# SystemVerilog Assertions Are For Design Engineers Too!

Don Mills
LCDM Engineering
mills@lcdm-eng.com

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

**ABSTRACT**

SystemVerilog Assertions (SVA) are getting lots of attention in the verification community, and rightfully so. Assertions Based Verification Methodology is a critical improvement for verifying large, complex designs. But, we design engineers want to play too! Verification engineers add assertions to a design after the HDL models have been written. This means placing the assertions on module boundaries, binding external assertions to signals within the model, or modifying the design models to insert assertions within the code.

Design engineers can, and should, add assertions within a design while the HDL models are being coded. There are challenges with doing this, however. A decision needs to be made before design work begins as to what types of scenarios the design engineer should provide assertions for. The design and verification teams need to coordinate efforts to avoid writing redundant assertions, or worse, to miss adding assertions on key functionality. Design engineers have to learn the SVA (or PSL) assertions language, which is a complex languages in and of itself. The design process will probably take longer, because adding (and debugging) assertions as the design is being coded may not be trivial.

This paper explores the challenges of having design engineers add assertions as HDL models are developed. The test case used is a small DSP design. Suggestions and guidelines are presented for partitioning assertions between the design and verification teams, and minimizing the impact on design time.

## Table of Contents

1.0 Introduction ....................2
2.0 A brief overview of SystemVerilog Assertions ....................3
    2.1 SystemVerilog Assertion advantages .................... 4
    2.2 Immediate assertions .................... 4
    2.3 Concurrent assertions .................... 5
    2.4 SystemVerilog constructs with built-in assertions .................... 7
    2.5 Invariant, sequential and eventuality assertions .................... 8

3.0 Where assertions can be specified ....9
4.0 An assertions test plan for a small DSP design ....10
4.1 The DSP design blocks .... 10
4.2 An assertions plan for the DSP .... 11
4.3 Rule of thumb for delegating assertions .... 12
4.4 An example assertions test plan .... 12
5.0 Example assertions per the assertion test plan ....16
5.1 ALU block: No X on inputs; All opcodes decoded .... 16
5.2 RAM: Mutually exclusive read/write control lines .... 17
5.3 Controller state machine: At posedge of clock, state is always one-hot .... 17
5.4 Controller state machine: Proper state sequencing .... 17
6.0 Lessons learned ....18
6.1 Developing the assertions test plan takes time .... 18
6.2 The assertion test plan helps identify similar assertions .... 18
6.3 Take advantage of SystemVerilog constructs with built-in assertions .... 19
6.4 An assertion should not just duplicate RTL functionality .... 19
6.5 Disabling assertions during resets is important .... 19
6.6 Need to be flexible on who writes the assertions .... 19
6.7 Testing logic that has enumerated types requires access to the enumerated names .. 20
6.8 Enumerated type labels disappear after synthesis .... 20
6.9 Assertion based verification may require different design partitioning .... 21
7.0 Conclusions ....21
8.0 References ....22
9.0 About the authors ....23

## 1.0 Introduction

Design engineers have a number of objectives that must be met for each design. These objectives include meeting area, timing, and power requirements. And of course, the design must also represent the specification. There are a number of tools and techniques used by design and verification engineers to verify that a design correctly represents its specification. The focus discussed in this paper is to show design engineers how they can take advantage of SystemVerilog Assertions (SVA) as a tool to prove the intent of the design. This paper is written from a design engineer's perspective, targeted toward design engineers that do not have a strong background in assertion techniques and usage.

The application of assertions to a design involves a number of considerations, the first of which is determining which assertions will be written by design engineers and which will be written by the verification and/or system engineers. Many of the assertions to be used to verify a design can, and should, be determined before the design implementation phase begins. During the implementation, the designer may need to add assertions beyond what was initially planned, based on the design implementation. The designer may also need to make recommendations to the verification team for additional top level assertions based on the details of the implementation.

There will be a learning curve for a designer who is new to assertions, but with the help of this paper and other available resources, a designer can start implementing SystemVerilog Assertions into a design within a short period of time. As a beginning point, there are two main

considerations a designer must make when adding assertions. The first consideration is to recognize where to add assertions in a design. The second is to determine which type of assertion to add. This paper will address these two considerations from a designer's perspective.

A small 16-bit DSP is used as a case study for implementing SystemVerilog Assertions. This DSP is used as a student design lab for Sutherland HDL training courses, and is designed to be simple enough that the students can complete the DSP design in four to six hours. The design is complex enough to require modeling of a variety of RTL blocks including a finite state machine (FSM), an arithmetic logic unit (ALU), various registers and counters, and a bidirectional bus. This DSP lab serves a dual purpose. The DSP design is a final project for students learning to write synthesizable Verilog/SystemVerilog models. A complete, but broken, version of the DSP design is used for students learning to write SystemVerilog assertions.

The DSP design lab has been used for a number of years in Sutherland HDL training courses. Over the years the instructors have observed a number of common coding errors and specification misunderstandings. These errors include:

- Bus contention on the multi-driver, bidirectional data bus
- MSB of the 16-bit data bus not tri-stated
- The trailing edge of reset not properly synchronized with the clocks
- The ALU exception and zero flags not properly cleared on the next operation after a flag is set
- Wrong bits of the instruction word bus connected to the program counter and controller
- Race conditions due to misuse of blocking and nonblocking assignments

Assertions will provide feedback to the designer during simulation regarding these common errors, as well as many others potential coding discussed later in this paper.

The conclusions of this paper will summarize the ease (or difficulty) of adding assertions to the DSP design, as well as several lessons learned from having students (both design engineers and verification engineers) specify SystemVerilog Assertions.

## 2.0 A brief overview of SystemVerilog Assertions

SystemVerilog has two types of formal assertion statements: *immediate assertions* and *concurrent assertions*. Both immediate and concurrent assertions perform a test on an aspect of the design. At the completion of the test, pass or fail statements can be executed. In addition to the fail statements, the failure severity level can be set for each assertion. The severity levels are set by calling system tasks which include:

- **`$fatal`** — run-time fatal
- **`$error`** — run-time error
- **`$warning`** — run-time warning
- **`$info`** — notes that the assertion failed, but without any severity

Each of these severity level system tasks will print a tool-specific message, and may also include user comments using the same syntax as the Verilog `$display` system task. Specifying a severity level is optional; the default for assertion failures is **`$error`**, if not explicitly specified.

