reg-output procedural-block inverter models (1-bit, 8-bit and 32-bit) and to wire-output continuous-assignment inverter models (8-bit and 32-bit). The Verilog models were run both with and without the **`+2state`** switch.

Using a cycle CNT of 250 million to run the benchmarks on both the Linux and Solaris machines, and a subsequent cycle CNT of 1 billion on the Linux machine, the captured simulation times are recorded in Table 1, Table 2 and Table 3 (Appendix - section 15.8).

The most interesting result was that all of the SystemVerilog inverter models ran faster than the Verilog procedural block inverter models.

The biggest surprise was how slow the 8-bit and 32-bit procedural block inverters ran compared to the equivalent SystemVerilog models and equivalent Verilog continuous assignment models. The Verilog 8-bit and 32-bit procedural block inverters ran about %1,300 slower than the equivalent SystemVerilog and Verilog continuous assignment models.

We do not have a good explanation for the poor performance of the Verilog procedural inverters. In fact, the results are very counter intuitive. We expected the 8-bit and 32-bit 2-state Verilog models to be efficiently packed and simulated using native machine word sizes and operations. We may have just selected and unlucky operation to test for this benchmark. Nonetheless, these results were a real surprise.

Other observations from the inverter benchmarks were that the SystemVerilog models, whether they were 2-state or 4-state and whether they were scalar, 8-bit or 32-bit, made very little performance difference.

The Verilog non-scalar 2-state inverter models did run 5-50% faster than the equivalent 4-state models. The Verilog 2-state and 4-state inverter models simulated about the same (as was expected)

7.2 The 2-input and-gate benchmark simulations

Another simple operation, one that would likely have a machine-native equivalent operation is a 2-input **`and`**-ing operation. The Verilog and SystemVerilog files for this benchmark are included in the Appendix of section 15.

For the 2-input **`and`**-gate benchmarks, SystemVerilog configurable 4-state/2-state procedural-block **`and`**-gate models (1-bit, 8-bit and 32-bit) were compared to Verilog **`reg`**-output procedural-block **`and`**-gate models (1-bit, 8-bit and 32-bit) and to **`wire`**-output continuous-assignment **`and`**-gate models (8-bit and 32-bit). The Verilog models were run both with and without the **`+2state`** switch.

Using a cycle CNT of 250 million to run the benchmarks on both the Linux and Solaris machines, and a subsequent cycle CNT of 1 billion on the Linux machine, the captured simulation times are recorded in Table 4, Table 5 and Table 6 (Appendix - section**Error! Reference source not found.**).

The first observation is that all of the Verilog **`and`**-gate models simulated faster than the equivalent SystemVerilog **`and`**-gate models.

Also, there were no surprises this time with the 8-bit and 32-bit procedural **`and`**-gates and the **`+2state`** Verilog model simulations generally ran faster than equivalent 4-state Verilog models.

8 2-State Simulation Realities

HP's large scale server design projects have developed an advanced 2-state simulation methodology with accompanying tools.

However, these projects have not routinely used VCS **`+2state`** simulation. These project continue to apply a two-state Verilog design methodology enforced by Lint rules, supported by random initializations, assertions and **`$check_values`**, but they currently run on 4-state vendor Verilog simulators. Except for the **`X`**-assertions, **`$check_values`**, and incomplete **`/* 4state */`**, the logic would behave the same in 2-state RTL simulation, if any vendor fully supported the full 2-state methodology capabilities.

The HP large scale server design lab phased away from running 99% of their simulation cycles on in-house simulation models to all VCS and other Verilog simulators in the later 1990's. But, they retained the 2-state RTL design style in its Verilog designs. This design style continues into 2004 and beyond, through the power of lint rules leveraged from one project to the next.

In the early 1990's, this HP in-house RTL Verilog simulator was 20X-80X faster than vendor simulators, but by the later 1990's it was only 0.8X - 2.3X faster than commercial tools, so commercial tools largely replaced in-house simulators.

It should be noted that this in-house simulator ran in half the memory: 0.7 GB for RTL ~200M gate-equivalent system model, compared with 1.4 GB for their largest systems now.

