SNUG-2016
Silicon Valley, CA
Voted Best Paper
1st Place

Sunburst Design

World Class SystemVerilog & UVM Training

SystemVerilog Assertions - Bindfiles & Best Known Practices for Simple SVA Usage

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

ABSTRACT

SystemVerilog Assertions (SVA) can be added directly to the RTL code or be added indirectly through bindfiles. Best known practices suggest that it is better to add most assertions using bindfiles. This paper will explain why adding assertions directly to the RTL code can be problematic and why bindfiles solve many of the problems. This paper also explains how to use bindfiles efficiently and why engineers should generally use concurrent assertions while avoiding immediate assertions. This paper will also give assertion coding guidelines and styles that help reduce assertion coding effort, assertion coding mistakes and encourage designers to be more proactive in adding assertions to their designs.

Table of Contents

Table of Figures

Table of Examples

1. Introduction

In 2009 I wrote a paper entitled, "SystemVerilog Assertions - Design Tricks and SVA Bindfiles,"[3] so why seemingly write another paper on the same topic?

In 2010, along with SystemVerilog and Formal Verification expert, Harry Foster, I co-presented a series of seminars in North America and Europe on "SystemVerilog Assertion (SVA) Based Verification," to local design and verification engineering audiences as well as a few onsite, captive seminars for large companies. During the course of those seminars Harry and I were able to share our best SVA practices with seminar attendees and with each other. We also were given direct feedback, especially from large companies and power SVA users on their best SVA usage practices. Traveling with Harry gave me the opportunity to ask Harry, co-inventor of the OVL (Open Verification Library), co-author of the first Assertion Based Design book [2], Chairman of the Accellera Formal Verification Committee, and Chairman of the IEEE-1850 PSL[1] Working Group questions about why certain tokens and capabilities were added to both SVA and PSL.

Based on the information presented at the seminars, feedback shared with us from power-SVA users and conversations that Harry and I held with each other, I developed a new set of recommended SVA usage Best Practices.

In recent years other excellent papers on SVA usage have been shared and have similarly given useful recommendations [5][9]. This paper will detail new SVA coding practices and explain why I find these practices to be superior to other techniques that have been previously presented. Readers are invited to consider these guidelines, compare them to alternate guidelines offered by other respected colleagues and choose for themselves.

1.1 Assertion terminology

In this paper I will refer to the following abbreviations and the following different types of assertions:

DUT - Device Under Test.

SVA - SystemVerilog Assertions.

Immediate assertions - uses the keyword `assert` (not `assert property`), and is placed in procedural code and executed as a procedural statement.

Concurrent assertions - uses the keywords `assert property`, is placed outside of a procedural block and is executed once per sample cycle at the end of the cycle. The sample cycle is typically a `posedge clk` and sampling takes place at the end of the clock cycle, just before everything changes on the next `posedge clk`.

Embedded concurrent assertions - another form of concurrent assertions added to IEEE Std 1800-2009[7] and also uses the keywords `assert property` but is placed inside of a clocked `always` process. Placing the assertion in a clocked `always` process allows the concurrent assertion to inherit the clocking-sample signal from the `always` process.

See 0 for tool and Operating System versions used to test the examples in this paper.

[1] PSL - Property Specification Language and precursor to SVA

2. Design Engineers and SVA

One question that authors and experts have tried to address is, what types of assertions should be added by designers and what types of assertions should be added by verification engineers?

Some respected colleagues have suggested that immediate assertions should be added by designers and concurrent assertions should be added by verification engineers. I disagree.

I prefer the recommendation made by Harry Foster in the Assertion Based Verification seminars that Harry and I did back in 2010. Among other recommendations, Harry suggested that Design Engineers should create the low-level and simple assertions while Verification Engineers should create higher-level and perhaps more complex assertions (Harry had more distinctions regarding the separation of concerns between Design and Verification engineers not repeated in this paper - see [1]).

I too recommend that designers should generally add the low-level and simple assertions using concurrent assertions and in general should avoid immediate assertions.

If you want to encourage your design team to use assertions (and designers should be encouraged to use assertions) I have found that the following assertion coding goals should be followed:

- Start learning and using SVA after 2-3 hours of lecture and 1-3 hours of labs.
- Use ***bindfiles*** to add assertions to a design
- Use long, descriptive labels to:
  - document the assertions
  - accelerate debugging using waveform displays
- Use simple macros to:
  - efficiently add concise assertions
  - reduce assertion coding efforts
  - reduce assertion syntax errors
- Use concurrent assertions but avoid immediate assertions
- Use `|-> ##1` implications instead of `|=>` implications

Each of these recommendations, and more, will be described in this paper.

3. How much SVA training?

In recent years I have been called on to conduct SystemVerilog Assertion (SVA) training for companies that had previously taken multi-day SVA training, not because the training they had received was bad, but because the training they had received was too much and their engineering teams had a hard time remembering all of the SVA options and syntax possibilities. The problem I have seen is that designers use SVA sporadically for a few months on one project, then they might go many months before they need to use it again. This is a classic case of unless a skill is practiced, it is forgotten.

The SVA constructs in the SystemVerilog language are powerful and provide extensive assertion capabilities, which unfortunately for many design engineers is perhaps their greatest disadvantage. Unless an engineer has a fulltime job adding assertions to all of the projects at a company, then those engineers should focus on a subset of the SVA syntax and capabilities and add additional SVA capabilities as needed.

My experience has been that if a design engineer has to write 3 or more lines of assertion code to test each DUT feature, they will quickly abandon the use of SVA and will designate the task to be done by verification engineers. Concise, syntax error-avoidance coding styles are required to make a design engineer productive and keep them enthusiastic about adding SVA to their RTL designs.

My experience has shown that 2-3 hours of SVA lecture followed by 1-3 hours of SVA lab work can make design engineers both productive and enthusiastic about using SVA in their design work.

If an engineer ***DOES*** have the full-time job to add and support assertions on multiple projects, ***that*** engineer should take multi-day SVA training.

Guideline #1: Start learning and using SVA after 2-3 hours of lecture and 1-3 hours of labs.

4. Bindfiles

Let me summarize this section with two short statements.

Guideline #2: ***bindfiles*** - use them!

Guideline #3: Inline SVA code - avoid it!

This section will describe ***bindfile*** usage and also explain why I discourage placing assertions directly in the RTL code.

There are many sources that recommend embedding assertions directly into the RTL code, but again, I believe there is a better approach. Harry Foster, author of one of the first Assertion Based Design books, used to recommend putting assertions directly into the RTL code, but he now recommends NOT putting the assertions in the RTL code, and Harry and I are in agreement that assertions should instead be put in a separate ***bindfile***. Harry and I gave Assertion Based Verification seminars in North America and Europe in 2010 where we shared SVA Best Coding Practice techniques, including the proper use of ***bindfiles***.

4.1 How bindfiles work

In general, using ***bindfiles*** is actually doing indirect instantiation. The engineer will bind (indirectly instantiate) one **`module`** inside of another **`module`** using the **`bind`** keyword. The IEEE Std 1800-2012 Standard frequently refers to ***bindfiles*** as "*bind_instantiation*."[8]

An RTL designer might not even know that the ***bindfile*** has been instantiated into the RTL design unless the engineer opens a waveform viewer, like DVE, to see that the RTL design now has another sub-level of hierarchy that can be displayed in the waveform display. A new and perhaps unexpected level of hierarchy is the indirectly instantiated ***bindfile***.

To create a ***bindfile***, declare a **`module`** that will encapsulate the assertion code (and other verification code if needed). The **`module`** needs access to all of the important signals in the enclosing file so all of the ports and internal signals from the enclosing file are declared as **`inputs`** to the ***bindfile***.

4.2 The bind command

The **`bind`** command can be viewed as a 2-box command.

`bind fifo1`	`fifo1_asserts p1 (.*);`

Figure 1 - bind - 2-box / 2-part command

As shown in Figure 1, the first box of the **`bind`** command includes the **`bind`** keyword followed by the DUT **`module`** name, the DUT instance name, or both.

In the second box is an instantiation command that describes how the bound `module` would be instantiated if placed directly in the `module` being bound to. Simply put, if you had instantiated the ***bindfile*** directly into a `module`, what would the instantiation code be? That instantiation code is the second box in the `bind` command. Binding is simply indirect instantiation.

When creating ***bindfiles***, it is a good idea to copy the DUT **`module`** to a DUT_asserts **`module`**, keep all existing **`input`** declarations, change all **`output`** declarations to **`input`** declarations, and declare all internal signals as **`input`** declarations to the ***bindfile***. The ***bindfile*** will sample the port and internal signals from the DUT. In Example 1 the **`fifo1.sv`** file was copied to the **`fifo1_asserts.sv`** file, the **`dout output`** was changed to a **`dout input`** declaration, all **`input`** declarations were kept as originally declared, and all internal **`logic`** signals were modified to be **`input logic`** declarations. There was no need to copy the internal array from the **`fifo1`** design so that was omitted. The **`` `include "assert_macros.sv" ``** command with corresponding macro definitions is explained in Section 8.

```
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] fifo1mem [0:15];
  logic [3:0] wptr, rptr;
  logic [3:0] cnt;
```

```
module fifo1_asserts (
  input  logic [7:0] dout,
  input  logic       full, empty,
  input  logic       write, read,
                     clk, rst_n,
  input  logic [7:0] din,


  input  logic [3:0] wptr, rptr,
  input  logic [3:0] cnt);

  `include "assert_macros.sv"
```

Example 1 - DUT module header and corresponding DUT_asserts module header

It is not required to list all of the DUT signals in the asserts file, only those signals that will be checked by assertions; however, it is highly recommend to add *ALL* of the DUT signals to the asserts file because it is common to add more assertions in the future that might require previously unused DUT signals.

If you match all of the DUT signal names (outputs, inputs and internal signals) and convert them into `input` declarations on the asserts file, it is very simple to `bind` the asserts file to the DUT because all port connections can be made using `.*` implicit port connections. For those engineers who are uncomfortable using the `.*` port connections, all of the connections can also be made using the SystemVerilog `.name` or Verilog named port connections. (see Cummings `.*` paper[4])

4.3 Bindfile usage models

Bindfile usage is explained in section 23.11 of the IEEE Std 1800-2012 [8].

There are three ***bindfile*** usage models: (1) bind to all instances of a module, (2) bind to a specific instance of a DUT, and (3) bind to an instance name without specifying the DUT module name. Usage, advantages and disadvantages of these three use models are described below and in Appendix 3 and Appendix 4.

In all three usage models, the box #2 instantiation code is the same. The only visible differentiation is in the box #1 code.

4.3.1 Bind to all instances of a module

Guideline #4: Use the **bind** command style that binds to all DUT modules, not the **bind** style that only binds to specified instances.

This style specifies the target module name to bind to but does not specify an instance name.

```
bind fifo1 fifo1_asserts p1 (.*);
```

Example 2 - Recommended bind style - binds to all instances of a DUT

Example 2 shows how to bind the **fifo1_asserts** file to all instances of the **fifo1** design. This style does not require that the **bind** command be scoped to the same module as any instance of the **fifo1**. It does not matter where the **fifo1** modules are instantiated because this **bind** command indirectly instantiates the assertions to the **fifo1** designs regardless of where they are placed in the design hierarchy.