## 2.1 SystemVerilog Assertion advantages

Assertion-like checks can be added into a design using the standard Verilog language. There are, however, some serious drawbacks to writing assertion checks this way. One disadvantage is that complex assertion-like checks can require writing complex Verilog code. Another disadvantage is that checks written in Verilog will appear to be part of the RTL model to a synthesis compiler. A third disadvantage is that an assertion-like check written in the Verilog language will be active throughout simulation; there is no simple way to disable some or all checks during simulation.

One of the biggest advantages to using SystemVerilog assertions over embedding checks written using Verilog statements in the design is that SystemVerilog assertions are ignored by synthesis. The designer does not need to include `translate_off`/`translate_on` synthesis pragmas scattered throughout the RTL code. The reality is that most designers simply do not embed "homemade" assertions written in Verilog into their designs because the extra code and synthesis pragmas convolutes the RTL model. For instance, rarely is an `if` statement implemented as follows in an RTL model:

```
if (if_condition)
  // do true statements
else
//translate_off
if (!if_condition)
//translate_on
  // do the not true statements
//translate_off
else
  $display("if condition tested either an X or Z");
//translate_on
```

A second advantage to using SystemVerilog assertions over "homemade" assertion-like checks written in Verilog is that SystemVerilog Assertions can easily be disabled or enabled at any point during simulation, as needed. This allows designers to add assertions to the RTL code, but then disable the assertions for simulation speed later in the design process, or to focus assertion checks to a specific region of the design. Assertion can be controlled in several ways:

- All assertions in specific levels of hierarchy can be turned off and on.
- Assertions within a specific module can be disabled and enabled.
- Assertions can be individually disabled and enabled.

The system tasks to perform these operations are **`$assertoff`**, **`$assertkill`**, and **`$asserton`**.

## 2.2 Immediate assertions

Immediate assertions, as the name implies, execute immediately, in zero simulation time. Immediate assertions can be placed anywhere procedural statements can be placed; within always blocks, initial blocks, tasks, and functions. Any procedural statement can be used in the pass or fail statement part of an immediate assertion. Care must be taken to ensure that no design functionality is modeled in the pass/fail parts of an assertion, because it will be ignored by synthesis, thus causing a modeling difference between simulation and synthesis.

The syntax for an immediate assertion is:

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

Note that the `pass_statement` is optional. Most engineers will not specify any pass statements; engineers are looking for failures, and don't need to do anything when an assertion succeeds. The **`else`** fail_statement is also optional. If it is left off, a tool-generated message will be printed, with a default severity level of **`$error`**. Designers often specify an assertion fail condition in order to augment the default assertion failure information provided by simulation.

Below are two examples of immediate assertions to check for X or Z values in combinational logic conditional expressions.

```
module example_1 (
  input        ifcond, a, b,
  output logic if_out
);

  always_comb begin
    assert (^ifcond !== 1'bx);
      else $error("ifcond = X");
    if (ifcond)
      if_out = a;
    else
      if_out = b;
  end
endmodule


module example_2 (
  input        a, b, c, d,
  input [1:0]  sel,
  output logic out
);

  always_comb begin
    assert (^sel !== 1'bx);
      else $error("case_Sel = X");
    case (sel)
      2'b00 : out = a;
      2'b01 : out = b;
      2'b10 : out = c;
      2'b11 : out = d;
    endcase
  end
endmodule
```

For more examples and details on using immediate assertions to trap logic X and Z problems, refer to the paper "*Being Assertive With Your X*" [2], published in the proceedings of SNUG 2004.

## 2.3 Concurrent assertions

Concurrent assertions use a clock to trigger the assertion evaluation. The primary difference

between immediate and concurrent assertions is that concurrent assertions evaluate conditions over time, whereas immediate assertions test at the point in time when the assertion is called. The syntax difference between the two types of assertions is very slight. The concurrent assertion directive adds the key word **property**. The syntax for a concurrent assertion directive is:

```
assert property (property_expr) [pass_statement;] [else fail_statement;]
```

The argument to **assert property** is a *property expression* (whereas, the argument to an immediate assertion is a simple boolean expression). A property expression comprises of a clock specification and a sequence of Boolean expressions tested over time. The expressions are evaluated on the clock edge specified. The sequence of Boolean expressions can be spread over multiple clock cycles by using the **##** cycle delay operator between each expression.

The following code is an example of a completely self contained concurrent assertion directive.

```
example_3:  assert property @(posedge clk) ( req ##1  grant ##10
                                             !req ##1 !grant);
            else $error("bus request failed");
```

In English, the sequence in the example above is read as: *"`req` should be true (high) immediately, followed by `grant` being true (high) one clock cycle later. After ten more clock cycles, `req` should be false (low), followed by `grant` being false (low) one clock cycle later."* For this assertion to succeed, each expression must evaluate true at its specified time.

The property expression of a concurrent assertion can be defined in a separate block of code, between the keywords **property** and **endproperty**. This enables the same property expression to be re-used by multiple concurrent assertions.

```
property bus_req_prop2;
  @(posedge clk) req ##1 grant ##10 !req ##1 !grant;
endproperty

example_4:  assert property (bus_req_prop2};
            else $error("bus request failed");
```

A complex property expression can be broken into smaller sequence building blocks, specified between **sequence** and **endsequence**. This is illustrated in the following example.

```
sequence start_bus_req;
  req ##1 grant;
endsequence

sequence end_bus_req;
  !req ##1 !grant;
endsequence

property bus_req_prop3;
  @(posedge clk) start_bus_req ##10 end_bus_req;
endproperty

example_5: assert property (bus_req_prop3};
```

Most concurrent assertions are written so that the assertion "*fires*" each and every clock cycle,

throughout simulation. This allows the assertion to run in the background, concurrent with the design functionality. Since the assertion fires every clock cycle, an assertion with a sequence that takes twelve clock cycles to execute could possibly have twelve concurrent threads running at the same time; each thread starting on a subsequent clock cycle. In the bus request/grant sequence examples above, `req` will be tested every clock cycle, starting a new concurrent assertion thread. If `req` is true, the thread will continue and test for `grant` on the next clock cycle. If `req` is false, however, the assertion will fail at that point in time. This would be a false failure, since there is no design problem.

