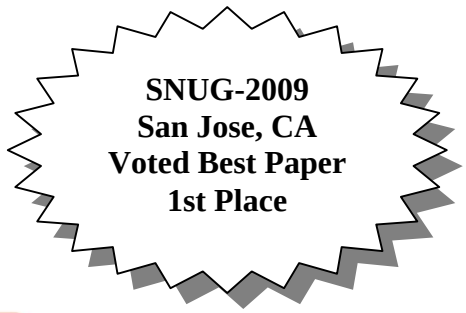


World Class Verilog & SystemVerilog Training

SystemVerilog Assertions Design Tricks and SVA Bind Files

Clifford E. Cummings
Sunburst Design, Inc.
cliffc@sunburst-design.com
www.sunburst-design.com

ABSTRACT

The introduction of SystemVerilog Assertions (SVA) added the ability to perform immediate and concurrent assertions for both design and verification, but some engineers have complained about SVA verbocity or do not understand some of the better methodologies to take full advantage of SVA.

This paper documents valuable SystemVerilog Assertion tricks, including: use of long SVA labels, use of the immediate assert command, concise SVA coding styles, use of SVA bind files, and recommended methodologies for using SVA.

The concise SVA coding styles detailed in this paper can reduce concurrent SVA coding efforts by 50%-80% over conventional SVA coding techniques.

Table of Contents

Table of Figures

Table of Examples

1 Introduction

As I have watched the enthusiasm and growing interest in SystemVerilog Assertions (SVA) over the past five years, I have witnessed multiple design teams who have taken SVA training, embraced the potential for rapid design and debug using SVA, but who have later largely abandoned the use of SVA due to the perceived verbose nature regarding the creation and implementation of SystemVerilog assertions. Over the past three years, I have made it a priority to develop SVA usage techniques that even design engineers would adopt. This paper details some SVA methodology techniques that I highly recommend, especially for design engineers.

There are some simple tricks that every design engineer should know to facilitate the usage of SystemVerilog Assertions.

Although this paper is not intended to be a comprehensive tutorial on SystemVerilog Assertions, it is worthwhile to give a simplified definition of a property and the concurrent assertion of a property.

1.1 What is an assertion?

An assertion is basically a *"statement of fact"* or *"claim of truth"* made about a design by a design or verification engineer. An engineer will assert or "claim" that certain conditions are always true or never true about a design. If that claim can ever be proven false, then the assertion fails (the "*claim*" was false).

Assertions essentially become active design comments, and one important methodology treats them exactly like active design comments. More on this in Section 2.

A trusted colleague and formal analysis expert[1] reports that for formal analysis, describing what should never happen using "not sequence" assertions is even more important than using assertions to describe always true conditions.

1.2 What is a property?

A property is basically a rule that will be asserted (enabled) to passively test a design. The property can be a simple Boolean test regarding conditions that should always hold true about the design, or it can be a sampled sequence of signals that should follow a legal and prescribed protocol.

For formal analysis, a property describes the environment of the block under verification, i.e. what is legal behavior of the inputs.

1.3 Two types of SystemVerilog assertions

SystemVerilog has two types of assertions:
(1) Immediate assertions
(2) Concurrent assertions

Immediate assertions execute once and are placed inline with the code. Immediate assertions are not exceptionally useful except in a few places, which are detailed in Section 3.

Concurrent assertions are the most valuable and most widely used type of assertion. Concurrent assertions are either placed directly in the RTL code or are bound to an RTL file using the **bind** command (see Section 6). Concurrent assertions activate properties (rules) that typically sample design signals or sequences of design signals just before each new active clock edge to determine if the design is behaving as it was claimed that it should behave.

Engineers have been adding assertions to their designs for years, but they often called them monitors and they were often placed in **always** blocks to sample at fixed intervals or only when certain signals changed.

Regarding formal analysis, immediate assertions are only valid for simulation but concurrent assertions are useful both for formal and simulation.

2 Long Labels

Adding labels to concurrent assertions is optional, but highly recommended. The long labels help to debug the assertions in a waveform display.

To demonstrate the effectiveness of using long labels when debugging a design, assume that an exceptionally incompetent engineer has coded a very flawed D-flip-flop as shown in Example 1:

```
module dff (
  output logic q,
  input        d, clk, rst_n);

  assign q = d; // This is clearly a mistake!!
endmodule
```

Example 1 - Incorrectly coded D-flip-flop model

To this **dff** module, let's first add a concurrent assertion with no label, but we will include an SVA action block with an error message that will display when the assertion fails, as shown in Example 2.

```
assert property (@(posedge clk) disable iff (!rst_n) (q==$past(d)))
else $display("ERROR: q did not follow d");
```

Example 2 - Assertion with $display command but no label

Using VCS, when the simulation fails, an error message will be displayed to the computer screen as shown on multiple lines of the output as shown in Figure 1.

```
  0ns: clk=0  rst_n=0  d=1  q=1
  5ns: clk=1  rst_n=0  d=1  q=1
 10ns: clk=0  rst_n=1  d=1  q=1
 15ns: clk=1  rst_n=1  d=1  q=1
 20ns: clk=0  rst_n=1  d=1  q=1
 25ns: clk=1  rst_n=1  d=1  q=1
 30ns: clk=0  rst_n=1  d=0  q=0
```

```
"sva_ex01.sv", 20: sva_ex01.unnamed$$_1: started at 35ns failed at 35ns
    Offending '(q == $past(d))'
ERROR: q did not follow d
 35ns: clk=1  rst_n=1  d=0  q=0
 40ns: clk=0  rst_n=1  d=1  q=1
"sva_ex01.sv", 20: sva_ex01.unnamed$$_1: started at 45ns failed at 45ns
    Offending '(q == $past(d))'
ERROR: q did not follow d
 45ns: clk=1  rst_n=1  d=1  q=1
...
```

Figure 1 - Monitored output from model with assertion $display command but no label

The waveform display for this same simulation is shown in Figure 2.

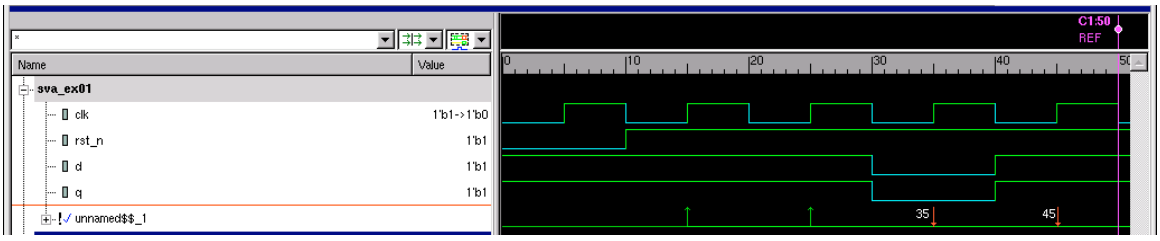


Figure 2 - Waveform display of failing assertion ($display command not visible)

During simulation, the assertion fails, but when the simulation is viewed in a waveform display, the **`$display`** error message is not visible. Only a non-descriptive generic name for the assertion is visible. Engineers looking at the waveform display of the first failing assertion will be unable to identify the problem until they consult the original source code.

Now let's add the assertion with long descriptive label shown in Example 3 to the **`dff`** source code of Example 1.

```
ERROR_q_did_not_follow_d:
  assert property (@(posedge clk) disable iff (!rst_n) (q==$past(d)));
```

Example 3 - Assertion command with long descriptive label

When the simulation fails, error messages will be displayed to the computer screen as shown on the output in Figure 3. Note how the label is included in the default error message generated by the assertion.

```
  0ns: clk=0  rst_n=0  d=1  q=1
  5ns: clk=1  rst_n=0  d=1  q=1
 10ns: clk=0  rst_n=1  d=1  q=1
 15ns: clk=1  rst_n=1  d=1  q=1
 20ns: clk=0  rst_n=1  d=1  q=1
 25ns: clk=1  rst_n=1  d=1  q=1
 30ns: clk=0  rst_n=1  d=0  q=0
```

```
"sva_ex01.sv", 17: sva_ex01.ERROR_q_did_not_follow_d: started at 35ns
failed at 35ns
    Offending '(q == $past(d))'
 35ns: clk=1  rst_n=1  d=0  q=0
 40ns: clk=0  rst_n=1  d=1  q=1
"sva_ex01.sv", 17: sva_ex01.ERROR_q_did_not_follow_d: started at 45ns
failed at 45ns
    Offending '(q == $past(d))'
 45ns: clk=1  rst_n=1  d=1  q=1
...
```

Figure 3 - Monitored output from model with long labeled assertion

The waveform display for this same simulation is shown in Figure 4.