The behavior is the same as if all of the assertions had been added to the DUT module directly and each instance of the DUT will have the same assertion checking during simulation. In general, engineers should do assertion checking on every instance of a DUT module because an engineer never knows which DUT might fail.

Another undocumented advantage of binding to any DUT module is that it does not matter where this **bind** command is placed. It is common practice to place this **bind** command in the top-level testbench but I prefer to place this **bind** command in a standalone module called `bindfiles` in a file called **bindfiles.sv**

```
module bindfiles;
  bind fifo1 fifo1_assert p1 (.*);
endmodule
```

Example 3 - Separate bindfiles module - bindfiles.sv

I also keep two copies of my simulation **run1.f** command file, one that does not call the ***bindfiles*.sv** file and one that does.

```
tb1.sv
fifo1.sv
```

Example 4 - run1.f command file - no assertions

```
tb1.sv
fifo1.sv
bindfiles.sv
fifo1_assert.sv
```

Example 5 - run2.f command file - bind assertions

Verilog-2001[6] made it legal to define and simulate two top-level designs. When I simulate with the **run2.f** command file, I am simulating both the design and the separate ***bindfiles*** module. The VCS simulation log in Figure 2 shows that there were two **"Top Level Modules: tb1 bindfiles,"** and indeed the when this simulation was run, it reported failing assertions (not shown).

```
Command: vcs -R -sverilog -f run1.f -l logfile -full64
                         Chronologic VCS (TM)
       Version K-2015.09-SP1_Full64 -- Wed Dec 16 11:33:13 2015
               Copyright (c) 1991-2015 by Synopsys Inc.
                         ALL RIGHTS RESERVED

This program is proprietary and confidential information of Synopsys Inc.
and may be used and disclosed only as authorized in a license agreement
controlling such use and disclosure.

Parsing design file '../SIMUTIL.v'
Parsing design file 'tb1.sv'
Parsing design file 'fifo1.sv'
Parsing design file 'bindfiles.sv'
Parsing design file 'fifo1_assert.sv'
Parsing included file 'assert_macros.sv'.
Back to file 'fifo1_assert.sv'.
Top Level Modules:
       tb1
       bindfiles
TimeScale is 1 ns / 1 ns
Starting vcs inline pass...
```

Figure 2 - Multiple top-level modules - tb1 and bindfiles

If I want to run without assertions, I can simply call the **`run1.f`** command file to run the simulation.

Having a separate **`bindfiles`** module allows different design and verification teams to put multiple **`bind`** commands in this **`bindfiles`** module without touching the RTL design or testbench. Any **`Makefile`** that is controlling simulation and synthesis will completely ignore the **`bindfiles`** module.

It should also be noted that the **`bindfiles`** module requires no additional signal declarations. This makes using the ***bindfiles*** approach very easy to do. The signal declarations must exist in the file that is being bound to because the **`bind`** is an indirect instantiation into that file, but since the **`bind`** is to another file and not the **`bindfiles`** module, the signal declarations do not and should not be declared in the **`bindfiles`** module.

4.3.2 Bind to specific DUT instance with or without using the module name

Recommendation: Do not use these styles.

It is possible to **`bind`** to a specific instance of a DUT using both the **`module`** name and instance name, or to **`bind`** to a specific locally-scoped instance name without referencing the **`module`** name corresponding to the instance name.

It is generally not wise to limit assertion checking to just one copy of a **`module`** or to a generic instance name in a local scope because a designer might **`bind`** to a working version of a **`module`** while another instance of the same **`module`** could be failing elsewhere in the design. Binding to the **`module`** name without limiting to just one instance allows checking to proceed on all instances of the **`module`**.

Reference Appendix 3 and Appendix 4 to see how other non-recommended bind-styles work.

4.4 Bindfiles for parameterized models

If an engineer is unsure how to use ***bindfiles*** especially when the engineer is uncertain how the instantiation will appear in a **bind** command, I recommend following these three steps:

(1) Put the assertions directly into the RTL design and figure out how they would work there.

(2) Then move the assertions to a ***bindfile*** and instantiate the ***bindfile*** directly into the DUT and make that work, as shown in Example 6.

```
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] fifo1mem [0:15];
  logic [3:0] wptr, rptr;
  logic [3:0] cnt;

  ...
  fifo1_assert p1 (.*);  // The assertion module could be
                         // instantiated here
endmodule
```

Example 6 - fifo1 module with assertion module directly instantiated

(3) Then remove the instantiated module and place the instance-code into the **bindfiles.sv** file with the **bind** command.

These steps are also sound advice when considering how to make ***bindfiles*** work with parameterized modules that might be instantiated multiple times with different parameters.

Consider the overly simple (but easy to understand) definition of a register module with **SIZE** parameter as shown in Example 7.

```
module register #(SIZE=8) (
  output logic [SIZE-1:0] dout,
  input  logic [SIZE-1:0] din,
  input  logic            clk, rst_n);

  always_ff @(posedge clk or negedge rst_n)
    if (!rst_n) dout = '0;
    else        dout = din;
endmodule
```

Example 7 - Parameterized register module

If three instances of this register are placed into a larger DUT with three different parameter values as shown in Example 8, this might seem to pose a problem for an assertion ***bindfile***.

```
module DUT (
  output logic [11:0] q,
  input  logic [11:0] d,
  input  logic        clk, rst_n);

  logic [3:0] n1;
  logic [7:0] n0;

  register              r1 (.dout(n0), .din(d[11:4]), .*);
  register #(.SIZE( 4)) r2 (.dout(n1), .din(d[ 3:0]), .*);
  register #(.SIZE(12)) r3 (.dout(q),  .din({n0,n1}), .*);

endmodule
```

Example 8 - DUT module with three uniquely parameterized register instances

In reality, different parameter values do not pose a problem. For the DUT module of Example 8, what is required is a ***bindfile*** that is also parameterized as shown in Example 9.

```
`include "assert_macros.sv"

module register_asserts #(SIZE=8) (
  input  logic [SIZE-1:0] dout,
  input  logic [SIZE-1:0] din,
  input  logic            clk, rst_n);

  ERR_dout_value_does_not_follow_din:
  `assert_clk_xrst ($changed(din) |-> ##1 (dout == $past(din)));

  ERR_dout_did_not_reset:
  `assert_clk (!rst_n |-> (dout == '0));
endmodule
```

Example 9 - Parameterized register_asserts module

This assertion module is rather contrived because it typically is not worthwhile to check a register to ensure that inputs were transferred to the outputs on a clock edge, but this overall example demonstrates the point that assertion files can be parameterized the same as any other module.

4.5 Bindfiles with .* port connections

As mentioned earlier, a properly coded asserts file can be very easily connected to the DUT using SystemVerilog's `.*` connections.

Unfortunately, SystemVerilog does not allow `.*` parameter matching so parameters must still be passed by name. Allowing `.*` parameter matching would be a welcome enhancement to SystemVerilog.

4.6 No bindfile nesting

From the IEEE Std 1800-2012:

"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 you bind a file to one module and then bind that module to another module, you have created a nested ***bindfile***, which is illegal per the IEEE Std 1800-2012. That having been said, I have successfully nested ***bindfiles*** using multiple tools, including VCS, without any reported errors. As I have discussed this inconsistency with EDA vendors, the vendors have emphasized that nested binding is technically illegal and may not be supported in future versions of their EDA tools.

In short, because nested binding is illegal per the IEEE Std 1800-2012, vendors can remove the nested ***bindfile*** capability at any time, so it is advised to avoid using nested ***bindfiles*** even if it appears to work.

5. Inline assertion code

SystemVerilog allows SVA to be added directly to the RTL code and one of the first books on Assertion Based Design by Harry Foster, Adam Krolnik and David Lacey [2], did indeed recommend that assertions be added directly to the RTL code and to place the assertions near to the RTL that the assertions were intended to check.

Other authors in recent years have similarly recommended adding assertions directly to the RTL code.

While doing the ABV seminars in 2010, I noticed that Harry Foster was telling the seminar attendees to NOT put the assertions into their RTL designs. At one of the seminars I asked Harry why he was contradicting recommendations from his own book. Harry mentioned two developments in recent years that had caused him to change the book recommendation: (1) engineers fighting with tools that did not support SVA, and (2) engineering teams that use `Makefiles` to control large simulations and large synthesis runs. Both are explained below.

I will also detail a third problem that an engineer experiences when adding assertions close to the RTL code. In practice, adding assertions close to the RTL code that they are intended to test is easier said than done.

5.1 Frontend tool SVA support

In theory, simulators and formal checking tools should read and support SVA, and on whole most of these tools do support SVA. Most other tools should quietly ignore any SVA constructs in the RTL code, but that is not universally true.

At DVCon 2015, my colleague, Stuart Sutherland, accurately showed a table that listed three different vendors (anonymously listed to protect the guilty!) and which vendor tools properly supported SVA capabilities. Although there was broad support for SVA capabilities, most of the tools had troubles with the `checker` statement and some had troubles with the `let` statement, two capabilities added to SystemVerilog 2009.

At SNUG-SV 2015[9], Stu Sutherland, also showed a table that listed SVA support by VCS (v. 2014.12), Design Compiler (v.2014.09-SP5) and Synplify-Pro (v.2014.09-SP2), again showing there was broad support for SVA capabilities, but that the tools still had support deficiencies in a few areas.

If engineers add SVA checking through a ***bindfile***, all of these support issues generally disappear. The RTL, without SVA, is pristine SystemVerilog code and there are no stray SVA keywords to cause a synthesis tool, or any other front-end tool to abort with errors. There could still be simulator support issues that are not covered by using a ***bindfile***.

The point is ***bindfiles*** remove all of the non-simulation support issues related to the RTL design.

5.2 Makefiles

Perhaps the biggest reason to avoid adding SVA directly to the RTL code is `Makefiles`.

Harry Foster and I heard from large companies that large projects often control simulation and synthesis runs through the use of `Makefiles`. Engineering teams put the RTL code under revision control and an engineer might checkout the design and execute the `Makefile` for simulation or synthesis. If there have been no design changes, no simulation will be run and no synthesis tools will be called to re-synthesize the design. The `Makefile` ensures that the most current design has been simulated and synthesized without tying up expensive simulation or synthesis licenses when unnecessary.

The problem that exists with projects that allow engineers to add assertions directly to the RTL code is that any time an engineer adds a new assertion to one of the RTL files, the timestamp on the file is updated and calling the `Makefile` for either simulation or synthesis can cause a lengthy simulation or synthesis run to start even though there were no changes to the actual design. This could unnecessarily tie up tool licenses for hours.

There are some companies that allow their engineers to add inline assertions to the RTL until the design is put under `Makefile` control. After that, engineers are then encouraged to put remaining assertions into a ***bindfile***. Yet other companies allow the ***bindfile*** assertions to be transferred into the RTL design when there is an actual design change, but managing that scenario can prove tricky.