Property expressions can be specified with an implication operator, either **`|->`** or **`|=>`** (there is an important difference in these two operators, but we won't go into that level of detail in this paper—our examples will just use the **`|->`** implication operator, which is the style we recommend in our training courses). An implication operator tells the property not to evaluate a sequence (the *consequent*) unless the first condition before the operator (the *antecedent*) is true. In the code examples above, the designer will most likely only want to test the request/grant sequence when `req` is true. For clock cycles where `req` is false, the assertion is a don't care, and the request/grant sequence should not be evaluated. In the following example, the implication operator prevents the sequence from continuing when `req` is not true. The assertion does not fail, it simply does not run.

```
property bus_req_prop4;
  @(posedge clk) req |-> ##1 grant ##10 !req ##1 !grant;
endproperty

example_6:  assert property (bus_req_prop4);
```

A property expression can automatically be disabled under specific conditions using a **`disable iff`** statement. When the condition specified with `disable iff` is false, the assertion does not fire, and kills any threads that are already running. A primary usage of `disable iff` is to prevent false assertion failures from occurring while a design is being reset. The next example illustrates this usage.

```
property bus_req_prop5;
  @(posedge clk)
  disable iff (reset)  // disable assertion during reset
  req |-> ##1 grant ##10 !req ##1 !grant;
endproperty

example_5:  assert property (bus_req_prop5);
```

In addition to `disable iff` and the implication operator, there are several other property operators that are useful in defining properties. These other operators are not covered in this paper.

### 2.4 SystemVerilog constructs with built-in assertions

In addition to the **`assert`** construct, SystemVerilog provides some important RTL modeling constructs that have built-in assertion behavior. Three primary constructs are briefly covered in this section. By using these constructs, design engineers gain the advantages of adding assertions to a design, without having to actually write the assertions.

**`always_comb`** — This specialized procedural block provides several capabilities that the Verilog `always` procedural block does not have. In brief, this specialized block enforces a synthesizable modeling style. In addition, the `always_comb` block allows software tools to check that the code within the block functions as combinational logic. For example, there are a number of coding mistakes in intended combinational logic that can result in latches. By using `always_comb`, these coding mistakes can be detected early in the design stage. Writing assertions to detect inadvertent latched behavior is not necessary; the checks are built into the `always_comb` construct. More details on `always_comb` and other SystemVerilog specialized procedural blocks (`always_ff` and `always_latch`) can be found the book "*SystemVerilog for Design*" [3].

Note that the IEEE SystemVerilog standard does not require a tool to detect and report latched logic functionality in an `always_comb` procedural block. At the time this paper was written, VCS does not report such warnings, but DC Compiler does.

**`unique`/`priority` decisions** — These decision modifiers require that tools check that a decision sequence (a `case` statement or an `if...else...if` statement) be completely specified. If during simulation, the decision statement is entered and no branch is taken, a run-time warning is issued. This is a useful built-in assertion that does not need to be written as a separate assert statement. In addition, the `unique` modifier requires that simulation report a warning any time two or more decision branches are true at the same time. This is another built-in assertion that designers should take advantage of. For a much more detailed description on unique and priority decisions, refer to the papers "*SystemVerilog Saves the Day—the Evil Twins are Defeated! "unique" and "priority" are the new Heroes*" [4], and *"SystemVerilog's priority & unique - A Solution to Verilog's 'full_case' & 'parallel_case' Evil Twins!"*[5].

**Enumerated types** — Enumerated types provide a mechanism to give names (called labels) to specific logic values. Enumerated types do much more than make code more readable. They also specify a legal set of values for a variable. Enumerated variables are more strongly typed than Verilog variables. It is illegal to assign an enumerated variable a value that is not in the enumerated value set. In addition, it is illegal to have two enumerated labels with the same value. The strongly typed nature of enumerated types is a form of built-in assertions that can prevent inadvertent coding errors such as assigning a state machine state variable the value of a non-existent state, or having a state value that decodes to more than one state. More details on the advantages of enumerated types can be found in the book "*SystemVerilog for Design*" [3].

### 2.5 Invariant, sequential and eventuality assertions

Languages such as C, SystemC and VHDL have an assertion construct. These constructs, however, do not have near the capabilities of SystemVerilog Assertions (neither does PSL, for that matter). The C, SystemC and VHDL assert construct is essentially the same as SVA's immediate assertion. What makes SVA unique is the ability to specify concurrent assertions that evaluate as a separate thread in parallel with the design code, and can span any number of design clock cycles. (PSL also has concurrent assertions, but lacks some features and advantages of SVA).

SystemVerilog Assertions can be categorized into three general classifications.

***Invariant assertions*** look for conditions that are always true or are never true. For example, a FIFO should never indicate full and empty conditions at the same time. An invariant assertion can

be represented using either an immediate assertion or a concurrent assertion. Most of the assertions that are specified by design engineers will probably be invariant assertions.

***Sequential assertions*** look for a set of conditions occurring in a specific order and over a specific number of clock cycles. The examples shown in section 2.3, above, are sequential assertions, where `req` should be followed by `grant` followed by `!req` followed by `!grant`.

***Eventuality assertions*** check that one condition is followed by a second condition, but with any number of clocks between the two conditions. For example, when an active-low reset goes to zero, it should eventually return to one. Eventuality assertions can also be used for monitoring handshaking signals between two asynchronous clocks. The monitoring will take place in one clock zone but will monitor signals from the other clock zone.

## 3.0 Where assertions can be specified

The initial place to start considering assertions is during the development of the design specification. The assertions specified during this phase of the design process will most likely be extensions of the design specification itself, written in assertion pseudo code. This will recommend, from a system point of view, which assertions should be defined.

The test plan is another place where assertion pseudo code should be specified. Part of defining how, and what, will be tested in the design should be defining where assertions will be used. Specifying assertions as part of the test plan is presented in much greater detail in Section 4.2 of this paper.