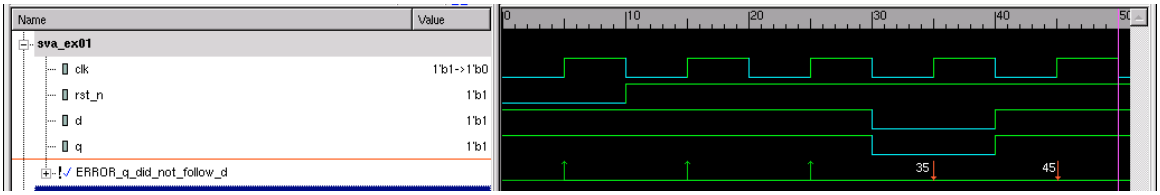


Figure 4 - Waveform display of failing assertion (descriptive assertion label is visible)

During simulation, the assertion code of Example 3 would fail, but now when the simulation results are viewed in a waveform display, the long and descriptive label name will be visible to document the failing behavior of the labeled assertion.

Contrasting the assertion code of Example 2 with the assertion code of Example 3, both would fail and report errors, but when the simulation results from both assertion styles are viewed in separate waveform displays, the **`$display`** error message of Example 2 will not be visible while the long and descriptive label name of Example 3 will be visible to quickly help the engineer identify and debug the problem.

Since these long label names are visible in a waveform display, it is also a good idea to use a label naming convention. The naming convention that I use starts each label with "**`ERROR_`**" followed by a description of what the error is if the assertion fails.

I have watched engineers for the past five years use SVA in their designs and the power-users consistently add long labels to document the intent of the assertions while simultaneously making the intended assertions visible in a waveform display to aid the debugging effort.

I tell design engineers to think of the labels as a comment that describes the purpose of the assertion that will show up in the waveform display if the assertion fails. The assertion with label has simultaneously become a monitor with active design comment.

3 Immediate Assertions

Immediate assertions as defined in the IEEE Std 1800-2005 are not exceptionally useful in a SystemVerilog design or verification environment, but there are a couple of places where they can be quite useful.

3.1 Casting

There are two forms of casting in SystemVerilog (1) ***Static-Speed casting***, and (2) ***Dynamic-Safe casting***. In this paper, the terms *static-speed casting* and *static casting* will be used interchangeably, while the terms *dynamic-safe casting* and *dynamic casting* will also be used interchangeably.

The SystemVerilog Standardization Committee introduced the two distinct styles of casting because vendors explained that the type and boundary checking required by dynamic casting could seriously degrade simulation performance. By allowing two forms of casting, engineers that were interested in higher performance simulations and who were also confident that all of the casting in their code was type and boundary-safe could choose to use the faster static casting, while engineers concerned about type and boundary checks could use the slower dynamic casting.

Consider the type definition of the **`valid_e`** enumerated type, and the declaration of the **`valid_bit`**, declared to be of the **`valid_e`** enumerated type as shown in Example 4

```
typedef enum {good, bad} valid_e;
valid_e                  valid_bit;
```

Example 4 - Enumerated valid_e typedef and valid_bit declaration

In SystemVerilog, enumerated types are a strongly-typed type when used as the target of an assignment so it is illegal to make direct integer assignments to an enumerated variable.

3.1.1 Static casting

The SystemVerilog static cast uses a data type to cast an argument enclosed within parentheses to the new data type and then assigns the cast-modified value to a target of the same or compatible type. In Example 5. The there are two legal and one illegal static cast assignments.

```
initial begin
  valid_bit = valid_e'(0);
  $display("static cast: 0->valid_bit=%0d", valid_bit,
           "  name=%s", valid_bit.name());
  valid_bit = valid_e'(1);
  $display("static cast: 1->valid_bit=%0d", valid_bit,
           "  name=%s", valid_bit.name());
  valid_bit = valid_e'(2);    // <----- illegal cast value
  $display("static cast: 2->valid_bit=%0d", valid_bit,
           "  name=%s", valid_bit.name());
end
```

Example 5 - Static cast example

All three assignments actually do complete but when an attempt is made to execute the third **$display** command, the valid bit value of 2 is shown while the name value is blank in the display since the assigned value is out of the legal enumerated range. Any attempt to use the **valid_bit** after the third assignment will probably give unintended and unexpected results.

Since a static cast can never be used as a system function (static cast can not return a pass-fail status code), there is no reason to attempt to query the success of the static cast operation by use of assertion or some other means. If we want to conduct a casting operation that allows us to query the success of the cast, we should use the dynamic cast.

3.1.2 Dynamic casting

The SystemVerilog dynamic cast system call takes two arguments. The dynamic cast assigns the second argument to the first argument and in the process of making the assignment, converts the second argument into the type of the first argument.

In Example 6, three dynamic cast statements are used to cast and assign the second arguments (integers **0**, **1** & **2**) to the first arguments (**valid_bit** of **valid_e** enumerated type). The first two assignments are legal but the third assignment is illegal since the only legal enumerated values are 0 and 1.

```
initial begin
  $cast(valid_bit, 0);
  $display("valid_bit=%s", valid_bit.name());
  $cast(valid_bit, 1);
  $display("valid_bit=%s", valid_bit.name());
  $cast(valid_bit, 2);    // <----- illegal cast value
  $display("valid_bit=%s", valid_bit.name());
end
```

Example 6 - Dynamic cast - $cast used as a system task

In the dynamic cast shown in Example 6, **$cast** is used as a system task (no return value) and hence if the cast fails, there will be a run-time simulation error and the destination variable will remain unchanged. The **$cast** can also be used as a system function that will return a pass value=1 or a fail value=0. The modified assignments shown in Example 7 use **$cast** as a system function.

```
initial begin
  if ($cast(valid_bit, 0)==0) $display("casting error");
  $display("valid_bit=%s", valid_bit.name);
  if ($cast(valid_bit, 1)==0) $display("casting error");
  $display("valid_bit=%s", valid_bit.name);
  if ($cast(valid_bit, 2)==0) $display("casting error"); // <-- illegal
  $display("valid_bit=%s", valid_bit.name);
end
```

Example 7 - Dynamic cast - $cast used as a system function and tested with if-statement

In Example 7, the last dynamic cast **`if ($cast(valid_bit, 2)==0)...`** will fail, therefore the enclosing if-statement will pass and the corresponding **`$display`** statement will be printed.

The if-tests shown in Example 7 can also be written in more concise form as shown in Example 8 and will yield the exact same results.

```
initial begin
  if (!($cast(valid_bit, 0))) $display("casting error");
  $display("valid_bit=%s", valid_bit.name);
  if (!($cast(valid_bit, 1))) $display("casting error");
  $display("valid_bit=%s", valid_bit.name);
  if (!($cast(valid_bit, 2))) $display("casting error"); // <-- illegal
  $display("valid_bit=%s", valid_bit.name);
end
```

Example 8 - Dynamic cast - $cast used as a system function and tested with concise if-statement

I refer to these coding styles as the "if-error-display-message" coding styles.

3.1.3 Dynamic casting with immediate assertion

The SystemVerilog immediate assertion style (shown in Example 9) offers a useful replacement for the if-error-display-message coding styles.

```
initial begin
  ERROR_bad_valid_bit_cast0: assert ($cast(valid_bit, 0));
  $display("valid_bit=%s", valid_bit.name);
  ERROR_bad_valid_bit_cast1: assert ($cast(valid_bit, 1));
  $display("valid_bit=%s", valid_bit.name);
  ERROR_bad_valid_bit_cast2: assert ($cast(valid_bit, 2)); // <-- illegal
  $display("valid_bit=%s", valid_bit.name);
end
```

Example 9 - Dynamic cast - $cast used as a system function and tested with an immediate assert

The first assertion in Example 9 is read as, "assert that the dynamic cast of 0 to the **`valid_bit`** is legal." If this assertion fails, an error message is printed that will include the long assertion label and the simulation will continue to execute. Since this is a SystemVerilog assertion, one can also modify the error-handling behavior by adding an action block to an else condition to display a different message or to execute other SystemVerilog commands, including **`$fatal`**, **`$error`**, **`$warning`** or **`$info`**.

As shown in the preceding examples, the dynamic cast keyword **`$cast()`** in SystemVerilog can be used as either a system task or a system function.

3.2 Randomization

A common testbench activity is to randomize class variables. The built-in **`randomize()`** method is frequently misunderstood by the new user. Consider the **`TestVars`** class definition with two randomizable variables, **`addr`** and **`data`** as shown in Example 10. This class also

includes **constraint TestVars_c1** that constrains random **addr** variables to be greater than **7**. The class also overrides the built-in **post_randomize** method, which has been included to display the class variables after randomization.

```
class TestVars;
  rand bit [3:0] addr;
  rand bit [3:0] data;

  constraint TestVars_c1 { addr > 7; }

  function void post_randomize;
    $display("addr=%h   data=%h", addr, data);
  endfunction
endclass
```

Example 10 - TestVars class definition

A common new user mistake is shown in the **initial** block labeled **block1** in Example 11.

```
TestVars B1=new;

initial begin: block1
  repeat(10) B1.randomize();  // not technically legal
end
```

Example 11 - Illegal use of randomize() method

An engineer has tried to randomize the class variables by calling the **randomize()** method as if it were task or void function. Some simulators warn the user that an implicit void-cast will be performed, which is equivalent to doing a static-void cast as shown in Example 11.

```
repeat(10) void'(B1.randomize());
```

Example 12 - Void-cast of randomize() method

The **randomize()** method always completes whether the randomization passes (positive return value) or whether the randomize fails (0). The implicit void-cast of the randomization is not desirable since it can hide a randomization failure.

```
TestVars B2=new;

initial begin: block2
  repeat(10) if(!(B2.randomize()))
      $display("B2 randomization failed");
end
```

Example 13 - If-test of randomize() method

In Example 13, the **initial** block, with label **block2**, uses an if-test with display command to test the success of the **randomize()** method and reports an error if the randomization fails. This is another example of the if-error-display-message coding style.

```
TestVars B3=new;

initial begin: block3
  repeat(10) assert(B3.randomize());
end
```

Example 14 - Assertion of randomize() method

In Example 14, the **`initial`** block, with label **`block3`**, uses an immediate **`assert`** command to test the success of the **`randomize()`** method and automatically reports an error if the randomization fails. Of course, users may add either a long label to the assertion or they can add an else clause with user defined code and messages to report assertion failures.