9 X-assignments versus other flawed RTL design practices

Much emphasis in this paper has been placed on design failures related to stray **`X`**-assignments, reset **`X`**-startup problems, and **`X`**-optimism and **`X`**-pessimism. Although 4-state simulation with **`X`**-related failures can and do occur, the authors believe that more startup and design problems are actually related to poor reset design and poor multi-clock design.

The 2-state reproducible random initialization methodology can help identify and facilitate debugging of **`X`**-related design problems during simulation. Reset synchronization and multi-clock design flaws are almost impossible to find with either 2-state or 4-state simulation due to the ideal transition of signals in an RTL simulation; hence, engineers are encouraged to still employ good reset design [4] and multi-clock [2] design practices in actual designs.

10 Conclusions

There are two main reasons to consider 2-state simulation, simulation performance and enhanced verification.

Although we have heard of some design teams that claim upwards of 15% increases in simulation performance, we have not seen this level of performance improvement. In fact, at this

time, we do not believe that current levels of increased simulation performance offer generally compelling reasons to use the VCS **+2state** simulation mode.

On the other hand, 2-state simulations using the 2-state methodologies described in this paper offer very compelling reasons to do 2-state simulations. Unfortunately, the **+2state** mode of VCS and current SystemVerilog 2-state capabilities lack the important reproducible random initialization capabilities.

SystemVerilog does have useful 2-state data types and the ability to create user-defined types. These capabilities make it easy to switch back and forth between 2-state and 4-state data types and simulations, but as noted above, SystemVerilog still lacks critical 2-state features that keep it from being capable of doing all the 2-state simulation methodologies described in this paper. To be flexible enough to implement the 2-state methodologies contained in this paper, SystemVerilog would need to implement additional 2-state centric capabilities described in section 11.

SystemVerilog 2-state simulation currently works best on designs that prohibit tri-state buses, but even with these designs it still lacks reproducible random state initialization, which could facilitate identification of many power-on initialization design flaws.

The authors also agree that engineers that do 4-state or 2-state simulation, should not do **X**-testing within **case**, **casex** and **casez** statements.

10.1 casex/casez disagreement

The debate over the use of **casex**/**casez** continues to be the most contested difference between the authors. We both agree that in a 2-state simulation environment, **casex** and **casez** are Boolean equivalents, are safe because 2-state simulations initialize all variables to **0** or **1** (thereby avoiding the potential for uninitialized variables making incorrect **casex**-don't-care matches), and will yield the same 2-state simulation results as equivalent 4-state simulations, but we strongly disagree on which **casex**/**casez** statement should be preferred in RTL design.

Bening prefers **casex** because it is the 2-state Boolean equivalent to **casez**, but does not allow the absurd **X**-comparisons in **case** items.

Cummings prefers **casez** because in 4-state simulation, **casex** will treat uninitialized variables as don't-cares and allow the wrong **case**-item to be matched during pre-synthesis simulations, especially after reset. Cummings, believes using **casex** in a 4-state RTL design borders on malpractice.

Since most engineers do not have access to HP's patented reproducible random state initialization technology and advanced 2-state optimized linting tools and environment, Cummings claims that for the foreseeable future more than 90% of all engineering simulations will continue to be done using 4-state simulations and an engineer who follows the

recommendation from Bening's book to use **casex**, is likely to experience completely avoidable design problems.

Cummings admires Bening's 2-state simulation environment, but Cummings claims that it would be better to use **casez** and allow linting tools to find the absurd **X-case**/**casex**/**casez** item-test code and flag that code as unacceptable.

On the question of **casex** and **casez** usage, the authors agree to disagree.

10.2 X-assignment disagreement

X-assignment has been assailed in a number of papers due to the potential **X**-pessimism and **X**-optimism problems. The RTL 2-state methodology described by Bening prohibits the use of **X**-assignments in RTL code.