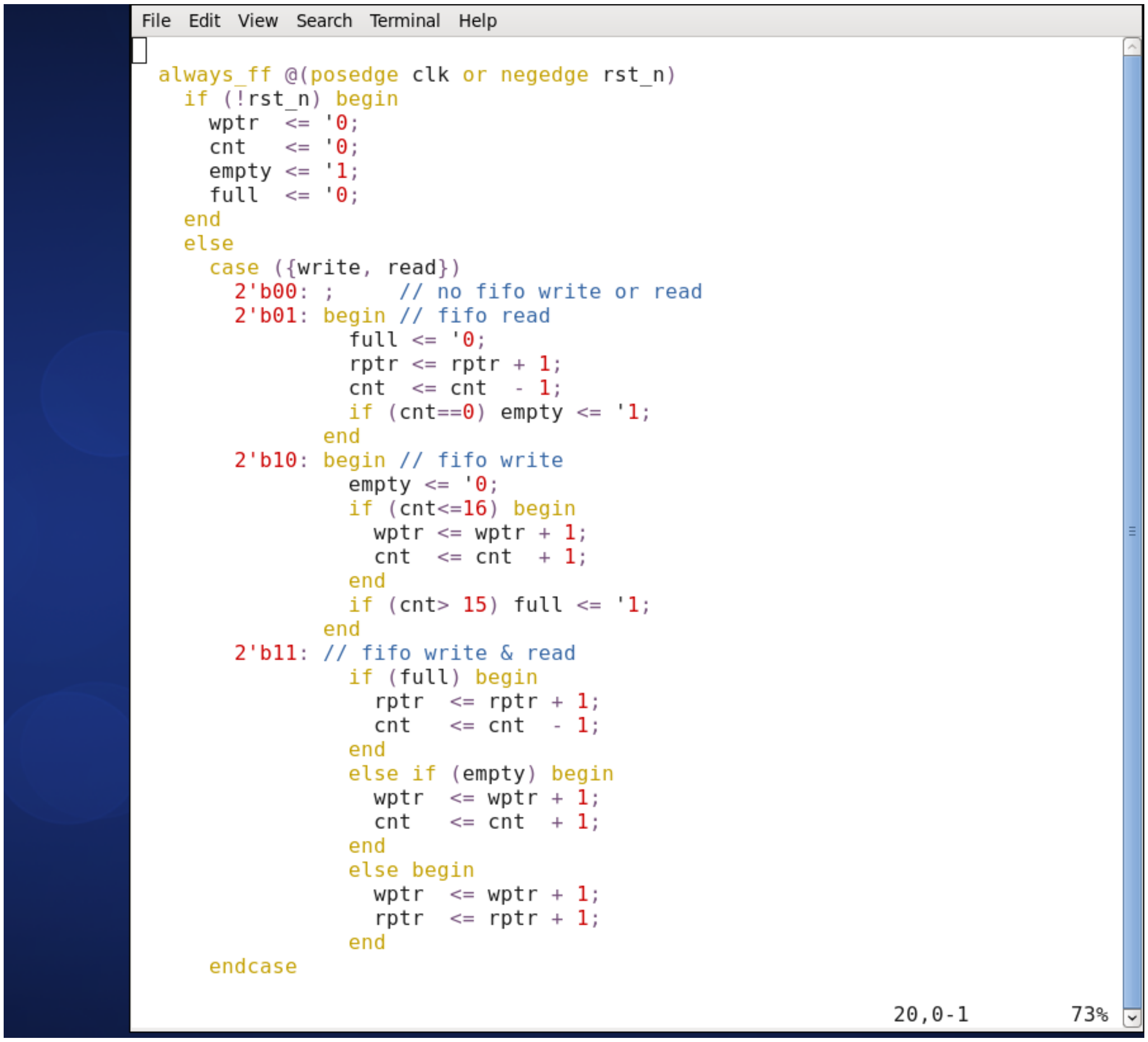
The ***bindfile*** eliminates this potential problem. The ***bindfile*** is not tested by the `Makefile` to initiate long regression simulations, nor is it tested by `Makefiles` for synthesis execution. This is a simple solution to the `Makefile` problem and the reason that many large companies have moved all assertion code to a separate ***bindfile***.

5.3 Visibility of SVA in bindfiles

One justification sometimes mentioned for not using ***bindfiles*** is that it distances the SVA code from the RTL code and that separation makes it hard to do cross-reference debugging between the design and assertion code. My experience has been just the opposite of this situation.

It has been assumed that placing an assertion next to the RTL that it is intended to check makes it easier to see and debug. It has also been claimed that adding immediate assertions to the RTL also makes the assertion easier to use for debugging. In practice I have not found either of these claims to be true.

Consider the buggy `fifo1` code show below. This is a screen shot from my laptop of a large portion of the `fifo1` code that I use in a training lab.

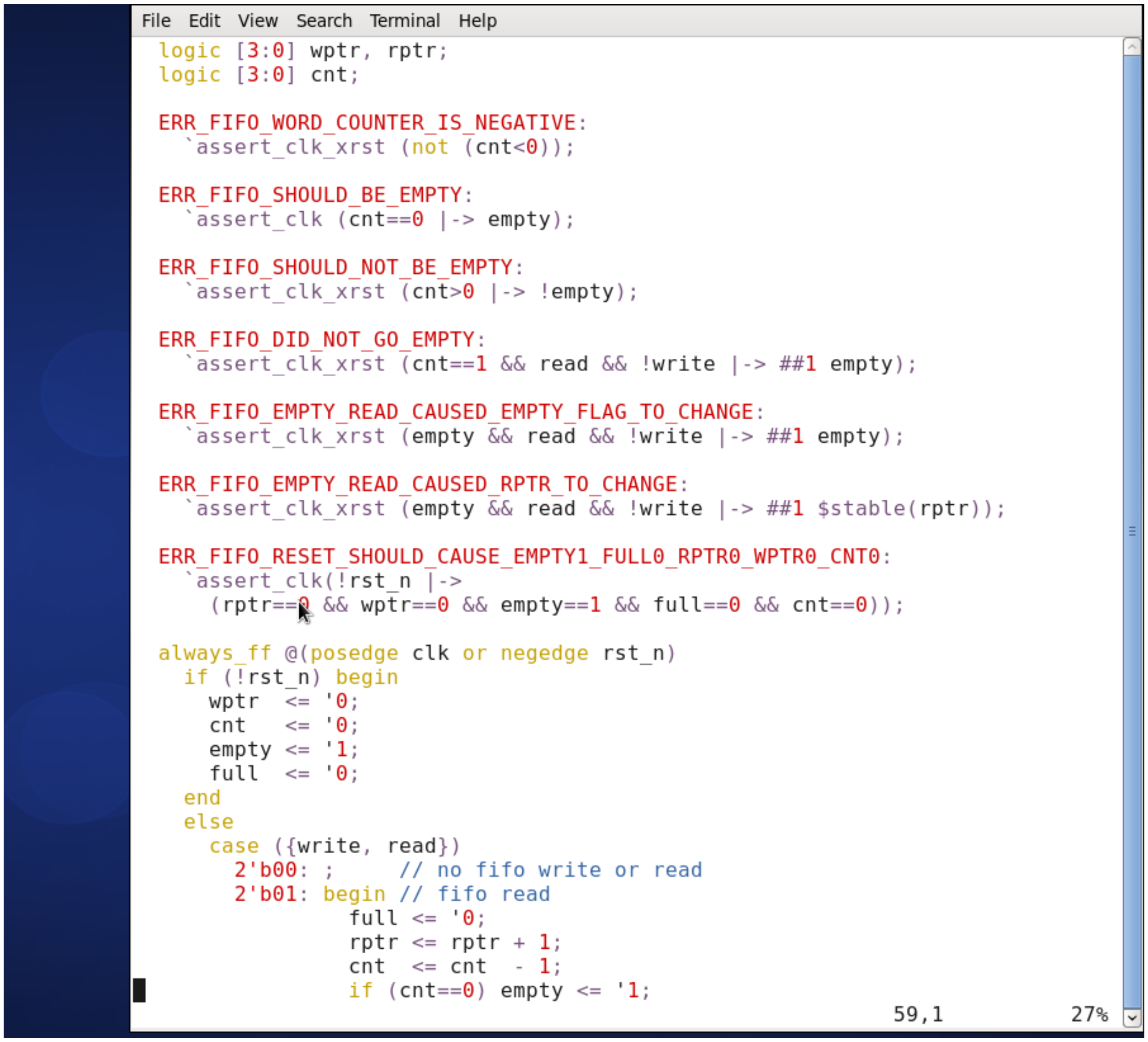


```
always_ff @(posedge clk or negedge rst_n)
  if (!rst_n) begin
    wptr  <= '0;
    cnt   <= '0;
    empty <= '1;
    full  <= '0;
  end
  else
    case ({write, read})
      2'b00: ;      // no fifo write or read
      2'b01: begin // fifo read
               full <= '0;
               rptr <= rptr + 1;
               cnt  <= cnt  - 1;
               if (cnt==0) empty <= '1;
             end
      2'b10: begin // fifo write
               empty <= '0;
               if (cnt<=16) begin
                 wptr <= wptr + 1;
                 cnt  <= cnt  + 1;
               end
               if (cnt> 15) full <= '1;
             end
      2'b11: // fifo write & read
               if (full) begin
                 rptr  <= rptr + 1;
                 cnt   <= cnt  - 1;
               end
               else if (empty) begin
                 wptr  <= wptr + 1;
                 cnt   <= cnt  + 1;
               end
               else begin
                 wptr  <= wptr + 1;
                 rptr  <= rptr + 1;
               end
    endcase
                                                  20,0-1        73%
```

Figure 3 - fifo1 module always_ff code

As is true with many RTL designs, it is common to find large **`always`** processes in the DUT with multiple lines of code.

