Novas Core Demo Guidelines

Introduction

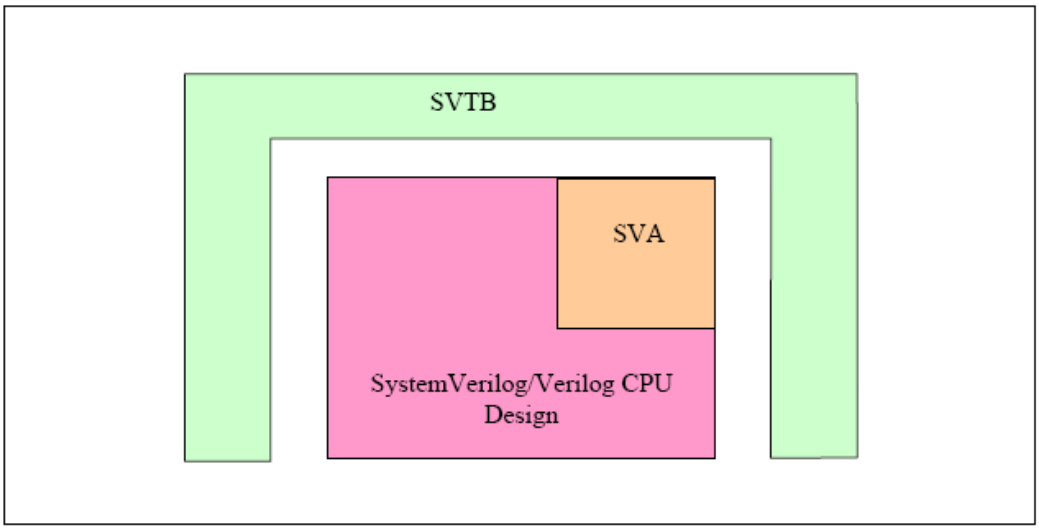
This document describes the Novas core demo and a suggested demo flow and guidelines. The goal of the core demo is to serve as an overall Novas demo as well as a detailed demo on SystemVerilog Assertions (SVA), SystemVerilog Design, and SystemVerilog Testbench (SVTB).

Note that these are guidelines only and instead of following the steps outlined in a very mechanical fashion, it is suggested that you follow your own natural style based on the features and examples on this document.

Demo Case Description

The design under test is coded using Verilog/ SystemVerilog /SystemVerilog Assertions (SVA), and SystemVerilog Testbench (SVTB). The design is based on the "standard" Novas CPU demo case. Some modules are converted into SystemVerilog while the remaining ones are in Verilog.

Several SystemVerilog assertions have been added to the design as well. The testbench is SVTB which includes constraint random generation.



Demo Highlights

Major capabilities shown in the demo:

- Assertion Management (Property Management)
- Assertion Evaluator (FSDB Checker) for SVA
- Assertion results visualization
- Assertion Analyzer (Source and Waveform)
- Automatic debug using Verdi Temporal Flow View (One button solution)
- Macro Debug
- Function Debug
- SVTB Logging
- SVTB Testbench Browser
- Declaration Tree
- Class viewer
- SVTB Interactive simulation

Demo Flow and Guidelines

Setup

The setup should already be done by a setup file. In any case, for reference, here is the setup:

setenv VERDI_SVTB_BETA 1

setenv VERDI_SVTB_ALPHA 1

Invoke Verdi for the demo

```
% cd verdi/sim
% RUN
```

Understanding your Design

The rate of adoption of SystemVerilog by previous Verilog users and even some VHDL users has been increasing dramatically and the trend is expected to continue. In this design, we have Verilog/SystemVerilog for the design part, assertions coded using SystemVerilog Assertion, and testbench coded using SystemVerilog Testbench constructs with VMM library. SystemVerilog covers the broad spectrum of design, testbench, and assertions and while this makes it very powerful, it also presents challenges as far as debugging is concerned. With all the different aspects of the design and verification environment under one language, it is even more critical to be able to easily understand your design. Verdi's leading-edge comprehension capabilities have been enhanced to overcome the challenges of understanding not only your design but also testbench and assertions.