Cummings has seen great value in making **X**-assignments in RTL designs. In FSM design, **X**-assignments quickly isolate missing **next** state assignments to help rapidly debug the RTL code (the **X**-assignment acts much like a missing **next**-assignment assertion that quickly helps the RTL designer find and correct missing **next** assignments) and also helps the synthesis tool to identify don't-care conditions to help optimize synthesized designs. In FSM design, the **state** and **next** variables are localized to the FSM modules so **X**-leakage (the possibility that **X**-logic values might escape the module and cause outside **X**-pessimism and **X**-optimism problems) is almost impossible.

Cummings' experience is that **X**-assignments, combined with 4-state simulation can help identify problems early in the debugging cycle. Cliff's methodology encourages **X**-assignment along with waveform dumping and then displaying all signals to identify where **X**'s erroneously exist in the simulation.

As noted in section 8, HP's large scale server design projects do not actually use a 2-state simulator but instead they use a 2-state design discipline with supporting tools and then run simulations on a 4-state simulator. Based on this information, Cummings agrees that prohibiting **X**-assignments in a disciplined 2-state environment running on a 4-state simulator is a reasonable RTL coding restriction.

10.3 Bening's Conclusions

The HP large scale server design lab has successfully applied a 2-state RTL design methodology for a decade, and continues to refine this and other simulation methodologies. In the absence of full support by vendors, this requires in-house support to make it work. For wider acceptance across HP and the worldwide design community, it will take more support by vendors, and those who do Verilog language and methodology training.

Hewlett-Packard's 2-state methodology successfully uses **casex** without problems. By forbidding the **casez**, the methodology eliminates designer time wasted in crafting **casez**s to

deal with **X**'s, and precludes the hideous coding style of doing comparisons to **X**. Rather than deal with complex lint rules that would forbid the testing of **X**'s in the **casez**, prohibiting **casez** use in favor of **casex** eliminates **casez** pitfalls at a minimum cost.

In 4-state simulation, **casex** and **casez** can both cause serious design flaws.

10.4 Cummings' Conclusions

In true 2-state simulation, it does not matter whether you use **casex** or **casez**.

In 4-state simulation, **casex** can cause serious design flaws and should never be used.

Even engineers who use 2-state simulation methodologies should be taught of the **casex** dangers in case they ever have the need to do 4-state simulations.

Do not do **X**-testing with any Verilog **case**, **casex** or **casez** statements.

11 Recommendations

SystemVerilog has useful 2-state variables, but they are not good enough for a general 2-state simulation strategy. System Verilog converts all **X**'s and **Z**'s to **0**, which can be problematic in some simulation environments. To make SystemVerilog good enough to apply the 2-state techniques described in this paper, certain capabilities must be added to the language:

- The ability to initialize all variables to **0**.
- The ability to initialize all variables to **1**.
- The ability to randomly initializate all variables to **0** and **1**.
- The ability to do REPRODUCIBLE random initialization of variables to **0** and **1** and to save the random initialization seed. At this time, this capability is patented by HP. Both authors agree that simulation vendors should work with the HP patent office to get this technology to their customers.

As Verilog is today, design projects can and should freely apply the **$set_values**, **$check_values**, assertion-based, and lint rule methods to support a completely 2-state RTL methodology. For repeatability, elimination of power-on **0**/**1** randomization races is also freely available, but requires some crafting, as well as checking (for which the VCS **+race** option is well-suited).

The other aspect of repeatability requires access to the HP patent, but it is not absolutely essential to a 2-state RTL methodology, only an aid in productivity.

In the current state-of-the-art, projects must develop their own expertise to support the 2-state discipline. This is admittedly a management challenge when working with designers with long-time RTL **X**-crafting experience.

It is the hope of the authors that this paper will enlighten vendors, consultants, and design engineers, to facilitate support and acceptance of the two-state RTL methodology.

12 Job Interview Questions

This paper has described interesting simulation issues and both authors believe that employers may want to consider the following two recommended job interview questions for Verilog RTL design and verification job applicants:

1. Show code that does **case**-item **X**-testing and ask what the candidate thinks of the code (example code follows). If the candidate does not point to the **2'bxx** line and call this coding style bad, the candidate needs to be informed of better coding practices

```
module mux4 (output reg y, input [3:0] a, input [1:0] s);

  always @*
    case (s)
      2'b00: y = a[0];
      2'b01: y = a[1];
      2'b10: y = a[2];
      2'b11: y = a[3];
      2'bxx: y = a[0];
      default: $display("Invalid select lines - s=%b", s);
    endcase
endmodule
```

2. Ask the candidate what the difference is between **casez** and **casex** and ask the candidate which is safer in RTL design.