3.3 Immediate Assertion Summary

Immediate assertions do not offer broad assertion usage value, but they do offer a concise and convenient form to test the success of dynamic casting and constrained randomization of variables.

4 Concurrent Assertions

The most valuable assertion style that can be used in design and verification environments is the concurrent assertion. Concurrent assertions are little monitors that sit down inside of a block of code to periodically sample and test signals and generate error messages if the assertion ever fails.

Concurrent assertions are typically sampled once per clock period at the end of the clock cycle, just before the next active clock edge.

Concurrent assertions require the assertion of a property, where a property is basically a design rule that should always be true. The simplest of concurrent assertions takes the form:

```
assert property ( property_definition );
```

Example 15 - Simple property assertion

The property definition requires a sampling signal (typically a clock edge) that is either explicitly listed in the property definition or inherited from a **`default clocking`** block definition[1].

The property definition can also specify a condition under which the property is disabled, followed by either a Boolean expression that should always be true or a user-defined sequence of signals that should always be true as shown in Example 16.

[1] Some implementations currently require the explicit clocking signal and do not recognize the existence of a clocking signal defined by a **`default clocking`** block.

```
assert property (
  @(sample_signal)
  disable iff ( expression )  // optional disable condition
  property_expression_or_sequence
);
```

Example 16 - Simple property assertion with property definition details shown

Each property can be declared individually and then separately asserted as shown in Example 17.

```
property p1;
  @sample_signal
  disable iff ( expression )  // optional disable condition
  property_expression_or_sequence
endproperty

assert property ( p1 ) optional_action_block ;
```

Example 17 - Separate property definition with subsequent property assertion

The separate property declarations can be grouped into a library of properties to be asserted on multiple designs.

5 Concise Assertion Coding Styles

I have observed that most engineers like the idea of using concurrent assertions, and enthusiastically embrace them in training, but frequently design engineers use them sparingly or abandon them altogether. The reason for this abandonment seems to be the verbose nature of declaring individual properties and then being forced to assert the properties separately.

There are two useful techniques ("tricks") to create concise and yet powerful assertions. Those methods are (1) to define **`default clocking`** blocks and **`always`** blocks that will disable assertions during reset, and (2) use simple macro definitions

5.1 Default clocking blocks and $assertkill

Before launching into a description of this technique, it should be noted that not all SystemVerilog simulators have implemented the ability to use concurrent assertions with a **`default clocking block`**, and not all simulators have implemented the full set of **`$assert`** system tasks. The **`$assert`** system tasks may not be well supported by formal tools. The macro definitions of Section 5.3.4 will work with all simulators and formal tools.

The **`default clocking`** block can be used to create a sample signal that is used by a concurrent assertion. If a **`default clocking`** block is defined, then the assertion property can inherit the **`default clocking`** definition to identify the sampling signal for a property. An example **`default clocking`** block is shown in Example 18.

```
default clocking cb1 @(posedge clk);
endclocking
```

Example 18 - Default clocking block - posedge clk is the assertion sample signal

Many concurrent assertions disable themselves with the **`disable iff (!rst_n)`** qualifier when a valid reset signal is detected in an active concurrent assertion as shown in Example 2.

Three system tasks were added to SystemVerilog to help manipulate assertions. The three **`$assert`** system tasks are:
(1) **`$assertoff`** - used to disable all assertions but allows currently active assertions to complete before being disabled.
(2) **`$assertkill`** - used to kill and disable all assertions including currently active assertions.
(3) **`$asserton`** - used to turn all assertions back on.

When these system tasks are called with no arguments, they affect all assertions, but they can also be called with one or more arguments.

If called with arguments, the first argument indicates how many levels of hierarchy are affected by the selected **`$assert`** task. This number is consistent with the number of levels of hierarchy that are called with the Verilog **`$dumpvars`** system task.

All subsequent arguments indicate specific properties that are affected by the selected **`$assert`** task. There is currently no way to indicate *"all properties except ..."* as an argument.

If the **`clocking block`** from Example 18 and the **`always`** block shown in Example 19 are both active, then the assertion from Example 3 can be re-written as shown in Example 20.

```
always @(rst_n)
  if (!rst_n) $assertkill;
  else        $asserton;
```

Example 19 - Reset block with $assertkill and $asserton

```
ERROR_q_did_not_follow_d: assert property (q==$past(d));
```

Example 20 - Concise assertion with active clocking block and $assertkill on reset

Unfortunately, not all SystemVerilog simulators support the use of the **`clocking block`** and the **`$assert`** system tasks, but it is possible to accomplish the same goals using clever macros with arguments.

5.2 Macros with arguments

A little known capability of the Verilog language is the ability to create macro definitions with arguments. This capability was added to the IEEE Std 1364-1995. This capability can be used to simplify the construction of concise concurrent assertions.

5.2.1 Simple macro definitions

One important use of the **`define** compiler directive is to perform text substitution. The defined macro is placed wherever the defined text string is to be inserted. When the macro is used, it must be preceded by the back-tic (`) character. An example of simple macro usage is the definition of a clock **CYCLE** and accompanying definition of a free-running clock oscillator using the defined **`CYCLE** macro as shown in Example 21.

```
`define CYCLE 100
...
initial  begin
  clk <= '0;
  forever #(`CYCLE/2) clk = ~clk;
end
```

Example 21 - Simple macro definition and usage to define a clock oscillator

5.2.2 Complex macro definitions with arguments

The 1995 Verilog Standard[7] specified that macros could be defined on multiple lines by adding the backslash (\) continuation character just before the newline character. At the point where the macro substitution takes place, all of the macro code is inserted with newlines but without the continuation characters. Macros can also be defined with single-line comments and the comments are preserved in the text substitution.

The 1995 Verilog Standard also added the ability to pass arguments to macros and that the scope of the arguments extended up to the end of the macro definition.

Consider the concurrent assertion shown in Example 3 and repeated below:

```
ERROR_q_did_not_follow_d:
  assert property (@(posedge clk) disable iff (!rst_n) (q==$past(d)));
```

This assertion contains several pieces that are likely to be repeated across multiple assertion definitions. The pieces that are likely to be repeated include:

- **assert property** - keywords to start the definition of an assertion.
- **(...);** - placeholder for the assertion.
- **@(posedge clk)** - sample signal for the concurrent assertion.
- **disable iff (!rst_n)** - definition of when the assertion should become inactive.

The only part that is likely to be unique to the assertion in Example 3 is:

- **q==$past(d)** - actual assertion test.

We can replace the repetitive portions of the assertion with the incomplete macro definition, as shown in Example 22:

```
`define assert_clk(  ...  ) \
  assert property (@(posedge clk) disable iff (!rst_n)  ...  )
```

Example 22 - Incomplete assertion macro with commonly used assertion code

Then we can complete the macro by adding the ability to pass the actual assertion test code into the macro using the argument (**`arg`**) as shown in Example 23.

```
`define assert_clk( arg ) \
  assert property (@(posedge clk) disable iff (!rst_n) arg )
```

Example 23 - Completed assertion macro with argument passed to the macro

With this macro definition in place, it is now possible to re-code the assertion of Example 2 using the macro definition as shown Example 24.

```
ERROR_q_did_not_follow_d:
  `assert_clk(q==$past(d));
```

Example 24 - Macro with argument used to declare concurrent assertion

After creating a few key macro definitions, the job of writing assertions becomes much more concise and much easier to do.

5.2.3 SystemVerilog-2009 macros with default arguments

New to SystemVerilog-2009 will be the ability to add macro arguments with default values. This means that it will be possible to have multiple macro arguments, where one or more of the arguments has an assigned default value, and then to call the macro with or without listing all of the arguments that have default values.

The macro definition of Example 23 has been augmented in Example 25 to include two new arguments, **`enable_error`** and **`msg`**, both with default values. The macro code also includes an assertion **`else`** clause that displays a default error message if the **`enable_error`** argument is set (by default it is not set) and prints a default error message with formatted simulation timestamp (**`%t`** with **`$time`**), the full scope path to the assertion that triggered (**`%m`**) and a message string (**`%s`** with contents of **`msg`** argument set to a null string **`""`** by default).

```
`define assert_clk(arg, enable_error=0, msg="") \
    assert property (@(posedge clk) disable iff (!rst_n) arg) \
    else if(enable_error) $error("%t: %m: %s", $time, msg)
```

Example 25 - SystemVerilog-2009 macro definition - two of three arguments have default values

The modified assertion of Example 25 can be called just as it was in Example 24 with no change in behavior when compared to the macro defined in Example 23, or it can now be called with the **`enable_error`** argument set to **`1`** and an optional message to be displayed, as shown in Example 26.

```
ERROR_Q_DID_NOT_FOLLOW_D: `assert_clk((q==$past(d)),1,"***ERROR!!***");
```

Example 26 - SystemVerilog-2009 macro called with non-default arguments

The ability to add arguments with default values to macro definitions means that existing SystemVerilog-2005 macros could be updated and extended with additional functionality without breaking backward compatible behavior when your chosen SystemVerilog simulator supports this new SystemVerilog-2009 feature.