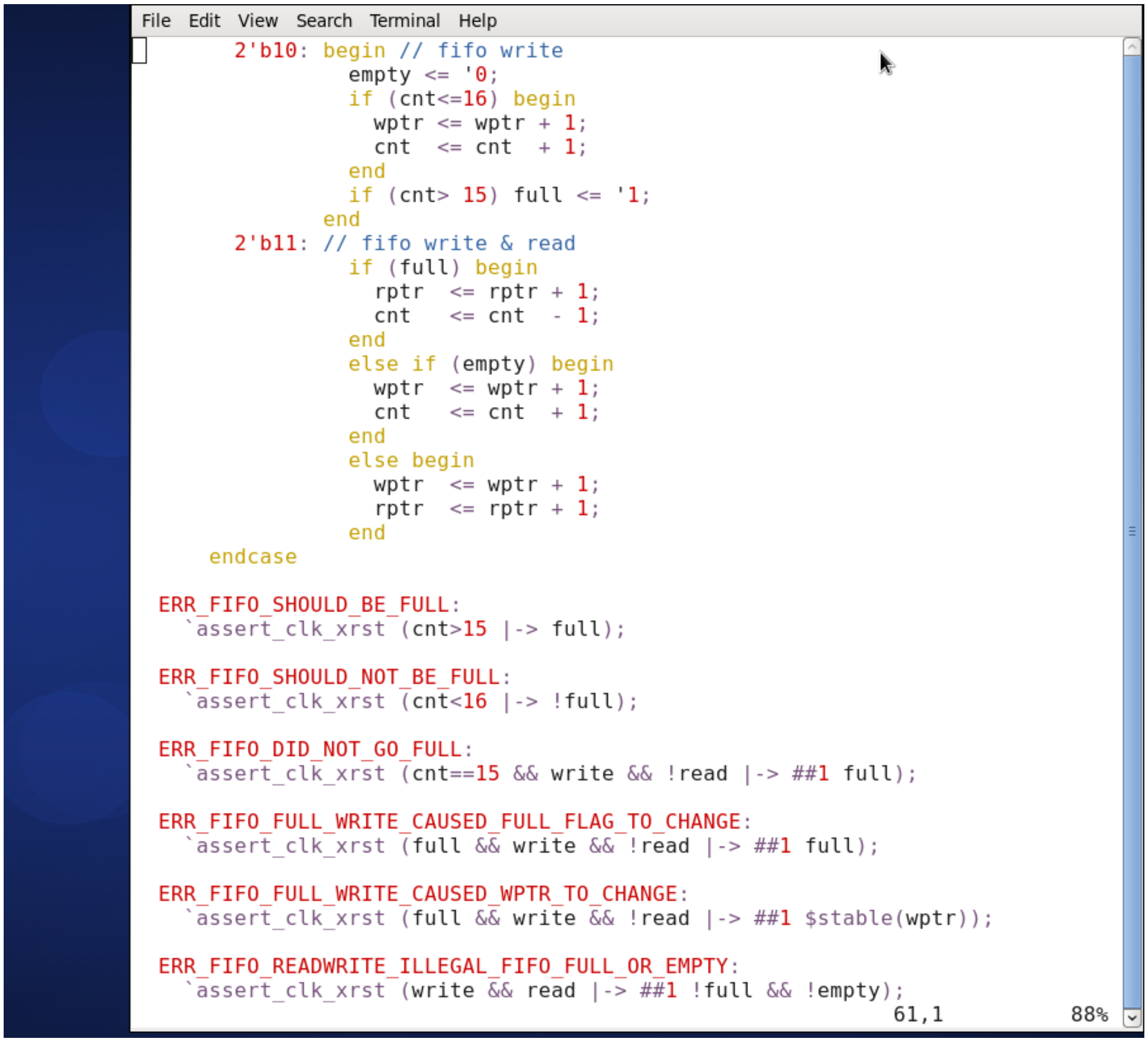
Trying to place multiple concurrent assertions related to this **`fifo1`** close to the code that the assertions should check is not an easy task. Concurrent assertions cannot be placed inside of the process so some of the assertions would be placed before the process, as shown in Figure 4.



```
logic [3:0] wptr, rptr;
logic [3:0] cnt;

ERR_FIFO_WORD_COUNTER_IS_NEGATIVE:
  `assert_clk_xrst (not (cnt<0));

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

ERR_FIFO_SHOULD_NOT_BE_EMPTY:
  `assert_clk_xrst (cnt>0 |-> !empty);

ERR_FIFO_DID_NOT_GO_EMPTY:
  `assert_clk_xrst (cnt==1 && read && !write |-> ##1 empty);

ERR_FIFO_EMPTY_READ_CAUSED_EMPTY_FLAG_TO_CHANGE:
  `assert_clk_xrst (empty && read && !write |-> ##1 empty);

ERR_FIFO_EMPTY_READ_CAUSED_RPTR_TO_CHANGE:
  `assert_clk_xrst (empty && read && !write |-> ##1 $stable(rptr));

ERR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0_RPTR0_WPTR0_CNT0:
  `assert_clk(!rst_n |->
    (rptr==0 && wptr==0 && empty==1 && full==0 && cnt==0));

always_ff @(posedge clk or negedge rst_n)
  if (!rst_n) begin
    wptr  <= '0;
    cnt   <= '0;
    empty <= '1;
    full  <= '0;
  end
  else
    case ({write, read})
      2'b00: ;     // no fifo write or read
      2'b01: begin // fifo read
               full <= '0;
               rptr <= rptr + 1;
               cnt  <= cnt  - 1;
               if (cnt==0) empty <= '1;
                                                        59,1          27%
```

Figure 4 - fifo1 module with assertions embedded above the always_ff code

Yet other assertions would be placed after the **`always`** process as shown in Figure 5.



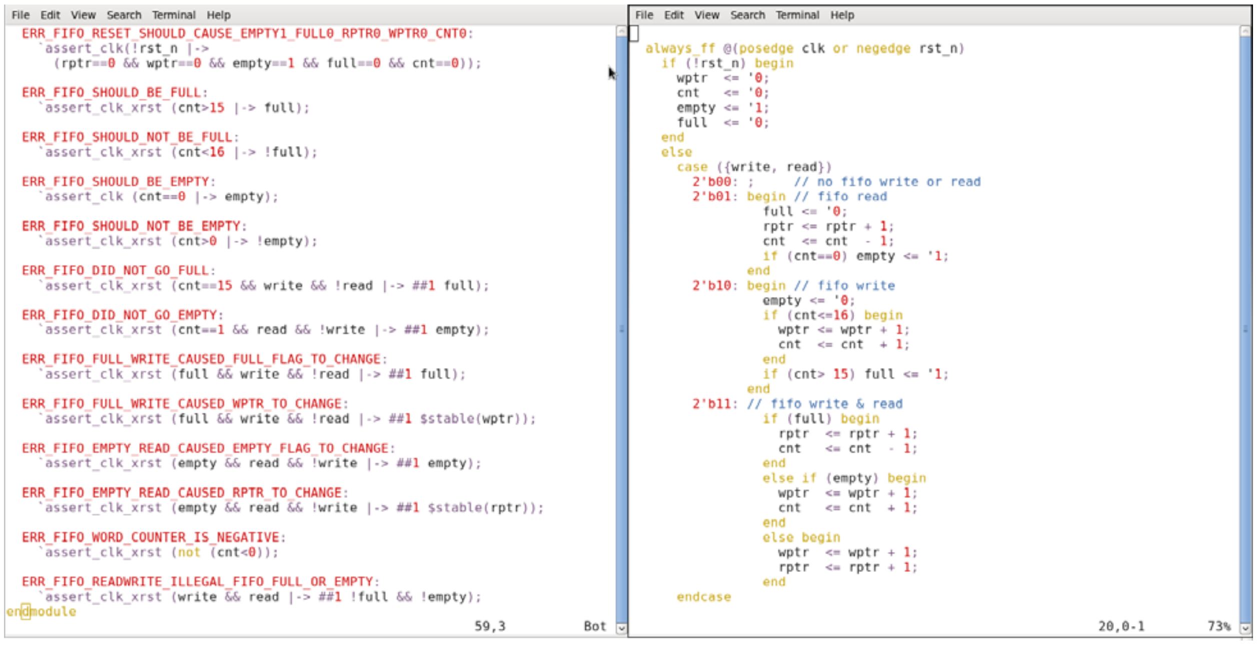
```
        2'b10: begin // fifo write
                 empty <= '0;
                 if (cnt<=16) begin
                   wptr <= wptr + 1;
                   cnt  <= cnt  + 1;
                 end
                 if (cnt> 15) full <= '1;
               end
        2'b11: // fifo write & read
                 if (full) begin
                   rptr   <= rptr + 1;
                   cnt    <= cnt  - 1;
                 end
                 else if (empty) begin
                   wptr   <= wptr + 1;
                   cnt    <= cnt  + 1;
                 end
                 else begin
                   wptr   <= wptr + 1;
                   rptr   <= rptr + 1;
                 end
      endcase

  ERR_FIFO_SHOULD_BE_FULL:
    `assert_clk_xrst (cnt>15 |-> full);

  ERR_FIFO_SHOULD_NOT_BE_FULL:
    `assert_clk_xrst (cnt<16 |-> !full);

  ERR_FIFO_DID_NOT_GO_FULL:
    `assert_clk_xrst (cnt==15 && write && !read |-> ##1 full);

  ERR_FIFO_FULL_WRITE_CAUSED_FULL_FLAG_TO_CHANGE:
    `assert_clk_xrst (full && write && !read |-> ##1 full);

  ERR_FIFO_FULL_WRITE_CAUSED_WPTR_TO_CHANGE:
    `assert_clk_xrst (full && write && !read |-> ##1 $stable(wptr));

  ERR_FIFO_READWRITE_ILLEGAL_FIFO_FULL_OR_EMPTY:
    `assert_clk_xrst (write && read |-> ##1 !full && !empty);
                                                     61,1          88%
```

Figure 5 - fifo1 module with assertions embedded below the always_ff code

Although an engineer can add inline embedded concurrent assertions (an enhancement added to SystemVerilog 2009[7]) and immediate assertions, both would further expand the large block of code to spread the code over multiple screens.

It should also be noted that my experience has shown that assertions often are not easily placed next to all of the pertinent procedural code in the design. Assertions frequently test the interaction of code from different blocks in the RTL design, and often the designer has to page up and down between different pages of code to examine part of the RTL design and the assertion that interacts with that part of the design.

On the other hand, my experience has shown that engineers typically now work with wide-screen laptops or a laptop and second screen, or both. Below is a screen shot from my laptop where I am working with the ***bindfile*** assertions in one screen and the DUT code in the second screen.



```
 ERR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0_RPTR0_WPTR0_CNT0:
   `assert_clk(!rst_n |->
     (rptr==0 && wptr==0 && empty==1 && full==0 && cnt==0));

 ERR_FIFO_SHOULD_BE_FULL:
   `assert_clk_xrst (cnt>15 |-> full);

 ERR_FIFO_SHOULD_NOT_BE_FULL:
   `assert_clk_xrst (cnt<16 |-> !full);

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

 ERR_FIFO_SHOULD_NOT_BE_EMPTY:
   `assert_clk_xrst (cnt>0 |-> !empty);

 ERR_FIFO_DID_NOT_GO_FULL:
   `assert_clk_xrst (cnt==15 && write && !read |-> ##1 full);

 ERR_FIFO_DID_NOT_GO_EMPTY:
   `assert_clk_xrst (cnt==1 && read && !write |-> ##1 empty);

 ERR_FIFO_FULL_WRITE_CAUSED_FULL_FLAG_TO_CHANGE:
   `assert_clk_xrst (full && write && !read |-> ##1 full);

 ERR_FIFO_FULL_WRITE_CAUSED_WPTR_TO_CHANGE:
   `assert_clk_xrst (full && write && !read |-> ##1 $stable(wptr));

 ERR_FIFO_EMPTY_READ_CAUSED_EMPTY_FLAG_TO_CHANGE:
   `assert_clk_xrst (empty && read && !write |-> ##1 empty);

 ERR_FIFO_EMPTY_READ_CAUSED_RPTR_TO_CHANGE:
   `assert_clk_xrst (empty && read && !write |-> ##1 $stable(rptr));

 ERR_FIFO_WORD_COUNTER_IS_NEGATIVE:
   `assert_clk_xrst (not (cnt<0));

 ERR_FIFO_READWRITE_ILLEGAL_FIFO_FULL_OR_EMPTY:
   `assert_clk_xrst (write && read |-> ##1 !full && !empty);
endmodule
                                                    59,3          Bot
```

```
  always_ff @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      wptr  <= '0;
      cnt   <= '0;
      empty <= '1;
      full  <= '0;
    end
    else
      case ({write, read})
        2'b00: ;     // no fifo write or read
        2'b01: begin // fifo read
                 full <= '0;
                 rptr <= rptr + 1;
                 cnt  <= cnt  - 1;
                 if (cnt==0) empty <= '1;
               end
        2'b10: begin // fifo write
                 empty <= '0;
                 if (cnt<=16) begin
                   wptr <= wptr + 1;
                   cnt  <= cnt  + 1;
                 end
                 if (cnt> 15) full <= '1;
               end
        2'b11: // fifo write & read
                 if (full) begin
                   rptr  <= rptr + 1;
                   cnt   <= cnt  - 1;
                 end
                 else if (empty) begin
                   wptr  <= wptr + 1;
                   cnt   <= cnt  + 1;
                 end
                 else begin
                   wptr  <= wptr + 1;
                   rptr  <= rptr + 1;
                 end
      endcase
                                                    20,0-1        73%
```

Figure 6 - assertions in separate file - side-by-side windows to help debug the DUT

With this very common wide-screen setup, I can easily place one or more assertions in my left-screen and scroll through my RTL code on the right screen, which is now more concise and fits on fewer pages. If an assertion relies on interactions between multiple procedural blocks in different sections of my RTL code, I can keep the applicable assertion visible in the left screen while tracing through different blocks of RTL code in the right screen. I find this type of RTL debugging with assertions to be much easier to handle than placing an assertion next to part of the RTL design and paging to the next part of my RTL design where the assertion is no longer visible. From my experience, I have found that debugging is much easier to do when I can keep my assertions visible in an adjoining terminal.