The correct response will be dependent upon whether the candidate is seeking a position on a project team supporting a 2-state RTL methodology or elsewhere, but the candidate should note that **casex** and **casez** are extremely dangerous in 4-state simulations, and for a 2-state project position, the candidate should indicate that for 2-state simulation, **casez** and **casex** are Boolean equivalents while **casez** is prohibited because **casez** permits very bad **X**-bit comparisons (as described in interview question #1), and may include **casez** with bad **X**-assignments. Even 2-state RTL project candidates should understand the differences in the event they ever decide to change jobs. This is especially true if they expect to lead project teams away from **X**-based RTL methods to a fully 2-state RTL methodology.

13 Acknowledgements

The authors wish to thank Dave Rich [5] and Peter Flake [14] of Synopsys for their expertise and responses concerning how VCS-Verilog and VCS-SystemVerilog work and status concerning SystemVerilog optimizations. The authors also thank Stuart Sutherland for his review, comments and feedback that help improve the content and flow of this paper.

14 References

[1] Clifford E. Cummings, '"full_case parallel_case", the Evil Twins of Verilog Synthesis,' *SNUG'99 Boston (Synopsys Users Group Boston, MA, 1999) Proceedings*, October 1999. (Also available online at www.sunburst-design.com/papers)

[2] Clifford E. Cummings, "Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs," *SNUG 2001 (Synopsys Users Group Conference, San Jose, CA, 2001) User Papers*, March 2001, Section MC1, 3rd paper. Also available at www.sunburst-design.com/papers

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

[4] Clifford E. Cummings, Don Mills, Steve Golson, "Asynchronous & Synchronous Reset Design Techniques - Part Deux" *SNUG (Synopsys Users Group) Boston, 2003 User Papers*, September 2003. Also available at www.sunburst-design.com/papers and www.lcdm-eng.com/papers.htm

[5] Dave Rich, Synopsys - Personal communication

[6] Don Mills and Clifford E. Cummings, "RTL Coding Styles That Yield Simulation and Synthesis Mismatches," *SNUG (Synopsys Users Group) 1999 Proceedings*, section-TA2 (2nd paper), March 1999. Also available at www.lcdm-eng.com/papers.htm and www.sunburst-design.com/papers

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

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

[9] Lionel Bening, "Simulation of High Speed Computer Logic," *Proc. 6th Design Automation Conf.*, June, 1969.

[10] Lionel Bening, "A Two-State Methodology for RTL Logic Simulation," *Proc. 36th Design Automation Conf.*, June, 1999.

[11] Lionel Bening, Kenneth Chaney, *Generation of Reproducible Random Initial States in RTL Simulators*, Hewlett-Packard patent US6061819, May, 2000.

[12] Lionel Bening and Harry Foster, Principles of Verifiable RTL Design, Kluwer Academic Publishers, May, 2001.

[13] Mike Turpin, "The dangers of living with an X (bugs hidden in your Verilog)," SNUG Boston, Sep. 2003, www.snug-universal.org/cgi-bin/search/search.cgi?Boston,+2003

[14] Peter Flake, Synopsys - Personal communication

[15] Synopsys SolvNet, Doc Name: 002415, “Values of "X" and "Z" in a 2-state VCS Simulation,” Last Modified: 08/26/2002 - solvnet.synopsys.com/retrieve/002415.html

[16] Synopsys SolvNet, Doc Name: 009291, “Random mapping of X’s and Z’s to 1’s and 0’s in 2state,” Last Modified: 04/06/2004 - solvnet.synopsys.com/retrieve/009291.html

[17] *SystemVerilog 3.1a Accellera’s Extensions to Verilog*, Accellera, 2004, freely downloadable from: www.eda.org/sv

[18] *VCS User Guide, Version 7.0.2*, Synopsys, September 2003

15 Appendix

This appendix contains the simple models that were used to benchmark 2-state and 4-state Verilog simulations and 2-state and 4-state SystemVerilog simulations described and analyzed in section 7.