The actual assertions can be specified in three source code locations:

- Embedded in the RTL code
- As part of the testbench code
- In a separate file that is bound into the design

Embedding assertions into the RTL code should be the responsibility of the design engineer(s) writing the model. Verification engineers should not modify the design code, itself. These embedded assertions can be specified by the designer to ensure that the detailed implementation is correct. Most of these assertions will be immediate assertions, but there will be places where concurrent assertions will need to be used. For example, concurrent assertions can be used to verify that the correct state machine transitions occur, and to flag bad transitions. Another place where imbedded assertions should be used is in SystemVerilog interface models to ensure that design blocks that use the interface correctly use the interface protocol. Note that any assertions embedded in the RTL code will be lost during synthesis, since synthesis ignores assertions.

***Assertion binding*** allows verification engineers to add assertions to design blocks, without touching or modifying the designer's code in any way. In essence, the assertion property blocks and assert statements are in a separate testbench file, which is bound during the simulation elaboration to specific design blocks. Assertion binding is a unique, and powerful, feature of SystemVerilog. Being able to bind assertions to the design provides expanded capabilities. For instance, verification engineers can write assertions for RTL models deep in the hierarchy of the design without touching the RTL code, which will make RTL designers very happy. Another

place where binding assertions is useful is in designs with mixed Verilog and VHDL models. The SystemVerilog assertions can be bound to the VHDL models within the design. Assertion binding can also be used to add assertions to IP models embedded in the design.

Guideline — Assertions that might be useful both before and after synthesis should be external to the RTL code and then bound to the design. Using binding allows the same assertions to be used with both the RTL models and the gate level synthesized design.

## 4.0 An assertions test plan for a small DSP design

In this section, we will show how assertions can be specified as the verification test plan is developed. We will not delve into writing test plans; that is a different topic. Our focus is on figuring out where assertions should be used, and whether the designer or verification engineer should write the assertions.

A small Digital Signal Processor (DSP) design was used as a test case to illustrate how design engineers can take advantage of assertions. This small DSP is somewhat contrived; it is not a commercial or production design. The DSP is used in Sutherland HDL training courses to teach design engineers to model complete, synthesizable designs in SystemVerilog. A complete, but faulty, version of the DSP model is used in the SystemVerilog Assertions course to teach engineers how to write assertions at both the design level and the testbench level.

### 4.1 The DSP design blocks

This small DSP design contains several blocks that can benefit from assertions.

**Clock Generator** — generates two phase-synchronized clocks, and synchronizes an external reset to those clocks.

**Program Memory** — a 16-bit by 1024 word Random Access Memory (RAM) that contains the program for the DSP to execute. Each instruction in the program memory contains a 16-bit *instruction word*. An instruction word has two parts; the upper four bits represent the DSP instruction, and the lower 12 bits represent the instruction data. To simplify the design for lab time constraints, the program memory is wired for read-only activity. The logic for loading the program memory is omitted from the DSP.

**Instruction Register** — a pre-fetch register that stores the next instruction to be executed by the DSP.

**Program Counter** — a loadable counter that controls which instruction word is read from the program memory.

**Controller** — a state machine that turns on and off eleven control signals that are used throughout the processor.

**Arithmetic Logic Unit (ALU)** — executes eight operations: add, subtract, multiply, AND, OR and exclusive-OR functions, plus a no-operation and a pass-through operation. The ALU also creates a zero flag and an exception flag that continually reflect the result of the current operation.

**Decoder** — a combinational logic block that decodes the current processor instruction, and

creates an operation code (opcode) for the ALU.

**Status Register** — stores the result of the ALU operation, operation flags, and a status flag that indicates if a branching instruction is being executed.

**Data Register** — stores a data word used as an input to the ALU.

**Data Memory** — a 16-bit by 1024 word Random Access Memory (RAM) used to store data for ALU operations and operation results.

**Tri-state Buffers** — control the multi-driver bidirectional busses in the DSP.

Figure 1 contains a high-level block diagram showing how these major parts of the DSP are connected together.

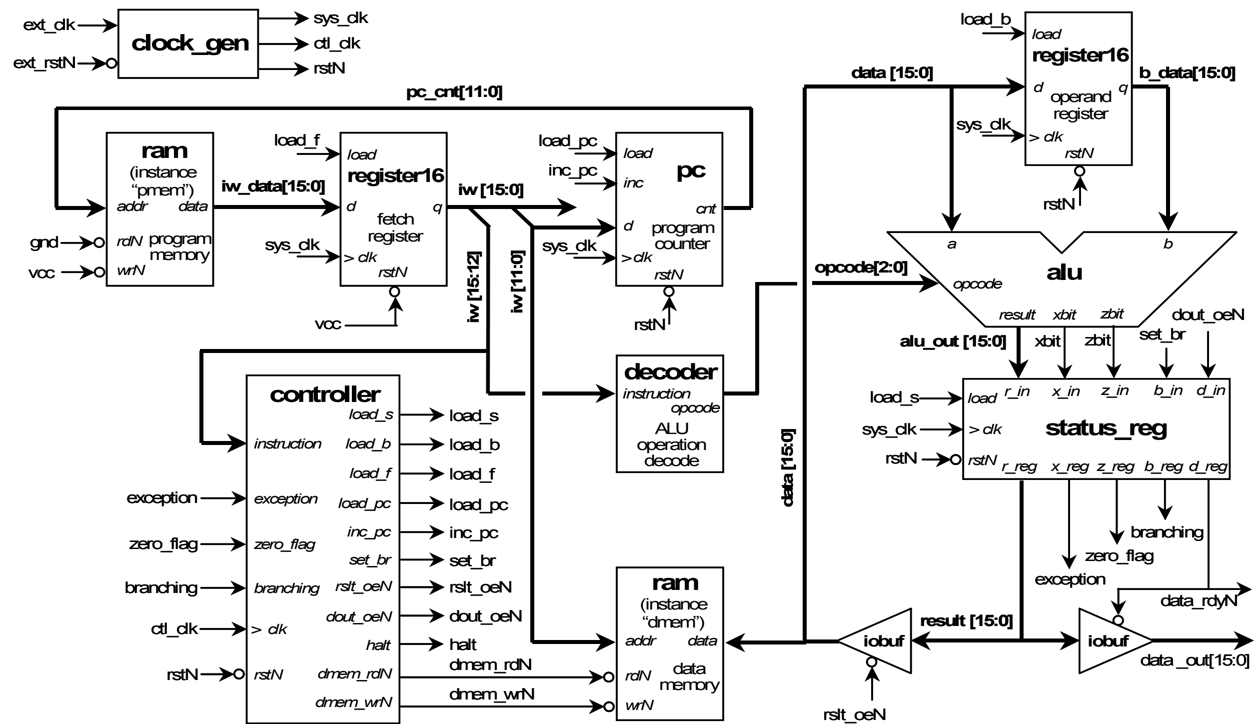


**Figure 1. Block diagram of the DSP test case**

## 4.2 An assertions plan for the DSP

As noted previously in section 3 of this paper, one of the first steps in using assertions is to develop an *assertions test plan*. This requires that the design team and the verification team sit down together and discuss the primary purpose of each block of the design. The focus of the discussion should be on what assertions will be needed to adequately verify each block. As these assertions are discussed, a decision should be made as to whether it would be best to specify the

assertion inside the design block, or external to the design block.