While conducting SVA training, I have similarly found that my students can debug the **`fifo1`** design quicker if they keep the assertions visible in one of the terminal windows.

5.4 Bindfile disadvantages

There are a couple of disadvantages to using ***bindfiles***:

(1) The ***bindfile*** code is not visible when the dutfile is viewed. Since assertions document the correct behavior of the DUT, placing assertions in a separate ***bindfile*** moves useful DUT documentation to a separate file. This is why many engineers prefer to add the assertions directly to the DUT code.

(2) ***bindfiles*** can slow simulation performance. A separate ***bindfile*** can require the simulation compiler to keep bound signals visible, by contrast removing the ***bindfile*** might allow a simulation compiler to remove some internal signal details that are unnecessary for optimized simulation. If improving simulation performance during a regression run is desired, engineers can use the two different **`run.f`** files shown in Example 4 and Example 5 to run with or without the bound assertions. The tradeoff is that a long regression simulation executed without the assertions could fail and the absence of the assertions would either make bug identification harder or require the long simulation to be repeated with the ***bindfile***.

6. Assertion labels for debugging

Guideline #5: Add descriptive labels to your assertion code.

Guideline #6: In general, do not use **`$error(...)`** or **`$display(...)`** messages in assertion action blocks.

Use long, descriptive labels to:

- Document the assertions
- Accelerate debugging using waveform displays

Adding labels to assertions is optional but highly recommended to accelerate assertion and DUT debugging. The labels help debug much better than using assertion action-block **`$error(...)`** or **`$display(...)`** commands. How is this possible?

First note that there is nothing in the IEEE Std 1800-2012 that describes how a vendor must display useful assertion details when assertion error messages are displayed, but most vendors will include the assertion label in displayed assertion failure messages. Now let's look at how VCS handles assertion error messages (most EDA tools show similar assertion error information).

Using assertion action-block **`$error(...)`** or **`$display(...)`** commands will cause the error messages to be displayed in the simulation transcript window, but those errors do not show up in waveform displays. Engineers will need to read the transcript error and timing messages to figure out where to look in the waveform display.

Conversely, adding descriptive labels to the assertions causes those labels to be displayed by the default assertion error messages in the simulation transcript window ***AND*** they also show up in the waveform display with the assertion label visible as shown in Figure 7.

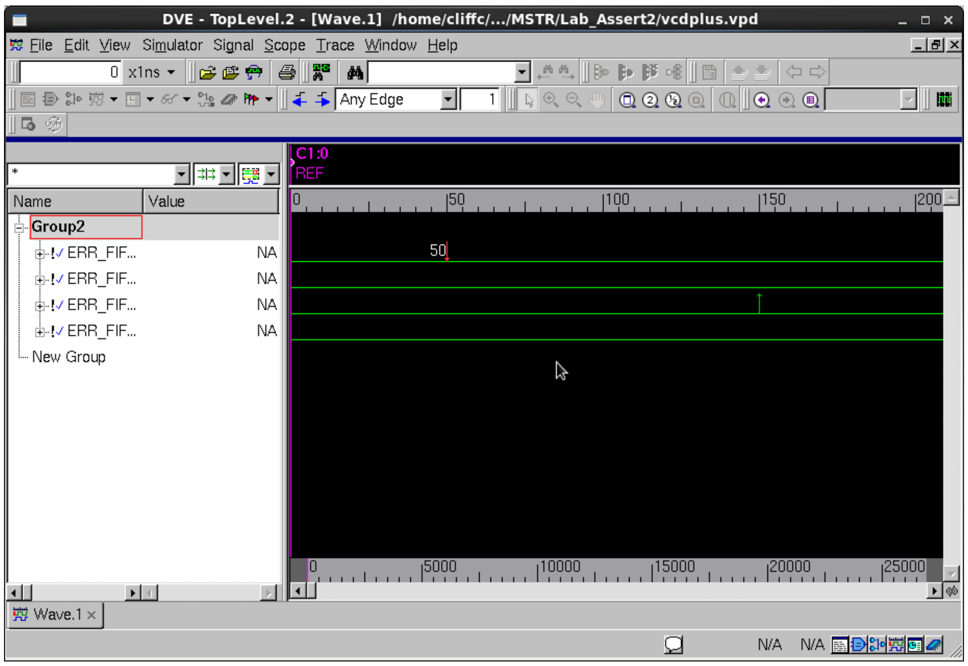


Figure 7 - Assertion labels displayed in the waveform display (before expanding the Name pane)

Note that in Figure 7, the label names are truncated by the size of the **Name** window pane. This is typical when the assertions are first added to the waveform window. The next step is to grow the **Name** pane so that we can see the full assertion label names.

There are two ways to grow the **Name** pane - the ***annoying-beginner way*** or the ***easy-expert way***.

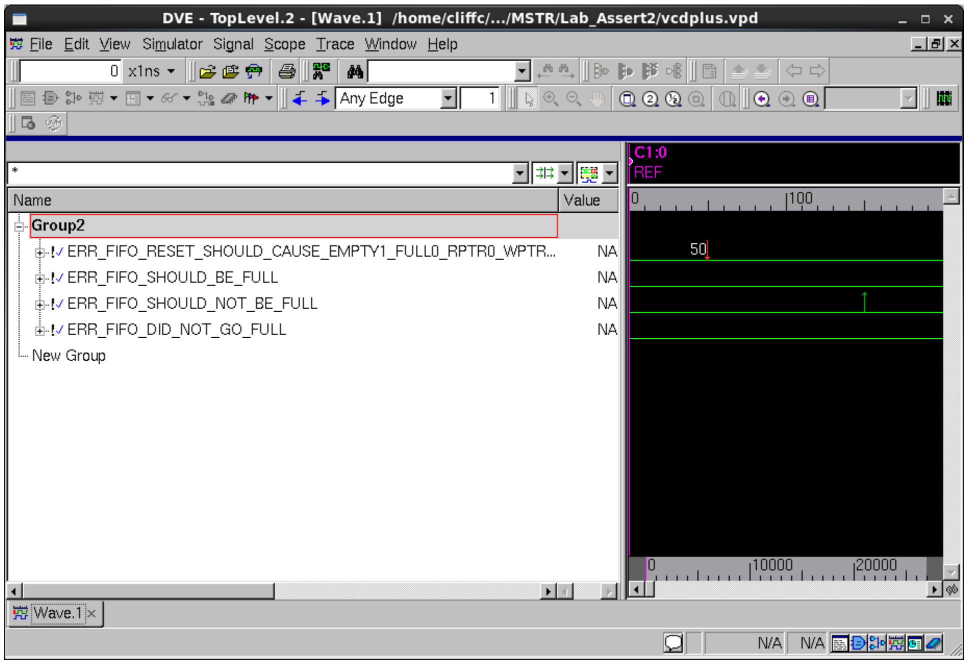
The ***annoying-beginner way*** is to grab the edge of the **Name** pane and to drag it just a little to the right, and drag it a bit more, and a bit more, and a bit more ... etc ... until all the label names are finally visible. Very annoying!!

The ***easy-expert way*** is to grab the edge of the **Name** pane and to drag it almost over to the right side of the waveform window, exposing the full label names, then drag it back to the trailing edge of the label names. Two moves and you are ready to debug!! *Please don't be annoying!*

In Figure 8 the label names have now been exposed and we are ready to view the simulation waveforms and assertions to help debug the RTL design.

Guideline #7: Use label names that start with "**ERR**" or "**ERROR**" and then include a short sentence to describe what is wrong if that assertion is failing.

This label technique helps describe a potential problem, making it easier to debug in a waveform display, plus this same label will be reported in the simulation transcript window when the assertion fails. This label is basically an active design comment and removes the need to add so many comments to the RTL code.


DVE - TopLevel.2 - [Wave.1] /home/cliffc/.../MSTR/Lab_Assert2/vcdplus.vpd
File Edit View Simulator Signal Scope Trace Window Help
Any Edge
C1:0
REF
Wave.1
N/A
N/A

Name	Value
Group2	
ERR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0_RPTR0_WPTR...	NA
ERR_FIFO_SHOULD_BE_FULL	NA
ERR_FIFO_SHOULD_NOT_BE_FULL	NA
ERR_FIFO_DID_NOT_GO_FULL	NA
New Group	

Figure 8 - Assertion labels displayed in the waveform display (after expanding the Name pane)

The first label name visible in Figure 8 is:

```
ERR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0_RPTR0_WPTR0_CNT0:
```

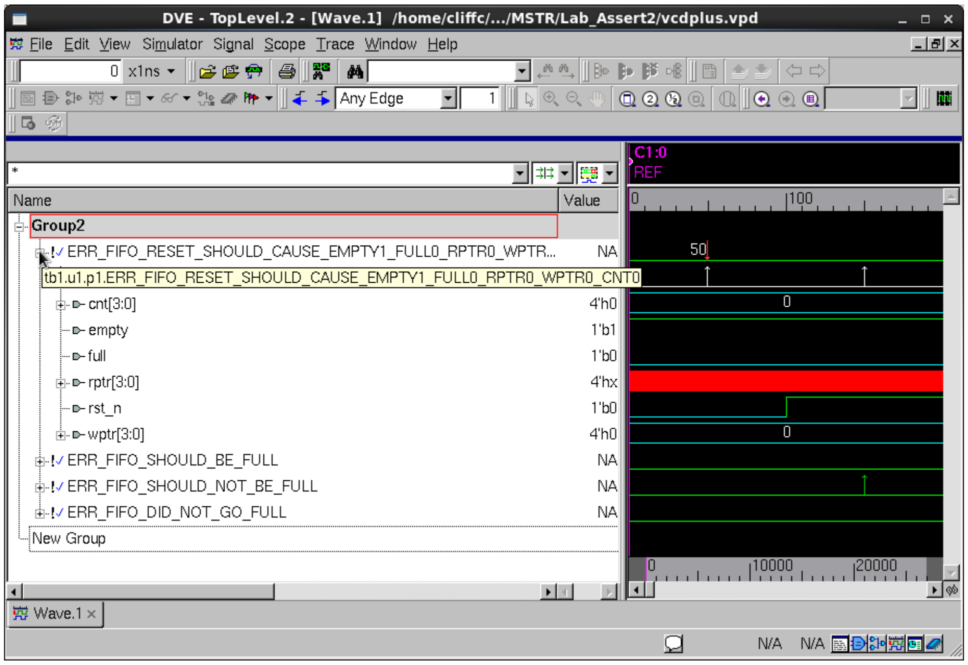
And notice that there is an assertion failure at time 50 (red down-pointing arrow).