15.1 SystemVerilog inverter models

All of the SystemVerilog inverter models use the switch-able **`bit_t`** type definition from Example 6. SystemVerilog 4-state simulations are the default simulation data types, while the 2-state simulation data types were invoked using the **`+STATE2`** command line switch.

The scalar SystemVerilog inverter model was shown in Example 7.

The second SystemVerilog model is an 8-bit inverter using the SystemVerilog **`bit_t`** data type and procedural block.

```
module sv_vector8 (
  output bit_t [7:0] y,
  input  bit_t [7:0] a
  );

  always @(a)
    y = ~a;
endmodule
```

Example 10 - SystemVerilog 8-bit inverter model

The third SystemVerilog model is a 32-bit inverter using the SystemVerilog **`bit_t`** data type and procedural block.

```
module sv_vector32 (
  output bit_t [31:0] y,
  input  bit_t [31:0] a
  );

  always @(a)
    y = ~a;
endmodule
```

Example 11 - SystemVerilog 32-bit inverter model

15.2 Verilog reg-type inverter models

The first Verilog model is a simple scalar inverter using the Verilog **`reg`** data type and procedural block.

```
module vl_scalar (
  output reg y,
  input      a
  );

  always @(a)
    y = ~a;
endmodule
```

Example 12 - Verilog 1-bit inverter model - reg output

The second Verilog model is an 8-bit inverter using the Verilog **`reg`** data type and procedural block.

```
module vl_vector8 (
  output reg [7:0] y,
  input      [7:0] a
  );

  always @(a)
    y = ~a;
endmodule
```

Example 13 - Verilog 8-bit inverter model - reg outputs

The third Verilog model is a 32-bit inverter using the Verilog **`reg`** data type and procedural block.

```
module vl_vector32 (
  output reg [31:0] y,
  input      [31:0] a
  );

  always @(a)
    y = ~a;
endmodule
```

Example 14 - Verilog 32-bit inverter model - reg outputs

15.3 Verilog net-type inverter models

The fourth Verilog model is an 8-bit inverter using the **`wire`** data type and continuous assignment instead of a variable data type.

```
module vl_wire8 (
  output wire [7:0] y,
  input  wire [7:0] a
  );

  assign y = ~a;
endmodule
```

Example 15 - Verilog 8-bit inverter model - wire outputs

The fifth Verilog model is a 32-bit inverter using the **`wire`** data type instead of a variable data type.

```
module vl_wire32 (
  output wire [31:0] y,
  input  wire [31:0] a
  );

  assign y = ~a;
endmodule
```

Example 16 - Verilog 32-bit inverter model - wire outputs

15.4 SystemVerilog 2-input and-gate models

All of the SystemVerilog 2-input **`and`**-gate models use the switch-able **`bit_t`** type definition from Example 6 (same as the SystemVerilog inverter models). SystemVerilog 4-state simulations are the default simulation data types, while the 2-state simulation data types were invoked using the **`+STATE2`** command line switch.

The first SystemVerilog model is a single 2-input **`and`**-gate using the SystemVerilog **`bit_t`** data type and procedural block.

```
module sv_scalar (
  output bit_t y,
  input  bit_t a, b
  );

  always_comb
    y = a & b;
endmodule
```

Example 17 - SystemVerilog 1-bit 2-intput and gate model

The second SystemVerilog model is eight, 2-input **`and`**-gates using the SystemVerilog **`bit_t`** data type and procedural block.

```
module sv_vector8 (
  output bit_t [7:0] y,
  input  bit_t [7:0] a, b
  );

  always_comb
    y = a & b;
endmodule
```

Example 18 - SystemVerilog 8-bit 2-intput and gate model

The third SystemVerilog model is 32, 2-input **and**-gates using the SystemVerilog **bit_t** data type and procedural block.

```
module sv_vector32 (
  output bit_t [31:0] y,
  input  bit_t [31:0] a, b
  );

  always_comb
    y = a & b;
endmodule
```