The ability to add default values to task and function arguments was added to SystemVerilog-2005. This is just an extension of that ability applied to macro definitions. Although this feature will prove to be a very useful capability in SystemVerilog-2009, it is not yet supported by all simulation vendors so this capability will not be used in the examples shown in the rest of this paper.

5.3 Measuring the efficiency of macro assertion coding styles

So how effective are the complex macro definitions with arguments discussed in section 5.2 compared to the concurrent assertion coding styles of section 4 that are typically used by many engineers? To measure the efficiency of different assertion coding styles, this section will examine a synchronous FIFO SVA example coded using: (1) separate properties and assertions, (2) combined properties and assertions, and (3) macros and assertions with arguments.

5.3.1 Synchronous FIFO assertion subset

Consider the example of a 16-deep, 1-clock synchronous FIFO design. Six sample assertions that could be applied to the design to test the FIFO with respect to correct operation when the FIFO is either asynchronously reset or full/near-full conditions include:

(1) When the FIFO is reset, the FIFO **`empty`** flag should be set and the **`full`** flag, **`wptr`** (write pointer), **`rptr`** (read pointer) and **`cnt`** (word counter) should all be cleared.
(2) If the word counter (**`cnt`**) is greater than 15, the FIFO is full.
(3) If the word counter (**`cnt`**) is less than 16, the FIFO is not full.
(4) If the word counter is 15 and there is a write operation without a simultaneous read operation, the FIFO should go full.
(5) If the FIFO is full, and there is a write operation without a simultaneous read operation, the **`full`** flag should not change.
(6) If the FIFO is full, and there is a write operation without a simultaneous read operation, the write pointer should not change.

This subset of synchronous FIFO assertions will be coded three different ways: (1) using individual property declarations and separately asserting each property (shown in Section 5.3.2), (2) asserting each property without a separate property declaration (shown in Section 5.3.3), and (3) using assertion macros (shown in Section 5.3.4).

5.3.2 Separate properties and assertions

One of the first techniques typically shown to engineers in assertion training, is the declaration of separately named properties, followed by the assertion of the named properties. Although the technique has merits especially for verification teams that intend to construct a large set of reusable properties that can then be used by others on the project team, the technique is far too verbose for the average design engineer who might intend to construct simple design-specific assertions to test simple corner cases in the design.

The FIFO assertion subset described in Section 5.3.1 is declared as a set of properties, each one separately asserted, as shown in Example 27 over the next couple of pages.

Asynchronous reset property:

```
property reset_rptr0_wptr0_empty1_full0_cnt0;
  @(posedge clk)
    (!rst_n |->
        (rptr==0 && wptr==0 && empty==1 && full==0 && cnt==0));
endproperty
```

FIFO full condition properties:

```
property full_fifo_condition;
  @(posedge clk) disable iff (!rst_n)
      (cnt>15 |-> full);
endproperty

property not_full_fifo_condition;
  @(posedge clk) disable iff (!rst_n)
      (cnt<16 |-> !full);
endproperty

property fifo_should_go_full;
  @(posedge clk) disable iff (!rst_n)
      (cnt==15 && write && !read |=> full);
endproperty

property full_write_full;
  @(posedge clk) disable iff (!rst_n)
      (full && write && !read |=> full);
endproperty

property full_write_wptr_no_change;
  @(posedge clk) disable iff (!rst_n)
      (full && write && !read |=> $stable(wptr));
endproperty
```

Now assert the predefined FIFO properties. Asynchronous reset assertion:

```
ERROR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0_RPTR0_WPTR0_CNT0:
  assert property (reset_rptr0_wptr0_empty1_full0_cnt0);
```

FIFO full condition assertions:

```
ERROR_FIFO_SHOULD_BE_FULL:
  assert property (full_fifo_condition);

ERROR_FIFO_SHOULD_NOT_BE_FULL:
  assert property (not_full_fifo_condition);

ERROR_FIFO_DID_NOT_GO_FULL:
  assert property (fifo_should_go_full);

ERROR_FIFO_FULL__WRITE_CAUSED_FULL_FLAG_TO_CHANGE:
  assert property (full_write_full);
```

```
ERROR_FIFO_FULL__WRITE_CAUSED_WPTR_TO_CHANGE:
  assert property (full_write_wptr_no_change);
```

Example 27 - FIFO assertion subset declared as separate properties and assertions

The declaration and assertion of these properties requires 37 lines of code (blank lines omitted) and 1,225 characters. That is a lot of code and effort to monitor six potential error conditions, which is why design engineers quickly abandon this assertion coding style.

5.3.3 Combined properties and assertions

Another technique frequently shown to engineers in assertion training, is the declaration of asserted properties without separate declaration of named properties. Although this technique does work, the technique is still too verbose for the average design engineer who might intend to construct design-specific assertions.

The FIFO assertion subset described in Section 5.3.1 is declared as a set of combined properties and assertions as shown in Example 28 over the next couple of pages.

Asynchronous reset assertion:

```
ERROR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0_RPTR0_WPTR0_CNT0:
  assert property (@(posedge clk)
    (!rst_n |->
        (rptr==0 && wptr==0 && empty==1 && full==0 && cnt==0)));
```

FIFO full condition assertions:

```
ERROR_FIFO_SHOULD_BE_FULL:
  assert property (@(posedge clk) disable iff (!rst_n)
      (cnt>15 |-> full));

ERROR_FIFO_SHOULD_NOT_BE_FULL:
  assert property (@(posedge clk) disable iff (!rst_n)
      (cnt<16 |-> !full));

ERROR_FIFO_DID_NOT_GO_FULL:
  assert property (@(posedge clk) disable iff (!rst_n)
      (cnt==15 && write && !read |=> full));

ERROR_FIFO_FULL__WRITE_CAUSED_FULL_FLAG_TO_CHANGE:
  assert property (@(posedge clk) disable iff (!rst_n)
      (full && write && !read |=> full));

ERROR_FIFO_FULL__WRITE_CAUSED_WPTR_TO_CHANGE:
  assert property (@(posedge clk) disable iff (!rst_n)
      (full && write && !read |=> $stable(wptr)));
```

Example 28 - FIFO assertion subset declared as combined properties and assertions

The declaration and assertion of these properties requires 19 lines of code (blank lines omitted) and 809 characters. Although not as verbose as the separate property declarations and assertions of Section 5.3.2, it is still a lot of code and effort to monitor six potential error conditions, which is why design engineers also quickly abandon this assertion coding style.

For most design engineers, these last two assertion coding styles are the only styles that engineers have been taught, so many design engineers abandon adding assertions altogether.

5.3.4 Macros and assertions

As stated earlier in the paper, design engineers frequently avoid writing assertions, because it takes too much code to create the assertions to test even the simplest design features.

The technique that I encourage most design engineers to use is to define a couple of simple, yet powerful, macros that can reduce the coding effort required to add assertions to the typical design.

The FIFO assertion subset described in Section 5.3.1 is declared using a couple of simple macros as shown in the assertion macro definitions of Example 29.

Assertion macro definitions:

```
`define assert_clk(arg) \
    assert property (@(posedge clk) disable iff (!rst_n) arg)

`define assert_async_rst(arg) \
    assert property (@(posedge clk) arg)
```

Asynchronous reset assertion:

```
ERROR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0_RPTR0_WPTR0_CNT0:
    `assert_async_rst(!rst_n |->
        (rptr==0 && wptr==0 && empty==1 && full==0 && cnt==0));
```

FIFO full condition assertions:

```
ERROR_FIFO_SHOULD_BE_FULL:
    `assert_clk (cnt>15 |-> full);

ERROR_FIFO_SHOULD_NOT_BE_FULL:
    `assert_clk (cnt<16 |-> !full);

ERROR_FIFO_DID_NOT_GO_FULL:
    `assert_clk (cnt==15 && write && !read |=> full);

ERROR_FIFO_FULL__WRITE_CAUSED_FULL_FLAG_TO_CHANGE:
    `assert_clk (full && write && !read |=> full);

ERROR_FIFO_FULL__WRITE_CAUSED_WPTR_TO_CHANGE:
    `assert_clk (full && write && !read |=> $stable(wptr));
```

Example 29 - FIFO assertion subset declared and asserted using concise macro definitions

The declaration of the macros and use of the assertion-macros requires 17 lines of code (blank lines omitted) and 725 characters. The declaration of the assertions omitting the macro declarations requires 13 lines of code (blank lines omitted) and 568 characters. This is a reasonable effort to monitor six potential FIFO error conditions. This technique permits rapid

definition of assertions that offer great value to the design engineer. I have found that design engineers are much more willing to adopt assertion based design techniques when presented with this simple, yet powerful, assertion macro technique.