In addition, an initial decision should be made as to whether the assertion should be written by the designer as the RTL model is developed, or added by the verification engineer. Delegating this responsibility before any actual code is written enables both engineering teams to work in parallel, without duplicating effort. Just as important, specifying ownership for each assertion helps ensure that all assertions are written. Neither team will falsely assume the other team will write a specific assertion.

Experience has shown that the assignment of responsibility for each assertion must be flexible. The design and verification teams should meet together often during the design process to review the assertions test plan, and make adjustments if needed. For example, during the coding process, a designer might decide that a specific assertion requires the use of signals that are not part of that design block (such as access to a clock in a pure combinational logic design block). Therefore, responsibility for that assertion should be transferred to the verification team, which has access to both system level clocks and the combinational block. On the other hand, in the process of writing assertions, the verification engineer might discover that some signal names internal to the design block are required. Since those signal names are local to a design block, it would be more appropriate for the designer to specify the assertion as the design code is being written.

### 4.3 Rule of thumb for delegating assertions

A simple rule of thumb can be used for determining whether an assertion should be written by the design team as part of the design block, or specified by the verification team external to the design block. As with any rule of thumb, this is just a guideline, but it is a good place to start.

*Designer engineers should write assertions to verify assumptions that affect the functionality of a design block.* For example, the ALU block of the DSP assumes that the A, B and opcode inputs will never have a logic X or Z value. The RTL code within the ALU depends on that assumption to function properly. It is appropriate for the designer to add assertions to the design block that verify those assumptions hold true. It is a very easy assertion to write, and should the assumption prove false, the assertion failure helps quickly isolate where a functional problem arose.

*Verification engineers should write assertions that verify design functionality meets the design specification.* For example, the zero flag output of the ALU block of the DSP should always be set if the result output is zero. Again, an assertion failure will help quickly isolate the cause of a functional problem.

Not all of the assertions in the test plan will fall neatly into one of these two categories, but a majority of checks will. For those assertions that do not fall into either of these two categories, the design and verification teams will need to discuss which team gets the privilege of writing the assertion.

### 4.4 An example assertions test plan

The following tables illustrate an assertion test plan for the simple DSP design. Additional assertions could be added to this test plan, but what is shown is adequate to illustrate the intent.

**Table 1: Clock Divider / Reset Synchronization block**

| Functionality to Verify | Assertion Type | Assigned To |
|---|---|---|
| When external reset goes low, it should eventually go high again | eventuality | designer |
| ctl_clk is always the invert of ext_clk | invariant | verifier |
| After reset, a posedge of sys_clk occurs every third posedge of ext_clk | sequential | verifier |
| rstN is low whenever ext_rstN is low | invariant | verifier |
| At posedge ext_rstN, rstN goes high on the first posedge of ctl_clk before a posedge of sys_clk (must allow for gate delay skew on clock edges) | sequential | verifier |

**Table 2: RAM block**

| Functionality to Verify | Assertion Type | Assigned To |
|---|---|---|
| !rdN (read) and !wrN (write) are mutually exclusive | invariant | designer |
| address input never has any X or Z bits when writing into the RAM | invariant | designer |
| data never has any X or Z bits when writing into the RAM | invariant | designer |
| address input never has any X or Z bits when reading from the RAM | invariant | designer |
| data never has any X or Z bits when reading from the RAM | invariant | designer |

**Table 3: Program Counter block**

| Functionality to Verify | Assertion Type | Assigned To |
|---|---|---|
| load and increment are mutually exclusive | invariant | designer |
| If increment, then d input never has any X or Z bits | invariant | designer |
| If !load and !increment, then on posedge of clock, pc does not change (must allow for clock-to-q delay) | sequential | verifier |
| If increment, then pc increments by 1 (must allow for clock-to-q delay) | sequential | verifier |
| If load, then pc == d (must allow for clock-to-q delay) | sequential | verifier |

## Table 4: Decoder block

| Functionality to Verify | Assertion Type | Assigned To |
|---|---|---|
| After reset, instruction input never has any X or Z bits | invariant | designer |
| All instructions decoded | unique case | designer |
| If pass instruction (4’b1000), then ALU opcode == pass (3’b001) | invariant | verifier |
| If arithmetic and logical instructions (instruction[3] == 0), then ALU opcode == instruction[2:0] | invariant | verifier |
| If move and branch instructions (instruction == hex 9 through hex F), then opcode == no-op (3’b000) | invariant | verifier |

## Table 5: ALU block

| Functionality to Verify | Assertion Type | Assigned To |
|---|---|---|
| After reset, the A input never has any X or Z bits | invariant | designer |
| After reset, the B input never has any X or Z bits | invariant | designer |
| After reset, the opcode input never has any X or Z bits | invariant | designer |
| All instructions decoded | unique case | designer |
| zbit always set if result == 0 | invariant | verifier |
| zbit never set if result != 0 | invariant | verifier |
| After overflow/underflow, xbit cleared on next operation if no overflow/underflow | sequential | verifier |
| Add/multiply operation overflows when the A and B input MSB's are both set | sequential | verifier |
| Subtract operation underflows when the A input is less than the B input | invariant | verifier |
| Non-arithmetic operations never causes an exception | invariant | verifier |

## Table 6: Data Register block

| Functionality to Verify | Assertion Type | Assigned To |
|---|---|---|
| If !load, then at each clock, q does not change | sequential | verifier |
| If load, then after posedge clock, q == d (must allow for gate clock-to-q delay) | sequential | verifier |