Example 19 - SystemVerilog 32-bit 2-intput and gate model

15.5 Verilog reg-type 2-input and-gate models

The first Verilog model is single 2-input **and**-gate using the Verilog **reg** data type and procedural block.

```
module vl_scalar (
  output reg y,
  input      a, b
  );

  always_comb
    y = a & b;
endmodule
```

Example 20 - Verilog 1-bit 2-intput and gate model - reg output

The second Verilog model is eight, 2-input **and**-gates using the Verilog **reg** data type and procedural block.

```
module vl_vector8 (
  output reg [7:0] y,
  input      [7:0] a, b
  );

  always_comb
    y = a & b;
endmodule
```

Example 21 - Verilog 8-bit 2-intput and gate model - reg outputs

The third Verilog model is 32, 2-input **and**-gates using the Verilog **reg** data type and procedural block.

```
module vl_vector32 (
  output reg [31:0] y,
  input      [31:0] a, b
  );

  always_comb
    y = a & b;
endmodule
```

Example 22 - Verilog 32-bit 2-intput and gate model - reg outputs

15.6 Verilog net-type 2-input and-gate models

The fourth Verilog model is 8 2-input **and**-gates using the **wire** data type and continuous assignment instead of a variable data type.

```
module vl_wire8 (
  output wire [7:0] y,
  input  wire [7:0] a, b
  );

  assign y = a & b;
endmodule
```

Example 23 - Verilog 8-bit 2-intput and gate model - wire outputs

The fifth Verilog model is 32 2-input **and**-gates using the **wire** data type and continuous assignment instead of a variable data type.

```
module vl_wire32 (
  output wire [31:0] y,
  input  wire [31:0] a, b
  );

  assign y = a & b;
endmodule
```

Example 24 - Verilog 32-bit 2-intput and gate model - wire outputs

15.7 Benchmark testbench file

The benchmark testbench file is a simple testbench that conditionally compiles the desired benchmark simulation file and changes the inputs millions to billions of times to test the potential performance improvement that might be had by simple Boolean operations in either 4-state or 2-state modes.

```
`timescale 1ns/1ns
module tb;

  `ifdef SV_SCALAR
  bit_t y, a, b, t1, t2;

  sv_scalar u1 ( .y(y), .a(a), .b(b) );
  `endif

  `ifdef SV_VECTOR8
  bit_t [7:0] y, a, b, t1, t2;

  sv_vector8 u1 ( .y(y), .a(a), .b(b) );
  `endif

  `ifdef SV_VECTOR32
  bit_t [31:0] y, a, b, t1, t2;

  sv_vector32 u1 ( .y(y), .a(a), .b(b) );
  `endif

  `ifdef VL_SCALAR
  reg  a, b, t1, t2;
  wire y;

  vl_scalar u1 ( .y(y), .a(a), .b(b) );
  `endif

  `ifdef VL_VECTOR8
  reg  [7:0] a, b, t1, t2;
  wire [7:0] y;

  vl_vector8 u1 ( .y(y), .a(a), .b(b) );
  `endif

  `ifdef VL_VECTOR32
  reg  [31:0] a, b, t1, t2;
  wire [31:0] y;

  vl_vector32 u1 ( .y(y), .a(a), .b(b) );
  `endif

  `ifdef VL_WIRE8
  wire [7:0] y;
```

```
  reg  [7:0] a, b, t1, t2;

  vl_wire8 u1 ( .y(y), .a(a), .b(b) );
  `endif

  `ifdef VL_WIRE32
  wire [31:0] y;
  reg  [31:0] a, b, t1, t2;

  vl_wire32 u1 ( .y(y), .a(a), .b(b) );
  `endif

  initial begin
    $timeformat(-9,0,"ns",20);
    $monitor("%t: y=%h   a=%h", $time, y, a);
  end

  initial begin
    {b,a,t2,t1} <= 4'b0011;
    fork
      repeat(`CNT) #1 {t1,b,a,t2} = {b,a,t2,t1};
      #4 $monitoroff;
    join
    $monitoron;
    repeat(4) #1 {t1,b,a,t2} = {b,a,t2,t1};
    #1 $finish;
  end
endmodule
```

Example 25 - benchmark testbench

There is a **`$timeformat`** command to beautify the output and a **`$monitor`** command that is turned off after 4 toggles and turned back on 4 toggles before finishing the simulation. The macro definition for the **`CNT`** value is actually kept in a separate file that all of the tests call, so it can be easily changed and all the tests run again.