A good way to approach the use of assertion macros is to think of the assertion label as the comment that describes the condition if the assertion fails, thereby documenting the intent of the assertion, followed by the actual assertion test. All of the tedious overhead-code of the assertion has been collected into the macro definition itself.

5.4 Assertion coding benchmarks

So what is the effort required to code a reasonable set of assertions using the three techniques described in the preceding sections?

The Appendix in Section 12 includes a set of 13 assertions that could reasonably be applied to a 1-clock synchronous FIFO design. The assertions are coded using the three techniques described in the preceding sections.

After coding the assertions using all three techniques, the code volume was measured as the number of lines of code required and characters used to code each set of assertions. The blank lines were omitted from the measurements. The assertion macro definitions were also omitted from the measurements under the assumption that as more macro-assertions are added to a design, the six lines of definition code would eventually become insignificant.

Style (blank lines omitted)	Lines of code	Additional lines of code (%)	Characters	Additional characters (%)
Property/Assert Property	79	193%	2599	126%
Assert Property	40	48%	1707	48%
Macros	27	0%	1151	0%

Table 1 - Assertion coding effort

It can be seen from Table 1 that the coding effort required to add one of the traditional assertion coding techniques with assertion-labels required approximately 50%-125% more characters than what was required to add the same assertions using the concise macro definitions.

The same assertions were then measured after deleting the labels. The assumption is that the labels represent a minimal set of comments that an engineer should already be adding to the design regarding each corner case tested with an assertion. After omitting the labels, we can accurately measure the effort required to just add assertion tests to this FIFO design.

Style (with no labels & blank lines omitted	Lines of code	Additional lines of code (%)	Characters	Additional characters (%)
Property/Assert Property	66	371%	2067	249%
Assert Property	27	93%	1175	98%
Macros	14	0%	593	0%

Table 2 - Assertion coding effort with labels omitted

It can be seen from Table 2 that the coding effort required to add one of the traditional assertion coding techniques required approximately 100%-250% more characters than what was required to add the same assertions using the concise macro definitions.

Any of these SVA coding styles work well, but I have found that engineers are much more willing to add assertions to their designs if the effort required to add the assertions is reasonable. The concise macro definitions offer a much more attractive option over traditional SVA coding styles.

6 SVA Bind Files

There are times when there is a golden Verilog or VHDL model that cannot be touched. Under these circumstances, business decisions dictate that the model cannot be modified, yet it would be useful to add SVA to the model. How can SVA be added to such models? The answer is to bind an SVA file to the golden model as described in this section.

What if you could secretly (or not-so secretly) instantiate a module with SVA into the golden Verilog or VHDL RTL file without disturbing the exiting Verilog or VHDL code. This is the idea behind an SVA bind file. Binding an SVA file to another design is like poking or projecting an instantiation of an SVA module into the unmodified target file. Binding an SVA file to a target file is an out-of-body experience for the target file!

The official description of bind files and usage can be found in section 17.15 of the IEEE Std 1800-2005[9]. The examples in the IEEE Standard are somewhat abbreviated and can be difficult to understand, so additional and more complete examples are shown in this section of the paper.

Contrasting SystemVerilog bind files to the PSL **`vunit`**, the **`vunit`** is almost like an external **`` `include ``** statement, where the **`vunit`** code is included into the target module without placing the **`` `include ``** into the golden source code. A PSL **`vunit`** does not surround the set of included assertions with any type of scope container, such as a **`module`**, and does not require any port connections to connect the signals of the **`vunit`** to the signals of the target module. The **`vunit`** scope container is the target module that it is attached to itself. The signals in the **`vunit`** are coded to match the names of the signals in the target module. If a second copy of the **`vunit`** assertions is required to connect to a second set of signals in the target module, the **`vunit`** must be copied and signal names changed.

Unlike the PSL **`vunit`**, SVA bind files require that the assertions be wrapped in a module that includes port declarations as shown in Example 30.

```
`define assert_clk(arg) \  ...
`define assert_asyn_rst(arg) \  ...
module pLib_fifo (
  input [7:0] dout, din,
  input [4:0] cnt,
  input [3:0] wptr, rptr,
  input       empty, read, full,
  input       write, clk, rst_n);

  ERROR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0_RPTR0_WPTR0_CNT0:
    `assert_async_rst(!rst_n |->   ...

  ERROR_FIFO_SHOULD_BE_FULL:
    `assert_clk (cnt>15 |-> full);
  ...
endmodule
```

Example 30 - SystemVerilog assertions wrapped in a module for use as a bind file

The SVA bind file is externally instantiated into the target design module without making any modification to the target module itself. The SVA **bind** command is used to externally instantiate, or to "bind" the assertion module into the target module.

Since the SVA assertions are wrapped in an enclosing module with ports, it creates its own scope and the signal names in the assertions do not have to match the signal names of the target module. The mapping of target signal names to assertion-file signal names happens when the assertion module ports are connected through the bind-instantiation to the target module ports using standard Verilog named port connections (preferred method) or Verilog positional port connections.

If the assertion module uses the same signal names as the target module, the bind file port declarations are still required but the bind-instantiation can be done using the SystemVerilog **.*** implicit port connections[3]. A sample of this type of bind-instantiation is shown in Example 31.

```
module tb1;
  logic [7:0] dout;
  logic       full, empty;
  logic       write, read, clk, rst_n;
  logic [7:0] din;
  ...
  fifo1 u1   (.*);
  bind fifo1: u1 pLib_fifo p1 (.*);
  ...
endmodule

module fifo1 (
  output logic [7:0] dout,
  output logic       full, empty,
  input  logic       write, read, clk, rst_n,
  input  logic [7:0] din);
  logic [7:0] fifomem [0:15];
  logic [3:0] wptr, rptr;
  logic [4:0] cnt;
  ...
endmodule
```

Example 31 - pLib_fifo assertion file bound to the u1 instance of the fifo1 module with matching signal names

Note that the **fifo1** module has internal vectors named, **wptr**, **rptr** and **cnt**, and the **pLib_fifo** module has ports by the same names, but these same vectors are not declared in the **tb1** module, because these vectors do not exist in the **tb1** module. These vectors only exist in the **fifo1** and **pLib_fifo** modules and since the **pLib_fifo** module is indirectly instantiated into the **fifo1** module through the use of the "bind" mechanism, and does not really exist in the **tb1** module, there is no need to make the **wptr**, **rptr** and **cnt** declarations in the **tb1** module.

If the **tb1** module used named port connections instead of the **.*** implicit port connections as shown in Example 31, the **fifo1** instantiation and **pLib_fifo** **bind** commands would be expanded as shown in Example 32.

```
module tb1a;
  logic [7:0] dout;
  logic       full, empty;
  logic       write, read, clk, rst_n;
  logic [7:0] din;
  ...
  fifo1 u1   (.dout(dout),   .full(full), .din(din), .empty(empty),
              .write(write), .read(read), .clk(clk), .rst_n(rst_n));
  bind fifo1: u1 pLib_fifo p1 (
              .dout(dout),   .full(full), .din(din), .empty(empty),
              .write(write), .read(read), .clk(clk), .rst_n(rst_n),
              .wptr(wptr),   .rptr(rptr), .cnt(cnt));
  ...
endmodule
```

Example 32 - tb1a with fifo1 instantiation and pLib_fifo bind commands using named port connections

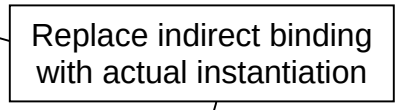
Again note that the bound **pLib_fifo** module references the **wptr**, **rptr** and **cnt** vectors that exist in the **fifo1** module but do not exist in the **tb1a** testbench module.

This is an important concept to understand with regards to binding files: that file binding is used to place a copy of the bound module into a different location and not in the file where the **bind** keyword is used. If it were permitted to instantiate the **pLib_fifo** SVA module directly into the **fifo1** module, we could delete the **bind** command from the **tb1** module and directly instantiate the **pLib_fifo** module into the **fifo1** module, as shown in Example 33.

```
module tb1;
  ...
  fifo1 u1   (.*);
  bind fifo1: u1 pLib_fifo p1 (.*);
  ...
endmodule

module fifo1 (
  output logic [7:0] dout,
  output logic       full, empty,
  input  logic       write, read, clk, rst_n,
  input  logic [7:0] din);
  logic [7:0] fifomem [0:15];
  logic [3:0] wptr, rptr;
  logic [4:0] cnt;
  ...

  pLib_fifo p1 (.*);
  ...
endmodule
```



Example 33 - The bound pLib_fifo instantiation replaced with an equivalent instantiation

6.1 A closer look at the bind command

Let's examine the **bind** command from Example 31 in greater detail. A second form of the bind command is discussed later in this section.

```
bind   fifo1: u1
```

```
pLib_fifo p1   (.*);
```

In the first box:

The command uses the **bind** keyword and is followed by the target module name (**fifo1**) that we are binding to. This example also shows the optional argument that allows us to only bind to the **u1** instance[2] of the **fifo1** module (**: u1**) and not to every instance of the **fifo1** module, which would happen if we had omitted the **: u1** argument.