The next step in the waveform display is to expand the failing assertion by selecting the [+]! to the left of the failing assertion. In DVE, selecting the [+]! symbol will expand the assertion so that all of the signals that were used in that assertion will be visible under the assertion name. This is extremely useful. This means the engineer does not have to hunt for the signals used in this assertion and drag them down to the assertion label name before debugging. A descriptive label name and access to all of the assertion signals means that debugging will be much easier to do with these assertions.

In Figure 9 the failing assertion has been expanded. Based on the assertion label name, we would expect reset to cause the listed signals to be assigned the following values:

```
rst_n = asserted
empty = 1
full  = 0
rptr  = 0
wptr  = 0
cnt   = 0
```

But from the waveform display in Figure 9, we see that the **rptr** is unknown (**X**). A quick examination of the **fifo1** code will show that **rptr** was not included in the reset operation and is therefore uninitialized.


DVE - TopLevel.2 - [Wave.1] /home/cliffc/.../MSTR/Lab_Assert2/vcdplus.vpd
File Edit View Simulator Signal Scope Trace Window Help
0 x1ns
Any Edge
1
C1:0
REF
0
100
50
0
0
0
10000
20000
Wave.1
N/A N/A

Name	Value
Group2	
ERR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0_RPTR0_WPTR...	NA
tb1.u1.p1.ERR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0_RPTR0_WPTR0_CNT0	
cnt[3:0]	4'h0
empty	1'b1
full	1'b0
rptr[3:0]	4'hx
rst_n	1'b0
wptr[3:0]	4'h0
ERR_FIFO_SHOULD_BE_FULL	NA
ERR_FIFO_SHOULD_NOT_BE_FULL	NA
ERR_FIFO_DID_NOT_GO_FULL	NA
New Group	

Figure 9 - Assertion label expanded to show all signals tested by the assertion

Long descriptive labels in the waveform display will certainly accelerate design debug using assertions. See Appendix 2 for editor macros that automatically change sentences to labels.

7. Concurrent assertions -vs- immediate assertions

Guideline #8: Use concurrent assertions and avoid using immediate assertions.

All of the examples in the paper up to this point use concurrent assertions.

Concurrent assertions sample all of the important signals only once per cycle at the end of the cycle, just before the signals change on the next clock edge, much like actual hardware registers sample the signals after they have settled at the end of the previous cycle. As with real hardware registers, we typically do not care about combinational settling or glitching during the cycle, we just care what the final values are at the end of the cycle.

Immediate assertions sample signals on demand when called. Immediate assertions can be quite useful to test asynchronous control signals such as reset, preset and latch enables. If any asynchronous control signals glitch they can indeed cause immediate changes that might require immediate evaluation.

This is the reason for the **Guideline #8** at the beginning of this section. ~95%+ of signals can be sampled at the end of a cycle and properly evaluated while perhaps only ~5% of the asynchronous control signals need to be sampled and evaluated immediately. This best describes my own usage ratio of concurrent and immediate assertions.

Remember that immediate assertions use the keyword **`assert`** and are placed in procedural code and executed as a procedural statement. They do not wait until the end of a sample cycle like concurrent assertions.

Consider the assertion example shown with potential combinational logic-settling race condition shown in Example 10.

```
logic S1, S2, S3, S4; // 1-bit variables to represent different states

always_comb begin // triggers on changes to each state bit
  assert ($onehot({S1,S2,S3,S4}) else $error("state bits not one-hot");
  case (1’b1)
    S1: ... // do stuff for state 1
    S2: ... // do stuff for state 2
    S3: ... // do stuff for state 3
    S4: ... // do stuff for state 4
  endcase
end
```

Example 10 - Immediate assertion - assert $onehot(...) FSM example with potential race condition

In Example 10, there is a potential race condition if the RTL code triggers multiple times allowing multiple onehot bits to be set momentarily before settling to the correct onehot **`next`** state value.

A potential solution to the problem is to use the **`final`** keyword added to the **`assert`** statement as shown in Figure 10.

```
assert final ($onehot({S1,S2,S3,S4}) else $error("state bits not one-hot");
```

Figure 10 - Immediate assertion - assert final $onehot(...) FSM example to correct race condition

Although this approach is valid and reasonable, the use of the **`final`** keyword still might report a false-error if unit delays are added to the RTL code during simulation. Running simulations with unit delays on a design with immediate assertions should not be done.

For designs like the one shown in Example 10, it is generally best to avoid any combinational settling issues and any potential race hazards by testing the same condition using a concurrent assertion from a ***bindfile*** as shown in Figure 11.

```
ERR_state_is_not_onehot: `assert_clk($onehot({S1, S2, S3, S4}));
```

Figure 11 - Concurrent assertion - `assert_clk($onehot(...)) FSM example

Another common and useful example is an immediate assertion used to trap illegal reset conditions as shown in Example 11.

```
// Assume the rst_n control line is never an X or Z
always_ff @(posedge clock or negedge rst_n) begin
  assert (!$isunknown(rst_n)) else $error("unknown value on rst_n");
  if (!rst_n) q <= 0;
  else        q <= d;
end
```

Example 11 - Flip-flop with asynchronous reset and inline immediate assertion

This style does indeed work under most conditions, but if there are hundreds of these registers in a design, any reset signal that goes unknown will execute hundreds of these error messages. Sending hundreds of error messages to the terminal can both slow down simulation performance and introduce a huge annoyance when hundreds of error messages are basically reporting the same problem.

One other problem with this immediate assertion example is that although it catches **`rst_n`** transitions from **`1-to-X/Z`**, it will not catch reset transitions from **`0-to-X/Z-to-1`** (rising edge on **`rst_n`**) that occur all within the same cycle.

A ***bindfile*** cannot insert immediate assertions into the middle of a procedural block, but a ***bindfile*** can include immediate assertions to check asynchronous control signals when necessary.

Instead of adding the immediate **`assert`** to do reset checking in every RTL register, put an immediate **`assert`** into an **`always`** process in a ***bindfile*** as shown in Example 12.

```
module DUT_asserts (
  // input declarations, including reset
);

  always @* begin
    assert(!$isunknown(rst_n))
      else begin
        $error("ERR_reset_went_unknown");
        repeat(2) @(posedge clk);
        $finish;
      end
  end
...
endmodule
```

Example 12 - DUT_asserts module with immediate assertion to check reset and $error reporting

Adding the immediate **`assert`** command to the ***bindfile*** offers three visible advantages over putting the immediate **`assert`** command into the DUT model itself:

- The **`assert`** can be checked once and issue just one error message.
- The **`always @*`** trigger will also catch the **`0-to-X/Z-to-1`** transitions missed by the immediate assert of Example 11.
- The **`always`** process can be setup to trigger an immediate **`assert`** error message and then wait for 2 more clock cycles before aborting the simulation with the **`$finish`** command, as shown in Example 12.

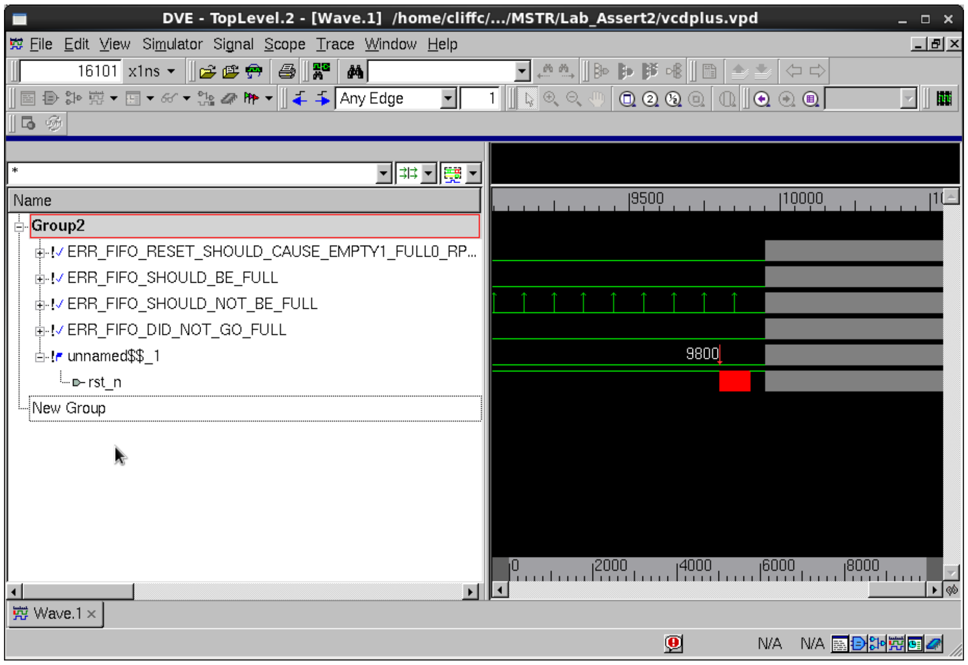

DVE - TopLevel.2 - [Wave.1] /home/cliffc/.../MSTR/Lab_Assert2/vcdplus.vpd
File Edit View Simulator Signal Scope Trace Window Help
16101 x1ns
Any Edge
Name
Group2
ERR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0_RP...
ERR_FIFO_SHOULD_BE_FULL
ERR_FIFO_SHOULD_NOT_BE_FULL
ERR_FIFO_DID_NOT_GO_FULL
unnamed$$_1
rst_n
New Group
9800
Wave.1
N/A N/A

Figure 12 - Immediate assertion DVE display without label name (Note: unnamed$$_1)

Allowing the **`assert`** to issue a message and then wait for two clock cycles typically makes the waveform display easier to read. Many waveform viewers might have a display-update race that would terminate the simulation before capturing the final signals that caused the assertion to fail. The waveform display in Figure 12 shows the failing assertion just before the end of the simulation.

One problem with the immediate **`assert`** command in Example 12 is that it has a **`$error`** message but no assertion label. In the absence of a label, DVE has assigned a default identifier of **`unnamed$$_1`** to the immediate assertion in the waveform window. If there were dozens of these immediate **`assert`** commands in use, the waveform window would be difficult to use.