15.8 The Verilog inverter & and-gate benchmark results

The simulations results from testing the various varieties of inverters are shown in Table 1, Table 2 and Table 3.

The simulations results from testing the various varieties of 2-input **`and`**-gates are shown in Table 4, Table 5 and Table 6.

IBM Thinkpad T30 - VM Ware / Red Hat Linux 7.3 Inverter Design - Cycle CNT = 250,000,000						
	SystemVerilog Inverter Design		Verilog-reg Inverter Design		Verilog-wire Inverter Design	
	Design File	Simulation Time (sec)	Design File	Simulation Time (sec)	Design File	Simulation Time (sec)
2-state	sv_s_2x.v	119.28	vl_s_2x.v	143.17	**SCALAR**	
4-state	sv_s_4x.v	119.38	vl_s_4x.v	146.22		
2-state	sv_v8_2x.v	119.31	vl_v8_2x.v	297.16	**8-BIT**	
4-state	sv_v8_4x.v	117.47	vl_v8_4x.v	376.22	vl_wire8.v	122.02
2-state	sv_v32_2x.v	118.82	vl_v32_2x.v	1510.68	**32-BIT**	
4-state	sv_v32_4x.v	127.20	vl_v32_4x.v	1620.21	vl_wire32.v	114.28

Table 1 - Linux inverter benchmark simulations - cycle CNT = 250 million

IBM Thinkpad T30 - VM Ware / Red Hat Linux 7.3 Inverter Design - Cycle CNT = 1,000,000,000						
	SystemVerilog Inverter Design		Verilog-reg Inverter Design		Verilog-wire Inverter Design	
	Design File	Simulation Time (sec)	Design File	Simulation Time (sec)	Design File	Simulation Time (sec)
2-state	sv_s_2x.v	526.49	vl_s_2x.v	542.15	**SCALAR**	
4-state	sv_s_4x.v	532.30	vl_s_4x.v	620.92		
2-state	sv_v8_2x.v	599.60	vl_v8_2x.v	1104.82	**8-BIT**	
4-state	sv_v8_4x.v	560.84	vl_v8_4x.v	1344.24	vl_wire8.v	458.19
2-state	sv_v32_2x.v	544.09	vl_v32_2x.v	6583.95	**32-BIT**	
4-state	sv_v32_4x.v	530.06	vl_v32_4x.v	6851.48	vl_wire32.v	460.47

Table 2 - Linux inverter benchmark simulations - cycle CNT = 1 billion

SUN UltraSparc 80 - Solaris 8 Inverter Design - Cycle CNT = 250,000,000						
	SystemVerilog Inverter Design		Verilog-reg Inverter Design		Verilog-wire Inverter Design	
	Design File	Simulation Time (sec)	Design File	Simulation Time (sec)	Design File	Simulation Time (sec)
2-state	sv_s_2x.v	337	vl_s_2x.v	349	**SCALAR**	
4-state	sv_s_4x.v	345	vl_s_4x.v	378		
2-state	sv_v8_2x.v	352	vl_v8_2x.v	796	**8-BIT**	
4-state	sv_v8_4x.v	345	vl_v8_4x.v	931	vl_wire8.v	302
2-state	sv_v32_2x.v	348	vl_v32_2x.v	3256	**32-BIT**	
4-state	sv_v32_4x.v	347	vl_v32_4x.v	4876	vl_wire32.v	302