**Table 7: Status Register block**

| Functionality to Verify | Assertion Type | Assigned To |
|---|---|---|
| If !load, then after posedge clock, r_reg, x_reg, z_reg do not change | sequential | verifier |
| If load, then after posedge clock, r_reg == r_in (allow for gate clock-to-q delay) | sequential | verifier |
| If load, then after posedge clock, x_reg == x_in (allow for gate clock-to-q delay) | sequential | verifier |
| If load, then after posedge clock, z_reg == z_in (allow for gate clock-to-q delay) | sequential | verifier |
| after posedge clock, b_reg == b_in (allow for gate clock-to-q delay) | sequential | verifier |
| after posedge clock, d_reg == d_in (allow for gate clock-to-q delay) | sequential | verifier |

**Table 8: Controller block**

| Functionality to Verify | Assertion Type | Assigned To |
|---|---|---|
| After reset, instruction input never has any X or Z bits | invariant | designer |
| At posedge of clock, state is always one-hot | invariant | designer |
| If !rstN, state == reset | sequential | verifier |
| If in decode state, prior state was reset or store | sequential | verifier |
| If in load state, prior state was decode | sequential | verifier |
| If in store state, prior state was load | sequential | verifier |

**Table 9: IO Buffer block**

| Functionality to Verify | Assertion Type | Assigned To |
|---|---|---|
| If !en, all bits of output should be tri-stated | sequential | verifier |

**Table 10: Top-level block**

| Functionality to Verify | Assertion Type | Assigned To |
|---|---|---|
| !dmem_rdN and !rslt_oeN are mutually exclusive | invariant | verifier |
| After reset, data bus never has any X or Z bits | invariant | verifier |
| After reset, PC d input == lower 12 bits of IW | invariant | verifier |
| After reset, decoder instruction == upper 4 bits of IW | invariant | verifier |
| After reset, controller instruction == upper 4 bits of IW | invariant | verifier |
| load_s, load_b and load_pc are mutually exclusive | invariant | verifier |
| load_f and load_pc are mutually exclusive | invariant | verifier |
| inc_pc and load_pc are mutually exclusive | invariant | verifier |
| !dmem_rdN and !dmem_wrN are mutually exclusive | invariant | verifier |

## 5.0 Example assertions per the assertion test plan

The following code snippets illustrate a few of the assertions design engineers can, and should, add to the design code. These examples come from the assertions test plan listed in section 4.4, above.

### 5.1 ALU block: No X on inputs; All opcodes decoded

```
always_comb begin
  // Assertions to test for bad inputs (no X or Z bits)
  ai_a_never_x:   assert (^a !== 1'bx);
  ai_b_never_x:   assert (^b !== 1'bx);
  ai_opc_never_x: assert (^opcode !== 1'bx);

  overflow = 0;    //assume no overflow by default
  unique case (opcode)
  ALU_NOP   : result = 0;
  ALU_PASS  : result = a;
  ALU_ADD   : {overflow[0], result} = (a + b);
  ALU_SUB   : {overflow[0], result} = (a - b);
  ALU_MULT  : {overflow, result}    = (a * b);
  ALU_AND   : result = (a & b);
  ALU_OR    : result = (a | b);
  ALU_XOR   : result = (a ^ b);
  endcase
end
```

The three assert statements are invariant immediate assertions. They verify the design assumption

that the inputs to the ALU are valid (no bits with X or Z values) each time the procedural block is entered. ***That was easy, wasn't it!***

To test that all opcodes are decoded, the `unique` modifier is added to the case statement. This modifier has built-in assertion behavior. Simulation is required to report a warning if the case statement is entered, and no branch is taken. Simulation is also required to report a warning if it detects that two or more branches are true at the same time. ***This was an easy check to add to the RTL code, too!***

Two other assertion checks are also illustrated in this code example. The `always_comb` procedural block informs software tools that the designer's intent is that this block of code represent combinational logic. Should a coding mistake cause latched logic behavior, tools can warn that the code does not match the designer's intent. (At the time this paper was written, VCS did not provide this warning, but DC Compiler does.) The opcode names are defined as an enumerated type (not shown in the code snippet, above). Enumerated types have built-in syntactic assertions that prevent two opcode names being inadvertently defined with the same opcode value.

### 5.2 RAM: Mutually exclusive read/write control lines

```
// assertion to check that read and write are never low at the same time
always @(rdN or wrN)
  #0 ai_read_write_mutex: assert (!(!rdN && !wrN));
```

This simple assertion is written as an invariant immediate assertion. A mutually exclusive assertion such as this would normally be written as a concurrent assertion that runs as a separate thread in parallel to the model's RTL code. However, concurrent assertions require a clock, and since this model is an asynchronous RAM, there is no clock. Therefore the assertion is written as a separate `always` procedural block. Synthesis will ignore the assert statement, making this an empty procedural block, which will also be ignored by synthesis.

### 5.3 Controller state machine: At posedge of clock, state is always one-hot

```
// FSM state should always be one-hot
property p_fsm_onehot;
  @(posedge clk) disable iff (!rstN) ($onehot(state));
endproperty
ap_fsm_onehot: assert property (p_fsm_onehot);
```

This test is modelled as an invariant concurrent assertion. It executes as a parallel thread with the RTL code. At every positive edge of clock, the assertion verifies that the value of the `state` variable has one, and only one, bit set.

Section 6.7, in "Lessons learned", discusses some additional considerations for this assertion.

### 5.4 Controller state machine: Proper state sequencing

```
// verify asynchronous reset to RESET state
property p_fsm_reset;
  @(posedge clk) !rstN |-> state == RESET;
```

```
endproperty
ap_fsm_reset: assert property (p_fsm_reset);

// verify DECODE state was entered from RESET or STORE state
property p_fsm_decode_entry;
  @(posedge clk) disable iff (!rstN)
    state == DECODE |-> $past(state) == RESET || $past(state) == STORE;
endproperty
ap_fsm_decode_entry: assert property (p_fsm_decode_entry);

// verify LOAD state was entered from DECODE state
property p_fsm_load_entry;
  @(posedge clk) disable iff (!rstN)
    state == LOAD |-> $past(state) == DECODE;
endproperty
ap_fsm_load_entry: assert property (p_fsm_load_entry);

// verify STORE state was entered from LOAD state
property p_fsm_store_entry;
  @(posedge clk) disable iff (!rstN)
  state == STORE |-> $past(state) == LOAD;
endproperty
ap_fsm_store_entry: assert property (p_fsm_store_entry);
```

The requirement for these assertions is that they look back in simulation time to see previous values of the `state` variable. In Verilog, this would require extra code to preserves past values of the variable. SystemVerilog Assertions makes this an easy test to write. The **`$past`** function is provided, which automatically looks back a specified number of clock cycles (the default is one cycle, which is what is used in these examples) and returns the previous value of a net or variable.