Just as we did with concurrent assertions, the immediate **`assert`** command can remove the **`$error`** message and prepend the **`assert`** keyword with an assertion label as shown in Example 13.

```
module DUT_asserts (
  // input declarations, including reset
);

  always @* begin
    ERR_reset_went_unknown:
    assert(!$isunknown(rst_n))
      else begin
        repeat(2) @(posedge clk);
        $finish;
      end
  end
...
endmodule
```

Example 13 - DUT_asserts module with immediate assertion to check reset and default ERR_label reporting

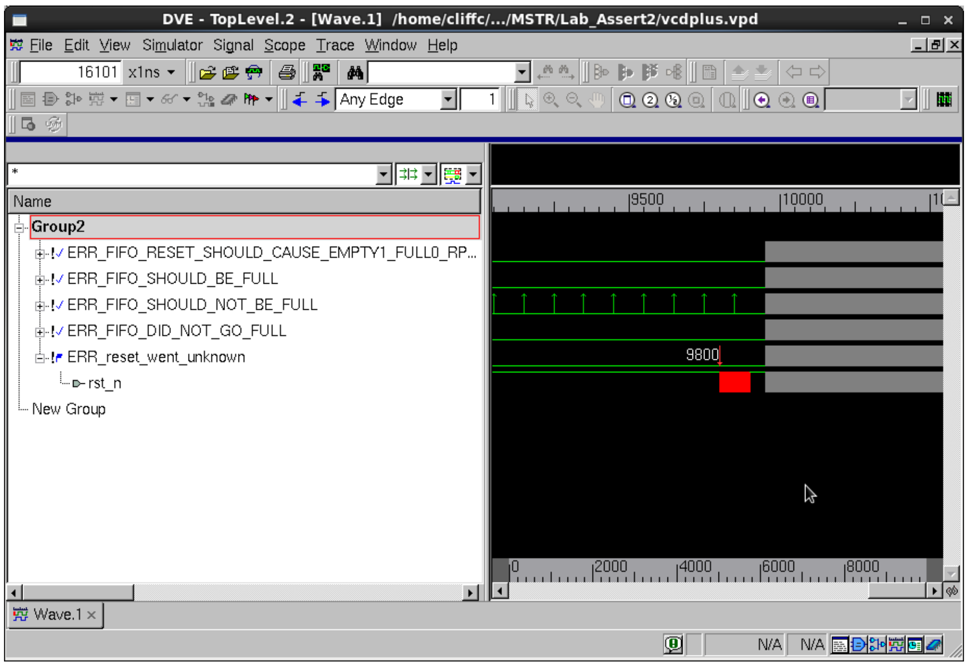

DVE - TopLevel.2 - [Wave.1] /home/cliffc/.../MSTR/Lab_Assert2/vcdplus.vpd
File Edit View Simulator Signal Scope Trace Window Help
16101 x1ns
Any Edge
Name
Group2
ERR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0_RP...
ERR_FIFO_SHOULD_BE_FULL
ERR_FIFO_SHOULD_NOT_BE_FULL
ERR_FIFO_DID_NOT_GO_FULL
ERR_reset_went_unknown
rst_n
New Group
9500
10000
9800
0
2000
4000
6000
8000
Wave.1
N/A

Figure 13 - Immediate assertion DVE display with label name (ERR_reset_went_unknown)

Adding the descriptive label to the immediate **`assert`** command in Example 13 causes DVE to display the label name in the waveform window as shown in Figure 13, just like it does with concurrent assertions. The immediate **`assert`** with descriptive label in the waveform window again simplifies the debugging task. The fact that the code of Example 13 was added to a ***bindfile*** and waited two clock cycles after the immediate **`assert`** failed, also allowed the waveform display to shown some useful information before aborting the simulation. This could simplify the debugging task.

To summarize this section, immediate assertions are very useful to test asynchronous control signals such as asynchronous reset, asynchronous set, and latch enables. If asynchronous control signals glitch in the middle of a clock cycle, they can cause unexpected behavior in the RTL simulation and detecting those glitches is most easily accomplished using immediate assertions.

When should engineers use concurrent assertions? Concurrent assertions should be used to test cycle based synchronous activity including combinational logic that may be harmlessly glitching and settling between clock cycles. In general engineers do not care if combinational logic is glitching as long as it settles to a valid and stable value in time to meet the setup time of the next register inputs.

Immediate assertions are sometimes used to test the success of **`$cast`** and the class-based **`randomize()`** method. Very large companies have shared with Harry Foster and I that they no longer use immediate assertions for these activities. I believe that Harry and I generally agree with these large companies; hence, the following additional immediate assertion guideline:

Guideline: Do not use immediate assertions to do **`assert($cast ...)`** or **`assert(class_variable1.randomize())`** success checking. These assertions are too easily disabled from the command line or from unseen code that disables all assertions with **`$assert_off`**. To test **`$cast`** and **`$randomize`** statements, it is better to use **`if`**-tests as shown in Figure 14.

```
  if (!$cast ...) $error("Cast operation failed ...");
  if (!class_variable.randomize())$error("Randomization failed ...");
```

Figure 14 - If-statement "assertion" testing of $cast and randomize() functions

8. Simple assertion macros

Guideline #9: Use macros to reduce SVA coding efforts.

For many years I have documented what I call:

> ***The SVA "circle of life"*** (also known as ***The disillusioned Design Engineer cycle!!***).

The SVA ***"circle of life"*** consists of the following steps:

- Engineers get excited about SVA capabilities
- Engineers take SVA training
- Engineers start to use SVA
- *Engineers find SVA to be too verbose*
- Engineers abandon SVA !!

Figure 15 - The SVA "circle of life"

Over the years, I have noted the following issues that have contributed to the abandoning of SVA usage by designers.

- There is a lot of SVA syntax to learn and remember.
- SVA syntax can be very verbose. The syntax often requires three or more (many more) lines of code to test one simple design feature (X lines of code to test one simple feature).
- SVA coding styles are very susceptible to syntax errors and typos (engineers complain that they are "spending too much time debugging assertions!")
- Assertions are sometimes hard to debug.
- Some frontend tools do not support some assertion keywords and abort.

Figure 16 - Reasons engineers abandon SVA

If you want design engineers to use assertions, I claim the assertion coding style has to be simple and it has to be concise. In 2009 I authored a paper that showed concise macros that could be used to reduce assertion coding errors significantly and reduce assertion coding verbosity by 40%-80% over conventional assertion coding styles [3]. Some colleagues have mentioned that they do not use the macros because they are not powerful enough, and my colleagues are correct, but I have found that the concise and simple nature of the macros is very attractive to most design engineers. If the macros are not powerful enough to write the assertions that you want it may be that you are writing assertions that are typically too complex for many designers.

When I show these macros to designers who have been trained, used and abandoned SVA, the overwhelming majority re-embrace SVA adoption by using the simple macros.

In recent years I have slightly modified the macro names and content from the macros that I showed in the 2009 paper. Section 8.1 documents the macro definitions that I now use.

8.1 Sunburst Design SVA macro definitions

I keep the following macro definitions in a file called **`assert_macros.sv`**.

```
`ifndef ASSERT_MACROS
`define ASSERT_MACROS

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

`define assert_clk( arg ) \
    assert property (@(posedge clk) arg);

`endif
```

Example 14 - assert_macros.sv file

The first two lines of code in this file are a common trick used by software engineers to ensure that only one copy of the assertion macros file is read.

The first macro, **`assert_clk_xrst( arg )`**, has all of the necessary code to properly add a concurrent assertion to a ***bindfile***. The " **`_xrst`** " portion of the macro indicates that assertion is checked on each clock edge, "*except when reset is asserted*" (**`disable iff (!rst_n)`**).

The second macro, **`assert_clk( arg )`**, has all of the necessary code to properly add a concurrent assertion to a ***bindfile*** and is not disabled for any condition, not even reset.

Not only do these macros reduce assertion coding effort, a major complaint by most design engineers, but they also reduce assertion syntax errors.

One of the most common syntax errors encountered by most new SVA users is the correct number and balancing of parentheses.

Consider the following assertion example without using a macro:

```
ERR_FIFO1_SHOULD_NOT_BE_EMPTY:
  assert property (@(posedge clk) disable iff (!rst_n) cnt>0 |-> !mt)
```

Example 15 - Properly coded concurrent assertion - verbose style

In this properly coded assertion example, there are three open parens **`(((`** and three closed parens **`)))`**. Coding the same example using the first macro shown in Example 14 would be done as follows:

```
ERR_FIFO1_SHOULD_NOT_BE_EMPTY:
  `assert_clk_xrst(cnt>0 |-> !mt);
```

Example 16 - Properly coded concurrent assertion - macro style

Since the overhead-code has been captured in the macro, the macro has reduced the required number of parentheses in this example to just one set.

9. Implications |-> |=>

Guideline #10: Use `|-> ##1` implications and not `|=>` implications.

`|->` tests for a valid consequent expression in the same cycle.

`|=>` tests for a valid consequent expression in the next cycle.

Implication operations can be described in English as, "***IF*** a qualifying (antecedent) condition is met, the implication is that the resultant (consequent) condition must also be true in the same cycle (`|->`), or in the next cycle (`|=>`), or in n-cycles (`|-> ##n`)."

These similar but slightly different implication operators are often confused or switched by new users and their similar syntax often makes them difficult to detect.

At multiple DAC conferences many years ago (~2004 and ~2005), my colleague, Alan Hunter of ARM shared that ARM engineers were not allowed to use `|=>` instead they were required to use `|-> ##1`

Another respected colleague, Harry Foster, explained to me the history of how `|=>` was added to PSL and subsequently to SVA. Harry explained that there was one unidentified member of the PSL committee who complained that he frequently was checking consequent expressions in the next cycle and wanted a dedicated shorter token to do that type of testing; hence, the `|=>` operator was born. This type of operator is often called "syntactic sugar."

> *In computer science,* ***syntactic sugar*** *is* ***syntax*** *within a programming language that is designed to make things easier to read or to express. It makes the language "sweeter" for human use: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer.[10]*

Basically, `|->` is a superset of the `|=>` operator. The `|=>` operator is completely unnecessary, but was added for engineers who wanted to avoid the slightly more verbose coding style: `|-> ##1`

Looking at the implications with large fonts makes it reasonably easy to see the different `|=>` and `|->` implication operators as shown in Figure 17.

```
ERR_b_did_not_follow_a: `assert_clk (a |=> b);
ERR_fifo1_not_full: `assert_clk ((cnt>15) |-> full);
```

Figure 17 - Viewing implication operators with large fonts

Young engineers with mega-perfect eyesight frequently use smaller fonts and it is somewhat harder to see the difference between the `|=>` and `|->` operators as shown in Figure 18.

```
ERR_b_did_not_follow_a: `assert_clk (a |=> b);
ERR_fifo1_not_full: `assert_clk ((cnt>15) |-> full);
```

Figure 18 - Viewing implication operators with small fonts

When I stand behind an engineer who uses a smaller font, I see code that looks similar to what is shown in Figure 19! The tiny fonts make it difficult to see any implication operator usage errors.

```
ERR_b_did_not_follow_a: `assert_clk (a |=> b);
ERR_fifo1_not_full: `assert_clk ((cnt>15) |-> full);
```

Figure 19 - Viewing implication operators with small fonts from a distance

Before I encouraged engineers to only use the overlapping implication operator, my students and I would both frequently miss that they were using the wrong implication operator and the debugging task was taking longer than it should.