In the second box:

Now we need to indicate which file is to be bound into the target module. The name of the file to be bound is the assertion file (**pLib_fifo**) and when the assertion file is bound into the target module, it shall have the instance name **p1**, and the instantiation shows that all of the ports on the bound assertion file are connected to signals with the same name (**(.*);**) in the target module.

Note that if the assertion file had been instantiated directly into the target module, the instantiation text would be exactly all of the text in the second box.

There is a second legal form of the bind command that allows binding to individual instances in a design without naming the instantiated module. This is done by dropping the ***module_name:*** from the first box shown above. To rewrite the bind command used in Example 31, simply reference the instance name with the bind command in the first box as shown below. This second form of the bind command is currently better supported by most tools to bind a file to just one instance of a module.

```
bind u1
```

```
pLib_fifo p1   (.*);
```

A frequently asked question about the **bind** command as shown above is, why is it necessary to include the instance name **p1**? Part of the answer to this question is that all instantiated modules must have an instance name attached to the instantiation. The instance name allows another module to hierarchically reference signals in the **p1**-scope.

A second reason to have the instance name **p1** is, suppose the target module **fifo1** actually is a dual fifo module and we would like to attach two copies of the exact same **pLib_fifo** assertions to each fifo block in the module.

If the assertion module uses different signal names than the target module, the bind file port declarations are still required and the bind-instantiation is done using named (or positional) port connections for all ports not connected by using **.*** implicit ports, as shown in Example 34.

[2] At the time that this paper was published, not all SystemVerilog implementations permitted binding to the optional single instance of the target module, but Questa does have this feature implemented

```
module tb2;
  ...
  fifo2 u1   (.*);
  bind fifo2: u1
    pLib_fifo1 p1 (.wptr(qptr), .rptr(iptr), .cnt(word_counter), .*);
  ...
endmodule

module fifo2 (
  output logic [7:0] dout,
  output logic       full, empty,
  input  logic       write, read, clk, rst_n,
  input  logic [7:0] din);
  logic [7:0] fifomem [0:15];
  logic [3:0] qptr, iptr;      // new names for wptr and rptr
  logic [4:0] word_counter;    // new name  for cnt
  ...
endmodule
```

Example 34 - Binding to a file where the bind-file port names do not match the target module signal names

In Example 34, the assertion bind-file has ports named **`wptr`**, **`rptr`** and **`cnt`**, that need to be connected to the **`fifo2`** **`qptr`**, **`iptr`** and **`word_count`** buses respectively. Since the names do not match, the **`bind`** command connects these **`pLib_fifo1`** ports to the corresponding **`fifo2`** ports by name and makes all other connections using **`.*`** implicit port connections.

If we were permitted to instantiate the pLib_fifo1 assertion file directly into the fifo2 module, we would need to use the exact same named port connections for non-matching signals and .* for all remaining signals, as shown in Example 35.

```
module tb2;
  ...
  fifo2 u1   (.*);
  bind fifo2: u1
    pLib_fifo1 p1 (.wptr(qptr), .rptr(iptr), .cnt(word_counter), .*);
  ...
endmodule

module fifo2 (
  output logic [7:0] dout,
  output logic       full, empty,
  input  logic       write, read, clk, rst_n,
  input  logic [7:0] din);
  logic [7:0] fifomem [0:15];
  logic [3:0] qptr, iptr,      // new names for wptr and rptr
  logic [4:0] word_counter;    // new name  for cnt
  ...

  pLib_fifo1 p1 (.wptr(qptr), .rptr(iptr), .cnt(word_counter), .*);
  ...
endmodule
```

Example 35 - The bound pLib_fifo instantiation replaced with an equivalent instantiation in the fifo2 module

Binding assertions to a golden Verilog RTL model allows design and verification engineers to still take advantage of assertion based design techniques without the requirement to modify a golden Verilog model. Although not defined in the SystemVerilog IEEE Std 1800-2005, many vendors also allow engineers working in a mixed Verilog & VHDL design and verification environment to bind SVA files to a golden VHDL model.

6.2 SystemVerilog bind file use and abuse

The bind command was originally intended to be used to add assertions to a design without modifying the original RTL code. The use has been further expanded to add simple verification capabilities to a design in a non-obtrusive way. When used in this context, the bind command is both safe and very useful.

One interesting use of the bind command was shared by my colleague, John Dickol[11]. John had used some SRAM memory behavioral models from a vendor and did not want to modify the source code, but needed a way to pre-load the contents of the memory at the start of simulation. John created an sram_loader module that he would then bind into each instance of the SRAM module. This gave him a handy way to access (set & get) the contents of the memory. Although the same sram_loader functionality could have been achieved by instantiating the loaders into the testbench and connecting the ports to the SRAM via hierarchical references, the bind technique is more convenient, especially if the sram_loader had been bound into *every* instance of the sram with a single bind command. Regarding recommended bind-command usage, modifying memory contents via a bound module is probably no worse than modifying memory contents using hierarchical references. Both are powerful and both can be abused.

When we bind an assertion file to a design, the assertion file generally does not drive any signals back into the design. In general, assertion files should only have inputs to monitor the signals in a design and to report problems when they are detected. It is relatively safe to bind any code into a module as long as the bound code does not have output drivers that could modify the behavior of the module that is touched by the bind statement.

The bind statement was never intended to be used to configure a design, or to setup a mechanism to instantiate modules externally; in fact such usage can be quite dangerous and is generally discouraged.

In my discussions with knowledgeable colleagues[2][5], a number of potential abuses and problems have been identified. Among potential problems that we have identified are the following:

6.2.1 Binding invisibility and multiple bound modules

The bound module is invisible to the casual reader of the target module (it is not in the code). If multiple engineers decide to bind modules into the same target module, there could be unexpected interactions between the design and the bound modules.

One must be sure that the multiple bound modules do not have the same instance name and again the practice of binding modules will be safest if the bound modules have input ports and no output parts.

6.2.2 Nested binding is not permitted

In IEEE Std 1800-2005[9], at the very end of section 17.15 is an important nested-bind restriction:

> It shall be an error for a **`bind`** statement to bind a *bind_instantiation* underneath the scope of another *bind_instantiation*.

What this means is that if a design module is bound into another design module, and then if an engineer decides to bind assertions into the bound module, this technically is not legal, so in theory, once you bind a module into another module, you are then prohibited from binding any assertions into the bound module.

Currently there are tools that permit the violation of this restriction, but in my discussions with SystemVerilog tool implementers, these implementers indicate that their tools might not support this nesting in future versions of the tools or in the unusual ways that users might try to use this restricted feature. Also, due to this restriction in the SystemVerilog Standard, one cannot guarantee that all tools will implement this feature the same way or even implement the feature at all.

6.2.3 complex design structure created through bind commands

One unexpected usage seen by an EDA vendor was an attempt to bind-instantiate an interface that was then connected to hierarchically referenced interface ports. Something like this:

```
interface intf();
  ...
endinterface

module m1 ();
  ...
endmodule

module m2 (intf i);
  ...
endmodule

module top();
  m1 u1();
  bind m1 intf i1 (); // interface i1 is instantiated inside of instance u1

  m2 u2 (u1.i1);      // module instance u2 attempts to connect to the i1
                      // interface instantiated inside of the u1 module.
                      // bind creates structural connection between u1 & u2
endmodule
```

Example 36 - Non-recommended complex design structure created using a bind command

The current SystemVerilog Standard may not make this example illegal, but it creates potential problems for EDA vendors. Interface port connections must be resolved before parameters are evaluated, because types and parameter values can depend on the interface connected to the port.

The bind command dependencies were not intended to create a structurally valid design.

7 SVA File Methodologies

Some of the best assertion based design methodologies, formulated over years of actual project experience, using various assertion languages can be found in Foster, Krolnik & Lacey[6].

There are a few overlapping and additional methodologies that are worth mentioning in this paper.

Using assertions with an RTL file is simple to do. Assertions can be employed by:
(1) adding the assertions directly into the RTL source code.
(2) placing the assertions into a separate file and including the file into the RTL source code using the **`include** compiler directive.
(3) placing the assertions into a separate module (also in a separate file) and instantiating the module into the RTL source code.
(4) placing the assertions into a separate module (separate file) and binding the module to the design from a third module, such as a testbench as previously shown in Section 6.

One of the frequently asked questions posed to me by engineers who use or intend to use design assertions is, how can we preserve the same assertions in the gate-level design after synthesis?

In theory, a synthesis tool could preserve many, if not all, of the RTL assertions and add them to the gate-level netlist. In the absence of this capability, engineers might consider placing most of the assertions into the separate files for inclusion (using `include), instantiating the assertions or binding the assertions.

7.1 Partitioning assertion files

What happens if the gate-level netlist is missing signals that are part of one or more assertions? Planning ahead and strategic partitioning can help to reduce the problem.

It may be wise to create two or more assertion files where the first assertion file only references ports on the RTL design. The RTL module ports are not likely to be removed in the synthesis process.