Table 3 - Solaris inverter benchmark simulations - cycle CNT = 250 million

IBM Thinkpad T30 - VM Ware / Red Hat Linux 7.3 And-gate Design - Cycle CNT = 250,000,000						
	SystemVerilog AND-gate Design		Verilog-reg AND-gate Design		Verilog-wire AND-gate Design	
	Design File	Simulation Time (sec)	Design File	Simulation Time (sec)	Design File	Simulation Time (sec)
2-state	sv_s_2x.v	128.84	vl_s_2x.v	97.26	**SCALAR**	
4-state	sv_s_4x.v	134.72	vl_s_4x.v	101.77		
2-state	sv_v8_2x.v	117.41	vl_v8_2x.v	79.30	**8-BIT**	
4-state	sv_v8_4x.v	125.34	vl_v8_4x.v	85.71	vl_wire8.v	89.46
2-state	sv_v32_2x.v	143.54	vl_v32_2x.v	129.33	**32-BIT**	
4-state	sv_v32_4x.v	147.22	vl_v32_4x.v	120.25	vl_wire32.v	112.88

Table 4 - Linux And-gate benchmark simulations - cycle CNT = 250 million

IBM Thinkpad T30 - VM Ware / Red Hat Linux 7.3 And-gate Design - Cycle CNT = 1,000,000,000						
	SystemVerilog AND-gate Design		Verilog-reg AND-gate Design		Verilog-wire AND-gate Design	
	Design File	Simulation Time (sec)	Design File	Simulation Time (sec)	Design File	Simulation Time (sec)
2-state	sv_s_2x.v	526.08	vl_s_2x.v	402.02	**SCALAR**	
4-state	sv_s_4x.v	533.67	vl_s_4x.v	442.69		
2-state	sv_v8_2x.v	467.92	vl_v8_2x.v	310.37	**8-BIT**	
4-state	sv_v8_4x.v	540.77	vl_v8_4x.v	414.97	vl_wire8.v	393.98
2-state	sv_v32_2x.v	607.87	vl_v32_2x.v	658.55	**32-BIT**	
4-state	sv_v32_4x.v	597.94	vl_v32_4x.v	604.31	vl_wire32.v	536.60

Table 5 - Linux And-gate benchmark simulations - cycle CNT = 1 billion

SUN UltraSparc 80 - Solaris 8 And-gate Design - Cycle CNT = 250,000,000						
	SystemVerilog Inverter Design		Verilog-reg Inverter Design		Verilog-wire Inverter Design	
	Design File	Simulation Time (sec)	Design File	Simulation Time (sec)	Design File	Simulation Time (sec)
2-state	sv_s_2x.v	358	vl_s_2x.v	296	**SCALAR**	
4-state	sv_s_4x.v	376	vl_s_4x.v	358		
2-state	sv_v8_2x.v	342	vl_v8_2x.v	245	**8-BIT**	
4-state	sv_v8_4x.v	378	vl_v8_4x.v	270	vl_wire8.v	288
2-state	sv_v32_2x.v	406	vl_v32_2x.v	312	**32-BIT**	
4-state	sv_v32_4x.v	430	vl_v32_4x.v	370	vl_wire32.v	366

Table 6 - Solaris And-gate benchmark simulations - cycle CNT = 250 million

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 22 years of ASIC, FPGA and system design experience and 12 years of Verilog, synthesis and methodology training experience.

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

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

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

Lionel Bening's professional career has been devoted to CAD tool development, evaluation, integration, and training, mostly related to HDLs and simulation. This career included stints with Control Data (Cray-based designs), Convex Computer Corporation (affordable supercomputers), and the University of Minnesota Computer Science Dept. (adjunct lecturer on logic simulation and timing verification topics).

For the past nine years, he has been a Logic Design Methodology Specialist with Hewlett-Packard (Richardson, TX).

His professional activities included participation in the June, 1981 Woods Hole Study Group, which began composing the requirements for VHDL, and the subsequent VHDL Modeling Group

He is co-author of patents on repeatable random initialization and HDL/regular-expression-based implicit interconnection, and a book of Verifiable RTL. He has also been a paper contributor to DAC, HDLCon/DVCon, DesignCon, SNUG, HP Journal, as well as design workshops and trade journals.

Lionel holds a BSEE from the University of Minnesota.
E-mail address: lionel.bening@hp.com. Web site: www.lionelbening.org

An updated version of this paper can be downloaded from the web site:
www.sunburst-design.com/papers
(Data accurate as of August 10, 2004)