First, let's take a quick look at this design.

This design includes Verilog, SystemVerilog, SystemVerilog Assertion and SystemVerilog Testbench. Let's look at the design side first.

1. Show the SV/Verilog modules and testbench <test>.
   - **In nTrace HB, double-click on <cpu_tb_top >**
   - **In nTrace HB, double-click on <system>**
   - **In nTrace HB, double-click on <i_cpu >**
2. Show a SystemVerilog testbench module – note the different icons for "module" vs. "program" blocks in HB.
   - **Double-click <test> to show SVTB – show the classes with C folders in HB.**
3. Show a SystemVerilog assertions in the module
   - **Expand <i_cpu> from design part to show SV design modules and SVA in the design.**
4. Show Verdi basic features in case customers aren't familiar with them.
5. To see the source code of any module.
   - **Double-click <i_ALUB> - show the source code**
   - **Double-click module <ALUB> in the source code - show where ALUB is being instantiated.**
   - **Double-click i_ALUB to jump back to module ALUB**
6. Show trace drivers and loads.
   - **Double-click i_ALUB in HB**
   - **Select signal "ALU" in the source code**
   - **Double-click "ALU" to trace drivers.**
   - **Show different drivers.**
7. Show loads
   - **Select "out" and click on "trace load" icon**
8. **NOTE** – this is a great way to show we have SV, SVA and SVTB syntax in this demo and how we can traverse through them seamlessly.
9. Show SVA code.
   - **In nTrace HB, double-click on INCPC**
10. It looks like **INCPC** is built upon several sequences.

Starting from an assertion failure

Let's start debugging.

Assertion-based Verification (ABV) is now a widely used methodology to monitor if the design's behavior meets the spec (intent). We have seen how you can spot assertions in the nTrace HB.

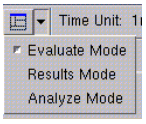
However, for a big design, there may be dozens or even hundreds of assertions in the design and they may be spread all over the design hierarchy. To manage assertions, visualize results, and debug assertions becomes a big issue in such an environment which is becoming increasingly common.

For assertion checking, you can check assertions during simulation. Verdi can import assertion results from simulator through the widely used FSDB format.

In this demo case, let's assume the user has just completed coding a few assertions and wants to do a quick checking on these assertions. Instead of checking these assertions during simulation, Verdi provides a unique Assertion Evaluation Engine which can check assertions against a signal level FSDB trace file for the design. This allows you to check your assertions in post simulation (data-mining) mode without re-running simulation every time when you want to verify the assertions.

Understanding SystemVerilog Assertions (SVA)

1. In Verdi, we provide a set of capabilities to help you to manage, visualize results, and debug your assertions. Invoke Property Tools window from nTrace
   - **nTrace →Tools→Property Tools….**
2. Explain the Property pop-up form. On the left hand side, you can easily see the assertions hierarchical tree view which shows where the assertions are with respect to the design hierarchy. You can also switch to table view mode to view all the assertions in your design.



   - **Select "Evaluate Mode" on Toolbar**
   - **Show Tree and Table views**
   - **DnD INCPC back to nTrace**
     - **i. To show the integration between Property Window and nTrace**
     - **ii. Turn on Active Annotation to show the NF of INCPC**
   - **(Optional) Show how user to debug on assertion on traditional source code**
     - **Click on e_INC to show the property**
     - **Click on e_r, it will jump to the sequence "sequence `M_SEQ"**
       - **For reuse purpose, user will write some commonly used property/sequence as macro**

- **If user trace into a macro, it's usually not easy to understand what the macro is**
- **Turn on Source→Expand Macro to show the source code of macro**
- **Expend macro by clicking on "+" sign next to line 128**
- **Not only source code but simulation result is annotated**
- **DC on ALU to trace driver**

- **Switch back to Property Tool Toggle the check mark –** you can use this to tell Verdi's assertion evaluation engine which assertions need to be checked. In this case, we are going to check all of them.