The second assertion file could reference internal RTL design signals (and ports). Even this file might be wisely partitioned into one assertion file that only references ports and internal signals that are registered outputs and another assertion file that references ports and all internal signals, including combinational signals that could be optimized away in the synthesis process.

The code in Example 37, Example 38 and Example 39 include the same set of assertions that were shown in Example 29, except that they have been repartitioned into three files. Note that there are now two assertions to test asynchronous reset conditions that replace the one asynchronous reset testing assertion found in Example 29:
ERROR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0 (ports only) and
ERROR_FIFO_RESET_SHOULD_CAUSE_RPTR0_WPTR0_CNT0 (ports and internal registered signals).

```
ERROR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0:
    `assert_async_rst(!rst_n |-> (empty==1 && full==0));

ERROR_FIFO_SHOULD_BE_FULL:
    `assert_clk (cnt>15 |-> full);

ERROR_FIFO_SHOULD_NOT_BE_FULL:
    `assert_clk (cnt<16 |-> !full);

ERROR_FIFO_DID_NOT_GO_FULL:
    `assert_clk (cnt==15 && write && !read |=> full);

ERROR_FIFO_FULL__WRITE_CAUSED_FULL_FLAG_TO_CHANGE:
    `assert_clk (full && write && !read |=> full);
```

Example 37 - pLib_fifo_ports.sv - Assertion partitioning - ports-only assertions

The code in Example 38 includes the second asynchronous reset-test assertion and an assertion that tests the internal **`wptr`** register.

```
ERROR_FIFO_RESET_SHOULD_CAUSE_RPTR0_WPTR0_CNT0:
    `assert_async_rst(!rst_n |-> (rptr==0 && wptr==0 && cnt==0));

ERROR_FIFO_FULL__WRITE_CAUSED_WPTR_TO_CHANGE:
    `assert_clk (full && write && !read |=> $stable(wptr));
```

Example 38 - pLib_fifo_regs.sv - Assertion partitioning - ports and internal registered signals assertions

There is no code in Example 39 because there were no internal combinational logic signals tested by any of the assertions in Example 29.

```
// None of the FIFO assertions use internals combinational signals
```

Example 39 - pLib_fifo_sigs.sv - Assertion partitioning - ports and all internal signals assertions

With these three assertion files: ports-only, ports with internal register signals, ports with all internal signals, one could easily include, instantiate or bind the appropriate assertion files that still reference valid signals after synthesis. The user needs to either be ready to abandon an assertion file that no longer references valid signals, or be prepared to create a modified version of one of the assertion files that will reference the internal signals by their new name after synthesis. The user needs to determine if the synthesis process will continually modify the names of certain internal signals, which might make it too tedious to keep one of the assertion files up to date after each pass through the synthesis tool.

7.2 Synthesis tool enhancement request

Engineers should ask their favorite synthesis tool vendor(s) to preserve and insert most or all assertions from an RTL design into the gate-level design. This might take on multiple different forms.

One challenge that would be faced by synthesis tool vendors is the issue of signals that are called out in assertions that could be optimized away in the synthesis process. One approach to this

problem would be to notify the user that specific assertions have been removed. If the assertion had a label, listing the removed assertions by label would be the easiest way to identify the assertions that have been removed. Listing the line number of the assertion in the RTL source code would also provide useful information.

Another approach to this problem would be to allow the user to identify assertions that should not be removed, with the understanding such indications are roughly equivalent to telling the synthesis tool to "keep" certain signals during synthesis, which could hinder the best optimization capabilities of the synthesis tool. It would be useful to have a global switch that could be called to disable all of the "keep" commands to allow the user to compare the quality of the synthesis results with and without assertion "keep" commands.

If any of these ideas seem like a good idea to the reader, the reader is encouraged to tell the synthesis tool applications engineers that you would like the desired features to be added to the users synthesis tool.

8 Summary & Conclusions

Design engineers frequently do not use SVA because they perceive that the effort to code SystemVerilog properties and assertions is too verbose for the potential debugging benefit derived from adding the assertions.

The concise **`assert_clk** and **`assert_async_reset** macros significantly reduce the effort required to add assertions to an RTL design. In the examples cited, the `assert_macros were shown to reduce assertion coding efforts by 50%-80% over conventional SVA coding techniques.

The easier it is to add assertions to a design, the greater the likelihood that design engineers will embrace the usage of design assertions. Concise macro usage translates into greater acceptance of assertion deployment by design engineers.

If design engineers will use the SVA macros shown in Section 5.3.4, their designs will be significantly better tested than a design where assertions have been omitted and the design engineer will spend less "quality time" with the verification engineers.

SVA bind files allow engineers to indirectly insert SVA into a golden RTL model without modifying the RTL source file. The SVA binding can be done with both Verilog and VHDL RTL files, adding value to the design and verification environments of both languages.

9 Acknowledgements

My thanks to colleague Kelly Larson of MediaTek for his review of the PSL **VUNIT** capabilities as compared to SystemVerilog bind files. I would also like to thank John Dickol for sharing an interesting application of the SystemVerilog bind command referenced in section 6.2.

I am also very grateful to my colleague and formal verification expert, Anders Nordstrom, for offering all of the valuable recommendations in this paper regarding the use of assertions for formal verification.

A special thanks to my colleague Heath Chambers of HMC Design Verification for offering valuable suggestions to improve the quality and content of this paper.

10 References

[1] Anders Nordstrom, personal communication.

[2] Arturo Salz, personal communication

[3] Clifford E. Cummings, “SystemVerilog Implicit Port Enhancements Accelerate System Design & Verification,” SNUG (Synopsys Users Group) September 2007 (Boston, MA), September 2007. Also available at www.sunburst-design.com/papers

[4] Chris Spear, "SystemVerilog for Verification, 2nd Edition", Springer, www.springeronline.com, 2008

[5] Gordon Vreugdenhil, personal communication

[6] Harry Foster, Adam Krolnik, David Lacey, "Assertion Based Design, 2nd Edition", Springer, www.springeronline.com, 2004

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

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

[9] "IEEE Standard For SystemVerilog - Unified Hardware Design, Specification and Verification Language," IEEE Computer Society, IEEE, New York, NY, IEEE Std 1800-2005

[10] "IEEE P1800/D8 - Draft Standard For SystemVerilog - Unified Hardware Design, Specification and Verification Language," IEEE Computer Society, IEEE, New York, NY, IEEE Std P1800-2009/D8

[11] John Dickol, personal communication

11 Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 27 years of ASIC, FPGA and system design experience and 17 years of SystemVerilog, synthesis and methodology training experience.

Mr. Cummings has presented more than 100 SystemVerilog seminars and training classes in the past six years and was the featured speaker at the world-wide SystemVerilog NOW! seminars.

Mr. Cummings has participated on every IEEE & Accellera SystemVerilog, SystemVerilog Synthesis, SystemVerilog committee, and has presented more than 40 papers on SystemVerilog & SystemVerilog related design, synthesis and verification techniques.

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

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

An updated version of this paper can be downloaded from the web site: www.sunburst-design.com/papers
(Last updated March 24, 2009)

12 Appendix

This appendix includes a reasonably full set of assertions that could be used in a 16-deep, 1-clock synchronous FIFO design. The assertions are coded three different ways: (1) using separate property declarations and separately asserting each property, (2) asserting each property without a separate property declaration, and (3) Using assertion macros.

This paper encourages design engineers to use the concise assertion-macro coding style.

12.1 Synchronous FIFO assertions

This is a set of assertions that can be used with a 16-deep, 1-clock synchronous FIFO design. This set of assertions assumes that there can be simultaneous read and write operations on the same clock; however, if the FIFO is full during a simultaneous read/write, the write operation will not execute but the read operation will execute, causing the FIFO to no longer be full. Similarly, if the FIFO is empty during a simultaneous read/write, the read operation will not execute but the write operation will execute, causing the FIFO to no longer be empty.

Although the author believes this set of assertions has been properly coded, this set of assertions is not guaranteed to be complete or even correct. Users should verify the assertion correctness before using them on an actual project.

Asynchronous reset assertion:

(1) When the FIFO is reset, the FIFO **`empty`** flag should be set and the **`full`** flag, **`wptr`**, **`rptr`** and **`cnt`** (word count) should all be cleared.

FIFO full condition assertions:

(2) When the FIFO is full, the **`full`** flag should be asserted.
(3) When the FIFO is NOT full, the **`full`** flag should NOT be asserted.
(4) When FIFO **`cnt`**=15 and a **`write`** occurs (no **`read`**), the FIFO should go **`full`**.
(5) If FIFO is **`full`** and a **`write`** occurs (no **`read`**), the FIFO should still be **`full`**.
(6) If FIFO is **`full`** and a **`write`** occurs (no **`read`**), the FIFO **`wptr`** should not change.

FIFO empty condition assertions:

(7) When the FIFO is empty, the **`empty`** flag should be asserted.
(8) When the FIFO is NOT empty, the **`empty`** flag should NOT be asserted.
(9) When FIFO **`cnt`**=1 and a **`read`** occurs (no **`write`**), the FIFO should go **`empty`**.
(10) If FIFO is **`empty`** and a **`read`** occurs (no **`write`**), the FIFO should still be **`empty`**.
(11) If FIFO is **`empty`** and a **`read`** occurs (no **`write`**), the FIFO **`rptr`** should not change.