## 6.0 Lessons learned

Developing assertions for this small DSP design illustrates a number of important lessons on Assertion Based Verification. As noted earlier in this paper, the DSP case study used in this paper is a training lab. The lab is used in two ways: for designers to learn to write synthesizable RTL models, and for design and verification engineers to learn to write SystemVerilog assertions.

Some of the lessons learned form the dual use of this DSP case study are listed in this section.

### 6.1 Developing the assertions test plan takes time

The design and verification teams should plan to invest a fair amount of time before any RTL coding begins developing the assertions test plan. This is not a trivial task, but it is critical to the success of assertion based verification. From the design engineer's perspective, the assertion test plan is important, because it is the guideline on what assertions the designer is responsible to write.

### 6.2 The assertion test plan helps identify similar assertions

As the assertion test plan is developed, it becomes obvious that some assertions are essentially the same for each design block. These assertions are prime candidates for the use of property blocks.

A generic property block can be used for multiple assertions. This greatly reduces the amount of time it takes to write the assertions.

### 6.3 Take advantage of SystemVerilog constructs with built-in assertions

The DSP design used as a case study for this paper has been used as a training lab in Sutherland HDL training courses for several years. The lab is a final project, giving engineers an opportunity to use all of the principles presented during the course to model a complete design. Students new to Verilog often make simple modeling mistakes, such as incomplete case statements and inadvertent latches in what should be combinational logic. In our Sutherland HDL training courses, we encourage engineers to use the SystemVerilog constructs that have built-in assertions, such as `unique`/`priority` case statements, enumerated types, and `always_comb`/`always_ff` procedural blocks. We have seen the use of SystemVerilog constructs with built-in assertions dramatically reduce the time it takes even the most novice of engineers to model the DSP design and get the model to correctly simulate and synthesize.

The assertions test plan should specify critical areas of the design where SystemVerilog constructs with built-in assertions should be used.

### 6.4 An assertion should not just duplicate RTL functionality

RTL code causes specific output changes. An assertion should monitor those changes, and verify what caused the change.

Designers are prone to writing assertions that duplicate RTL code. It is natural for designers to think along the lines that an action should cause an effect, and write assertions in the same way they write RTL code. For example, an assertion that verifies "*if the ALU result is zero, then the zero flag should be set*" only duplicates the RTL logic. Assertions should be written to test that an affect should be the result of a cause. A better assertion is for the zero flag is *"if zero flag is set, then result should be zero*".

One of the observations of the authors is that most designers will go through a learning curve in order to learn to think about writing assertions differently than they write RTL logic.

### 6.5 Disabling assertions during resets is important

Concurrent assertions, and immediate assertions in combinational logic, will begin executing at the very start of simulation. Before a design has been reset, and while it is being reset, a design is probably not in a known, stable condition. Dozens, possibly hundreds of false assertion failures can be reported prior to reset completing. These false failures could hide a real assertion failure. It is important to prevent the clutter of false assertion failures.

As discussed earlier in this paper, in sections 2.1 and 2.3, SystemVerilog makes it easy to disable assertions until the design reaches a stable point.

### 6.6 Need to be flexible on who writes the assertions

It is important to keep some flexibility in the dividing line between assertions the designer should write and assertions the verification engineer should write. In the development of the assertions

test plan, the dividing line can seem obvious, but students defining assertions for the small DSP design frequently encounter many situations where an assertion that seemed logical for the designer to write as the model is developed turned out to be difficult or ineffective at that level. The ALU block of the DSP is a prime example. The ALU block is pure combinational logic, which does not have a clock. Since concurrent assertions are cycle based, a clock is required. In order to concurrently test ALU inputs and functionality using assertions, the assertion needs to be written outside of the ALU block, where it has access to the clock and the internal signals within the ALU. The assertions test plan needs to allow the design engineer to re-assign the task of writing the assertion back to the verification team.

### 6.7 Testing logic that has enumerated types requires access to the enumerated names

The DSP controller is a state machine. The state names are typically modeled as either parameter constants or enumerated type labels. In traditional Verilog/SystemVerilog modeling styles, these state names are declared within the state machine module. Since the state names are local to the state machine module, a verification engineer cannot write assertions on state transitions until the state names are defined. This means the verification team cannot develop those assertions in parallel with the design team. In addition, a verification engineer cannot write the assertions outside of the state machine module, because the state labels do not exist outside of the design block. The assertions must be embedded in the design module in order to use the locally declared state names.

The solution to this dilemma—and a good coding guideline to adopt—requires two changes to how designers use enumerated types:

a. Make the names to be used for enumerated type labels, such as the state names of a state machine, part of the design specification. In this way, both the designer and the verification engineer know up front what names will be used within a module block.
b. Move the definition of the enumerated label names to a package, outside of the state machine module. This gives access to these names in both the design module and in verification blocks.

### 6.8 Enumerated type labels disappear after synthesis

Enumerated types provide a mechanism to give labels (names) to specific logic values. A common use of enumerated types is to label state machine names, ALU opcodes, and processor instructions. An abstract enumerated type definition does not specify the hardware logic value for each label. The definition of the actual logic values in a gate-level implementation is left to synthesis compilers. An example of a typical enumerated type definition from the DSP case study is:

```
typedef enum {RESET, DECODE, LOAD, STORE} states_t;

states_t state, next_state;  // state and next state have labeled values
```

Assertions on the state machine can be written using these enumerated value labels. However, after the design has been synthesized, the gate-level design has arbitrary, synthesis-selected logic values; and the assertions using enumerated labels are no longer valid.