3. As we mentioned earlier, Verdi can import assertion results from simulator through the widely used FSDB format. If you add or modify an assertion or change your design, you would need to re-run whole simulation to check the assertion(s). Verdi includes an Assertion Evaluation Engine which can check assertions against a signal-level FSDB trace file. This allows you to check your assertion in post simulation (data-mining) mode.
4. In this case, we have a FSDB file that contains design signal value from the previous simulation run. We have just completed coding several assertions. Instead of re-running simulation to check these assertions, we can use Verdi's assertion evaluation engine to check them.

   - Click on **Evaluate** icon 
   - Click **Yes** on popup form
   - **Note** - You need to make sure that you have the write permission for this directory; otherwise it may cause some unexpected result, like crash… (the DAC booth machines will be set up such that the required demo account has write permissions)
5. While it is running, you can explain how the Evaluator works. For example, "As you may know, assertions check expressions or sequences of signals in your design. The evaluation engine in Verdi will grab the signals that impact the assertion from the FSDB file and calculate the expressions and sequences and hence the assertion results".

Assertion Statistics and Debug

When the Assertion Evaluator is done checking the assertions against the design FSDB file, show the results in the Property view.

1. In the Property view, you can choose one of three layout modes. This allows you to focus on the appropriate aspect of the Property view.
2. For example, we have completed the checking and would like to focus on the results. We can switch to the result mode.
   - Change to **Results Mode** from the layout selection icon in the right hand side
   - See how the Property view gets re-configured so you can focus on the results.

3. You can easily see the statistical summary of the assertion check here. Note that these same facilities and the ones we will be showing you shortly are also available if the simulator had done the assertion checking.
4. In this example, there are a total of 4 properties and there are 3 assertion fails during this run.
   - **Double-click on 3 on Fail column**, this adds the assertions to the Property Details section.
5. In the "Property Details" it shows you the details of each property which include the number of successes, failures, start/end time, etc. This can be configured by users via options.
6. To add more properties to the "Property Details", simply select the property and click "Add".
   - **Select "Cover" and click on the "Add" icon.**
7. To see each failure, click on the property.
   - **In the Property Details field, expand INCPC.**
8. As you all know, assertions are usually based on the spec and are designed to check that your design behavior matches the expected behavior. If there's any assertion failure, there must be something wrong with your design or the assertion code itself. Let's focus on the assertion fails.
9. Let's use the first failure, F16 as an example. Note that the failures are sorted by time.

Tracing SVA to Design

Verdi includes a powerful assertion analysis engine which automates the process of finding the root cause of an assertion failure. It does this by analyzing all the underlying properties, sequences, and expressions that make up an assertion.

1. Assume we are going to debug the F16 failure. While you can start debugging from the waveform, the most straight-forward way is to start from the assertion fail in the "Property Details" Window. Select F16 (assertion fail, begin time:6650, end time:6850) in the table.
   - Select **F16**
   - Click on "calculator" icon

2 This reconfigures the layout of the property pop-up form so you can focus on analyzing the failure using Verdi's Assertion Analyzer engine.

3 Verdi's Assertion Analyzer engine will analyze the assertion code along with the simulation results to determine the real path within the assertion code that causes the assertion failure. It also automatically adds the associated design signals to the waveform. We will show you the waveform later. Let's first analyze the assertions and figure out what causes the failure.

- **Double-click F16 in the Property Details window** – this would automatically analyze the failure in the Analyzer view and add the contributed signals in the waveform.