Engineers must be taught and understand both syntax styles because there are many examples using each style in conference papers and being used in industry, but if you want to make fewer assertion coding mistakes, restrict your own usage to just the overlapping implication operator (`|->`) and use cycle delays as required.

I was aware of this recommendation for years and have shown this recommendation frequently in training, but only in recent years have I made this as a strong recommendation in my training classes.

After adding this strong recommendation, engineers in my training classes suddenly made dramatically fewer assertion coding mistakes. I am now a strong proponent of completely avoiding the `|=>` implication operator.

10. Conclusions

This paper has described a number of useful guidelines to improve simple assertion usage by all engineers. A summary of those guidelines follows:

Guideline #1: Start learning and using SVA after 2-3 hours of lecture and 1-3 hours of labs.

Guideline #2: ***bindfiles*** - use them!

Guideline #3: Inline SVA code - avoid it!

Guideline #4: Use the `bind` command style that binds to all DUT modules, not the `bind` style that only binds to specified instances.

Guideline #5: Add descriptive labels to your assertion code.

Guideline #6: In general, do not use `$error(...)` or `$display(...)` messages in assertion action blocks.

Using long, descriptive labels (a) documents the assertions, and (b) accelerates debugging using waveform displays.

Guideline #7: Use label names that start with "`ERR`" or "`ERROR`" and then include a short sentence to describe what is wrong if that assertion is failing.

Guideline #8: Use concurrent assertions and avoid using immediate assertions.

Guideline #9: Use macros to reduce SVA coding efforts.

Guideline #10: Use `|-> ##1` implications and not `|=>` implications.

Following these guidelines will simplify the use of assertions, reduce assertion coding errors and accelerate RTL design debug using assertions.

11. Acknowledgements

I am grateful to my colleague Don Mills of Microchip for his review and suggested improvements to this paper. Don Mills and Chuck McClish of Microchip are also responsible for the EMACS macro code shown in Appendix 2.2.

12. Postlude

It should be noted that it took me a few years to adopt some of the best practices described in this paper, even after they were shared with me by respected colleagues. I thought I was already using best practices when I was shown these contradictory recommendations. As engineers, we are sometimes resistant to new and often foreign recommendations before we arrive at our own conclusion that the new practices will serve us better than those that we are currently using and that have seemingly served us well.

I fully expect that the reader will also need time to consider the recommendations in this paper.

I also know that today's best practices may be superseded by new and better practices in the future. It would be a mistake for me to assume that I now have all of the final-best practices. As engineers it would be wise to keep an open mind to new recommendations that might come to us in the future. Please feel free to email me with your own best SVA practices.

13. References

[1] Harry Foster, "Maturing a Project's ABV Process Capabilities." Available at https://verificationacademy.com/sessions/maturing-abv-process-capabilities

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

[3] Clifford E. Cummings, "SystemVerilog Assertions - Design Tricks & SVA Bindfiles," SNUG 2009 (San Jose). Available at www.sunburst-design.com/papers

[4] Clifford E. Cummings, "SystemVerilog Implicit Port Enhancements Accelerate System Design & Verification," SNUG 2007 (Boston). Available at www.sunburst-design.com/papers

[5] Don Mills, "Being Assertive With Your X (SystemVerilog Assertions for Dummies)," SNUG 2004. Available at w ww.lcdm-eng.com/papers/snug04_assertiveX.pdf

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

[7] "IEEE Standard For SystemVerilog - Unified Hardware Design, Specification and Verification Language," IEEE, New York, NY, IEEE Std 1800-2009

[8] "IEEE Standard For SystemVerilog - Unified Hardware Design, Specification and Verification Language," IEEE, New York, NY, IEEE Std 1800™-2012

[9] Stuart Sutherland, "Who Put Assertions In My RTL Code? And Why? How RTL Design Engineers Can Benefit from the Use of SystemVerilog Assertions," SNUG 2015 (Silicon Valley). Available at www.sutherland-hdl.com/papers/2015-SNUG-SV_SVA-for-RTL-Designers_paper.pdf

[10] Syntactic sugar definition. https://en.wikipedia.org/wiki/Syntactic_sugar

14. Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 34 years of ASIC, FPGA and system design experience and 24 years of SystemVerilog, synthesis and methodology training experience.

Mr Cummings has presented more than 100 SystemVerilog seminars and training classes in the past 13 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

Last Updated: April 2016

Appendix 1 **Tools and OS versions**

The examples in this paper were run using the following Linux and Synopsys tool versions:

64-bit Linux laptop: CentOS release 6.5

VCS version K-2015.09-SP1_Full64

Running vcs and dve each required the command line switch -full64

Without the -full64 command line switch, vcs compilation would fail with the message:

```
g++: /home/vcs/linux/lib/ctype-stubs_32.a: No such file or directory
make: *** [product_timestamp] Error 1
Make exited with status 2
```

Appendix 2 **Editor key definitions to ease adding assertion labels**

Labeling is an important tip that I shared in this Assertions paper[3] and in Section 6 of this paper. In vim, I have a key definition that auto-inserts "_" characters between text and ensures that the last character on the label is a "`:`"

The key definition allows me to enter the type of error that is happening if the assertion is failing without the need to keep hitting <shift><_> in place of all of the blanks. It is a convenience key that allows me to enter assertion labels quickly and naturally.

Using VIM, I create labels by typing:

```
ERR definition of what is going wrong if the assertion fails
```

Figure 20 - Raw label entry using vim

I then exit the vim insertion mode and then press the <shift><_> keys, which converts this text into the following valid label:

```
 ERR_definition_of_what_is_going_wrong_if_the_assertion_fails:
```

Figure 21 - Label properly formatted using vim key definition

I have assigned the vim key definition to the "_" key, so after I add my text and exit the insert mode of vim, I can hit the <shift><_> keys once and it will insert the proper "_" characters and label termination. I have placed this key definition in the `.vimrc` file in my users home directory.

For example, if I have any of the following text examples:

```
ERR FULL write caused wptr to change
  ERR FULL write    caused 1 2 3 wptr to change  ::  : :
```

Figure 22 - Examples of raw label-text entry using vim

After hitting the _ key on each line, the text is properly modified as shown below:

```
ERR_FULL_write_caused_wptr_to_change:
  ERR_FULL_write_caused_1_2_3_wptr_to_change:
```

Figure 23 - Example labels properly formatted using vim key definition

For an equivalent EMACS key definition, see Appendix 2.2.

Appendix 2.1 **The VIM key assertion label definition**

The following is an explanation of my vim key definition.

VIM key definition (and explanation):

```
map _ ma:%s//                       /ge><cr>'a:s/\(\w\)\s\+\(\w\)/\1_\2/ge<cr>:s//\1_\2/ge<cr>:s[_:
]*$/:/<cr>:/:$/<cr>'az.
```

Explanation:

`map _`	`map` the "_" key
`ma`	Mark the current line as "`a`"
`:%s/ <tab character> / <8 spaces> /ge><cr>`	Whole file, replace all `<tab>` characters with 8 spaces (`ge` - globally and ignore not-found errors)
`'a`	Go back to line marked "`a`"
`:s/\(\w\)\s\+\(\w\)/\1_\2/ge<cr>`	This line only, find and save any word character `\(\w\)`, followed by one or more white space characters `\s\+`, and again find and save any word character `\(\w\)`, replace it with the 1st word character saved `\1`, followed by "_", followed by 2nd word character saved `\2`, and replace it globally on this line (`ge<cr>`).
`:s//\1_\2/ge<cr>`	Problem: the above command does not properly substitute single characters ("`full 1 empty`" becomes "`full_1 empty`") because the single character cannot both end the last found pattern and start the next found pattern, so repeat the previous substitution (`//` finds the last searched pattern again).
`:s[_: ]*$/:/<cr>`	This line only, find any pattern of "_" `:` and blanks at the end of a line and replace them with a single `:` (label termination).
`:/:$/<cr>`	Find "`:`" at the end of a line (this will highlight all label terminations in the file, useful for viewing labels).
`'a`	Go back to line marked "`a`"
`z.`	Center this line on the screen

Appendix 2.2 **EMACS auto-assertion label creation**

My colleagues Don Mills and Chuck McClish of Microchip have graciously supplied a similar definition that can be used in EMACS to insert the "_" characters just like the vim key description shown in Appendix 2.1. The code below is mapped to "C-x _" but the EMACS user can map the macro to any key they want.

```
(fset 'auto-underscore-insert
   [?\C-x ?r ?  ?q ?\C-a ?\C-  ?\C-e ?\M-x ?n ?a ?r ?r ?o ?w ?- ?t ?o ?- ?r
?e ?g ?i ?o ?n return ?\C-  ?\C-\M-r ?\\ ?w ?\C-m ?\C-f ?\C-w ?: ?\C-a ?\C-
\M-s ?\\ ?w ?\C-m ?\C-b ?\C-  ?\C-e ?\M-x ?r ?e ?p ?l ?a ?c ?e ?- ?r ?e ?g ?e
?x ?p return ?  return ?_ return ?\C-a ?\M-x ?w ?i ?d ?e ?n return ?\C-x ?r
?j ?q])

(global-set-key (kbd "C-x _") 'auto-underscore-insert)
```

Appendix 3 **Bind to specific DUT instance**

Recommendation: Do not use this style.

This style specifies both the target dutfile to bind to, as well as which instance of the target dutfile is being bound to. In theory, it is possible to list multiple **`bind`** commands of this style to different instances of the target dutfile.

```
  bind fifo1 :u1 fifo1_asserts p1 (.*);
```

Example 17 - Non-recommended bind style - binds to just one instance of a DUT

Example 17 shows how to bind to the **`u1`** instance of the **`fifo1`** design. This style requires that the **`bind`** command be scoped to the same module as the **`u1`** instance of the **`fifo1`** since the **`fifo1`** could be placed in multiple levels of hierarchy and in each scoped hierarchy the instance name could also be **`u1`**.

The **`:u1`** instance name could be a hierarchical path such as **`:w1.u1`** and could be a list of instances in the design with different hierarchical listings.

NOTE: Until recently, older versions of VCS did not support this **`bind`**-style but as of the VCS 2015.06 version, this style is now fully supported. I am not sure which VCS version started to support binding to a single instance. VCS' previous lack of support for this style was not an issue for me because in my opinion, it was not wise to limit assertion checking to just one copy of a **`module`**. A designer who uses this style or the style shown in Appendix 4 might bind to a working instance while another instance in the design could be failing.

Appendix 4 **Bind to an instance name without specifying the DUT module name**

Recommendation: Do not use this style.

This style only specifies the instance name of the target dutfile that is being bound to without specifying the module type, while the style in Appendix 3 first referenced the module name and then included the instance name.

As with the previous style, it is possible to list multiple **bind** commands of this style to different instances of different DUTs.

```
bind  :u1 fifo1_asserts p1 (.*);
```

Example 18 - Non-recommended bind style - binds to an instance name only

Example 18 shows how to **bind** to the **u1** instance in the current scope. This style also requires that the **bind** command be scoped to the same module as the **u1** instance of the DUT.

Once again, the **:u1** instance name could be a hierarchical path such as **:w1.u1** and could be a list of instances in the design with different hierarchical listings.

NOTE: even though older versions of VCS did not support the ability to bind to one instance of a module until recently, VCS has long supported this style to bind to an instance name.