SystemVerilog has the capability to define specific values for each enumerated label at the RTL level, before synthesis (VHDL, C and SystemC do not have this capability). For example:

```
typedef enum logic [3:0] {
  RESET  = 4'b0001,
  DECODE = 4'b0010,
  LOAD   = 4'b0100,
  STORE  = 4'b1000
} states_t;

states_t state, next_state;  // state and next state have one-hot values
```

Now the values of each enumerated label are known prior to synthesis. Synthesis compilers can be instructed to keep these same values in the gate level implementation of the design (the default behavior for DC Compiler). An assertion written using the enumerated labels will now work with at both the RTL level and the post-synthesis gate level of the design.

As a guideline, enumerated types that will be used in both the design and in assertions should always define the actual values for each enumerated label.

### 6.9 Assertion based verification may require different design partitioning

Concurrent assertions are cycle based, and require a clock definition. However, pure combinational logic does not have a clock. In addition, it is desirable to disable most assertions during reset, but pure combinational logic does not have a reset. The specification for the DSP design has two design blocks that are pure combinational logic, the ALU and Decoder. In order to effectively use assertions to verify these design blocks, the design specification should re-partition the design to combine the combinational logic and the sequential logic that registers the ALU results (the status register) into a single partition.

## 7.0 Conclusions

Assertions are not just for verification engineers! Design engineers can, and should, learn to take advantage of SystemVerilog assertions. Designers should embed assertions into their design code as the models are being written. Specifically, designers should add an assertion for any assumption about design conditions on which the model depends. At a minimum, this includes assertions that inputs to each module are valid values.

An assertion test plan is critical to assertion-based verification. The test plan defines what assertions need to be written, and whether the designer or the verifier is responsible for writing the assertion. This up-front delegation of responsibility ensures that all assertions are written, and than no assertions are missed.

Design engineers need to think differently when writing assertions. RTL code monitors input values and causes output changes. Assertions should monitor output values, and verify what caused the change. There is a learning curve involved in adopting assertion-based verification.

Design engineers should take full advantage of the SystemVerilog constructs that have built-in assertions. These include specialize always procedural blocks, enumerated types, and unique/priority decisions.

The case study of a small DSP design presented in this paper is used as a final project in Sutherland HDL Verilog/SystemVerilog training courses. The authors have observed many common modeling mistakes engineers learning Verilog often make. The assertions test plan presented in this paper effectively traps nearly all common design mistakes novice designers make on this final project. Without assertions, these mistakes are difficult to debug and isolate to the root cause. Assertions have proven to be very effective at detecting functional problems in a design, and pointing to the exact cause of the problem. Assertions can reduce troubleshooting time from a half hour or more to just minutes.

In discussions during training classes, the authors have often encountered skepticism from design engineers regarding adding assertions to their design code. Some of the reasons given for this hesitancy are: assertions are too hard to write, assertions will convolute the RTL code, adding assertions could possibly introduce bugs in the RTL logic, assertions will take too long to write, and assertions will slow down simulation run-times.

As noted earlier in this paper, three major advantages to having designers embed some assertion checking into the RTL code are:

- SystemVerilog makes it easy to write basic assertions (complex sequences that utilize SVA operators can be more difficult to write, but those types of assertions are usually written by the verification team).
- SystemVerilog Assertions are ignored by synthesis; there is no need to convolute the code with `translate_off`/`translate_on` synthesis pragmas.
- SystemVerilog Assertions can be disabled at any time during simulation, on a global basis, on a per module basis, or on a specific assertion basis. Disabling assertions can eliminate false failures during reset or other conditions where parts of the design are not stable. Disabling assertions can also be used to improve simulation run-time performance when some, or all, assertion checks are not needed.

When time is invested to write a good assertion test plan, adding assertions to RTL code is very simple. ***SystemVerilog Assertions really are for design engineers, too!***

## 8.0 References


[1] *"IEEE P1800-2005 standard for the SystemVerilog Hardware Description and Verification Language, ballot draft (D4)"*, IEEE, Pascataway, New Jersey, 2001. ISBN TBD. The P1800 standard was not yet published at the time this paper was written, but the P1800 standard is based on the Accellera SystemVerilog 3.1a standard, which it is available at www.accellera.org.

[2] "*Being Assertive With Your X*", by Don Mills. Published in the proceedings of SNUG San Jose, 2004

[3] "*SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling", by* Stuart Sutherland, Simon Davidmann and Peter Flake. Published by Springer, Boston, MA, 2004, ISBN: 0-4020-7530-8.

[4] "*SystemVerilog Saves the Day—the Evil Twins are Defeated! "unique" and "priority" are the new Heroes*", by Stuart Sutherland. Published in the proceedings of SNUG San Jose, 2005.

[5] "*SystemVerilog's priority & unique - A Solution to Verilog's 'full_case' & 'parallel_case' Evil Twins!*", by Clifford Cummings. Published in the proceedings of SNUG Israel, 2005.

## 9.0 About the authors

Mr. Don Mills has been involved in ASIC design since 1986. During that time, he has work with more than 30 ASIC projects. Don started using top-down design methodology in 1991 (Synopsys DC 1.2). Don has developed and implemented top-down ASIC design flow at several companies. His specialty is integrating tools and automating the flow. Don is an independent consultant, and is a certified instructor of Verilog Courses for Sutherland HDL, Inc. and Sunburst Design, Inc. Don has authored and co-authored numerous papers, such as *"How to Synthesize a Million Gate ASIC"* and *"RTL Coding Styles that Yield Simulation and Synthesis Mismatches"*. Copies of these papers can be found at www.lcdm-eng.com. Mr. Mills can be reached at mills@lcdm-eng.com.

Mr. Stuart Sutherland is a member of the IEEE P1800 working group that is defining SystemVerilog, and is the technical editor of the SystemVerilog Reference Manual. He has been involved with the definition of the SystemVerilog standard since work began in 2001. He is also a member of the IEEE 1364 Verilog standards working group. Stuart is an independent Verilog consultant, specializing in providing comprehensive expert training on the Verilog HDL, SystemVerilog and PLI. Stuart is a co author of the book *"SystemVerilog for Design"* and is the author of *"The Verilog PLI Handbook"*. He has also authored a number of technical papers on Verilog and SystemVerilog, which are available at www.sutherland-hld.com/papers. You can contact Stuart at stuart@sutherland-hdl.com.