Additional interesting FIFO assertions:

(12) The FIFO **`cnt`** (word count) should never be negative.
(13) If **`read`** & **`write`** occur at the same time, the FIFO should not be **`full`** or **`empty`**.

12.2 Separate property and assertion style

Asynchronous reset property:

```
property reset_rptr0_wptr0_empty1_full0_cnt0;
  @(posedge clk)
    (!rst_n |->
        (rptr==0 && wptr==0 && empty==1 && full==0 && cnt==0));
endproperty
```

FIFO full condition properties:

```
property full_fifo_condition;
  @(posedge clk) disable iff (!rst_n)
      (cnt>15 |-> full);
endproperty

property not_full_fifo_condition;
  @(posedge clk) disable iff (!rst_n)
      (cnt<16 |-> !full);
endproperty

property fifo_should_go_full;
  @(posedge clk) disable iff (!rst_n)
      (cnt==15 && write && !read |=> full);
endproperty

property full_write_full;
  @(posedge clk) disable iff (!rst_n)
      (full && write && !read |=> full);
endproperty

property full_write_wptr_no_change;
  @(posedge clk) disable iff (!rst_n)
      (full && write && !read |=> $stable(wptr));
endproperty
```

FIFO empty condition properties:

```
property empty_fifo_condition;
  @(posedge clk) disable iff (!rst_n)
      (cnt==0 |-> empty);
endproperty

property not_empty_fifo_condition;
  @(posedge clk) disable iff (!rst_n)
      (cnt>0 |-> !empty);
endproperty

property fifo_should_go_empty;
  @(posedge clk) disable iff (!rst_n)
      (cnt==1 && read && !write |=> empty);
endproperty

property empty_read_empty;
  @(posedge clk) disable iff (!rst_n)
      (empty && read && !write |=> empty);
endproperty
```

```
property empty_read_rptr_no_change;
  @(posedge clk) disable iff (!rst_n)
      (empty && read && !write |=> $stable(rptr));
endproperty
```

Additional interesting FIFO properties:

```
property illegal_fifo_cnt_condition;
  @(posedge clk) disable iff (!rst_n)
      (not (cnt<0));
endproperty

property fifo_not_empty_not_full;
  @(posedge clk) disable iff (!rst_n)
      (write && read |=> !full && !empty);
endproperty
```

Assert the predefined FIFO properties:

Asynchronous reset assertion:

```
ERROR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0_RPTR0_WPTR0_CNT0:
  assert property (reset_rptr0_wptr0_empty1_full0_cnt0);
```

FIFO full condition assertions:

```
ERROR_FIFO_SHOULD_BE_FULL:
  assert property (full_fifo_condition);

ERROR_FIFO_SHOULD_NOT_BE_FULL:
  assert property (not_full_fifo_condition);

ERROR_FIFO_DID_NOT_GO_FULL:
  assert property (fifo_should_go_full);

ERROR_FIFO_FULL__WRITE_CAUSED_FULL_FLAG_TO_CHANGE:
  assert property (full_write_full);

ERROR_FIFO_FULL__WRITE_CAUSED_WPTR_TO_CHANGE:
  assert property (full_write_wptr_no_change);
```

FIFO empty condition assertions:

```
ERROR_FIFO_SHOULD_BE_EMPTY:
  assert property (empty_fifo_condition);

ERROR_FIFO_SHOULD_NOT_BE_EMPTY:
  assert property (not_empty_fifo_condition);

ERROR_FIFO_DID_NOT_GO_EMPTY:
  assert property (fifo_should_go_empty);

ERROR_FIFO_EMPTY__READ_CAUSED_EMPTY_FLAG_TO_CHANGE:
  assert property (empty_read_empty);

ERROR_FIFO_EMPTY__READ_CAUSED_RPTR_TO_CHANGE:
  assert property (empty_read_rptr_no_change);
```

Additional interesting FIFO assertions:

```
  ERROR_FIFO_WORD_COUNTER_IS_NEGATIVE:
    assert property (illegal_fifo_cnt_condition);

  ERROR_FIFO_READWRITE_ILLEGAL_FIFO_FULL_OR_EMPTY:
    assert property (fifo_not_empty_not_full);
```

12.3 Combined assert property style

Asynchronous reset assertion:

```
ERROR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0_RPTR0_WPTR0_CNT0:
  assert property (@(posedge clk)
    (!rst_n |->
        (rptr==0 && wptr==0 && empty==1 && full==0 && cnt==0)));
```

FIFO full condition assertions:

```
ERROR_FIFO_SHOULD_BE_FULL:
  assert property (@(posedge clk) disable iff (!rst_n)
      (cnt>15 |-> full));

ERROR_FIFO_SHOULD_NOT_BE_FULL:
  assert property (@(posedge clk) disable iff (!rst_n)
      (cnt<16 |-> !full));

ERROR_FIFO_DID_NOT_GO_FULL:
  assert property (@(posedge clk) disable iff (!rst_n)
      (cnt==15 && write && !read |=> full));

ERROR_FIFO_FULL__WRITE_CAUSED_FULL_FLAG_TO_CHANGE:
  assert property (@(posedge clk) disable iff (!rst_n)
      (full && write && !read |=> full));

ERROR_FIFO_FULL__WRITE_CAUSED_WPTR_TO_CHANGE:
  assert property (@(posedge clk) disable iff (!rst_n)
      (full && write && !read |=> $stable(wptr)));
```

FIFO empty condition assertions:

```
ERROR_FIFO_SHOULD_BE_EMPTY:
  assert property (@(posedge clk) disable iff (!rst_n)
      (cnt==0 |-> empty));

ERROR_FIFO_SHOULD_NOT_BE_EMPTY:
  assert property (@(posedge clk) disable iff (!rst_n)
      (cnt>0 |-> !empty));

ERROR_FIFO_DID_NOT_GO_EMPTY:
  assert property (@(posedge clk) disable iff (!rst_n)
      (cnt==1 && read && !write |=> empty));

ERROR_FIFO_EMPTY__READ_CAUSED_EMPTY_FLAG_TO_CHANGE:
  assert property (@(posedge clk) disable iff (!rst_n)
      (empty && read && !write |=> empty));

ERROR_FIFO_EMPTY__READ_CAUSED_RPTR_TO_CHANGE:
  assert property (@(posedge clk) disable iff (!rst_n)
      (empty && read && !write |=> $stable(rptr)));
```

Additional interesting FIFO assertions:

```
ERROR_FIFO_WORD_COUNTER_IS_NEGATIVE:
  assert property (@(posedge clk) disable iff (!rst_n)
      (not (cnt<0)));
```

```
ERROR_FIFO_READWRITE_ILLEGAL_FIFO_FULL_OR_EMPTY:
  assert property (@(posedge clk) disable iff (!rst_n)
      (write && read |=> !full && !empty));
```

12.4 Assertion macro style

Assertion macro definitions:

```
`define assert_clk(arg) \
    assert property (@(posedge clk) disable iff (!rst_n) arg)

`define assert_async_rst(arg) \
    assert property (@(posedge clk) arg)
```

Asynchronous reset assertion:

```
ERROR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0_RPTR0_WPTR0_CNT0:
    `assert_async_rst(!rst_n |->
       (rptr==0 && wptr==0 && empty==1 && full==0 && cnt==0));
```

FIFO full condition assertions:

```
ERROR_FIFO_SHOULD_BE_FULL:
    `assert_clk (cnt>15 |-> full);

ERROR_FIFO_SHOULD_NOT_BE_FULL:
    `assert_clk (cnt<16 |-> !full);

ERROR_FIFO_DID_NOT_GO_FULL:
    `assert_clk (cnt==15 && write && !read |=> full);

ERROR_FIFO_FULL__WRITE_CAUSED_FULL_FLAG_TO_CHANGE:
    `assert_clk (full && write && !read |=> full);

ERROR_FIFO_FULL__WRITE_CAUSED_WPTR_TO_CHANGE:
    `assert_clk (full && write && !read |=> $stable(wptr));
```

FIFO empty condition assertions:

```
ERROR_FIFO_SHOULD_BE_EMPTY:
    `assert_clk (cnt==0 |-> empty);

ERROR_FIFO_SHOULD_NOT_BE_EMPTY:
    `assert_clk (cnt>0 |-> !empty);

ERROR_FIFO_DID_NOT_GO_EMPTY:
    `assert_clk (cnt==1 && read && !write |=> empty);

ERROR_FIFO_EMPTY__READ_CAUSED_EMPTY_FLAG_TO_CHANGE:
    `assert_clk (empty && read && !write |=> empty);

ERROR_FIFO_EMPTY__READ_CAUSED_RPTR_TO_CHANGE:
    `assert_clk (empty && read && !write |=> $stable(rptr));
```

Additional interesting FIFO assertions:

```
ERROR_FIFO_WORD_COUNTER_IS_NEGATIVE:
    `assert_clk (not (cnt<0));

ERROR_FIFO_READWRITE_ILLEGAL_FIFO_FULL_OR_EMPTY:
    `assert_clk (write && read |=> !full && !empty);
```