4. In the Assertion Analyzer view, you can see assertion INCPC comes from property e_INC. The sample time of property e_INC is at time 6850 – the property fails at that time and that causes the assertion failure. We can double click on the property to trace its cause. Note the sample (or evaluation) time of each property, sequence, and expression is annotated on its top. This is unlike active annotation which annotates values for one sample point only. Due to the temporal nature of assertions, you need to see the different sample points for each term.
5. Tile Assertion Analyzer and Waveform window (manually) together to show the automatic arrangement of waveform from assertion constructs
   - Cross reference the signal groups added by assertion analyzer and the assertion source code to help user understand the sequence/coding style of INCPC assertion
   - Also check the waveform of the signals. The assertion signal is fully dumped and property and sequence are partially dumped. This is because these signals are calculated on demand. This will ensure better performance and smaller database size.
6. Let's start to debug on Assertion analyzer to find out what's the real cause of the fail
   - **Scroll down the Analyzer window to show property e_INC and sequences e_r**.
   - The Assertion Analyzer automatically traces through the property **e_INC_b** since the sequence **e_r** is the only failure.
   - Let's look at the sequence **e_r** and continue debugging it. Sequence **e_r** is a temporal combinational statement made up of expressions. Note that since the code is temporal, Assertion Analyzer will also annotate the time information on the top.
7. In addition, for a cycle range operator (in the demo, ##[1 : 2]), the exact number of cycles that elapsed before the subsequent expression is evaluated is annotated on the top (#2 in the demo).
   - **Point to the line 10 in the Analyzer view - ##[1 : 2] …. which has #2 annotated on the top.**
8. The second part of the expression is a complex combinational expression. To figure out what is going on, you would have to manually write down the values of all the signals and variables and manually calculate the sub-expressions and expressions. Verdi's new Assertion Analysis engine analyzes the expressions and sub-expressions. Initially, you can see that the whole big expression after the delay is "FALSE". To find more detailed information for sub-expressions,
   - **Click on the Expand button.**
   - The engine splits the expression into sub-expression and annotates their state.

7. Now it's clear that the main suspect for the cause of the sequence failure is because **"ALU**" is not equal to **"ALU_prev+1"** which is 19.
8. Let's debug **ALU** which is a design signal.

Automatic Tracing of RTL

As we have seen so far, the assertion failure can be caused by HDL design bug – let us continue to debug the HDL design.

1. Let's debug ALU
   - **DC on ALU, Verdi will bring you to nTrace with the case statement**
2. (Optional)Take a closer look on the case statement. We know some user will write frequently used features into functions. In this case, you can notice there're some functions in the case statement. Because functions are 0 time operation, simulator will not dump out result from simulation. Verdi now can help user to calculate function value and also help user to trace into functions. (Make sure macro expansion is turned off)
   - Turn on Source→Function Annotation; you can see the functions are now annotated with value. Because the value is calculated by Verdi's engine, we use different color to differentiate with simulation result
   - DC on "sub1" to trace the function; Verdi switches to function trace mode. Left hand side of Verdi is now showing the function stack.
   - DC on "sub_min" to continue tracing into another function. You can understand how the value of the function is from
   - DC on first call stack [0]sbu1(c,0,0) to exit function tracing
3. (optional) Usually there're many signals in a scope, it's difficult for user to see all the signals in a list. Verdi now also provide "Signal List" allows use to list all the signals in current scope. Click on toolbar icon "Signal List" to open it
4. (optional)As you can see, signal not only shows the signals in current scope, for SystemVerilog struct signal we can also have hierarchical way to show it. Take "op" for example, it is a structure signal. Sometimes it user will need to understand the members of the structure signal. In this case you can see a[7:0], b[7:0] and cin are the members of op and the value of them can also be observed in the table.
5. (optional) You can also use RMB to do tracing for the selected signal.
6. The case statement is now active on `ADD. Let's debug on it. "op.a" seems to be the candidate which propagates the wrong value. To trace on op.a you can use Verdi's automatic tracing capability.
   - Select op.a @line258, click on Auto Trace button on toolbar
   - Verdi will automatically bring up TFV and trace to the source of the problem without any extra settings
   - TFV brings up to dataout[7:0], DC on it you can see the signal has come to the

boundary of the HDL design. The wrong value seems come from testbench.

Summary

We have just gone thought the Verdi's capabilities for SystemVerilog Design and SystemVerilog Assertions. Verdi provides full set of Assertion features, including Assertion Manager, Assertion Evaluator and Assertion Analyzer. With these capabilities, you can manage your assertions, evaluate assertion results independent of simulator, and also trace assertion behavior backwards through time. With Verdi, you can debug assertions easily. Also for HDL part, besides of Verdi's traditional strength for debugging automation, Verdi provides more handy features such as macro debug, function debug and one button solution for automatic tracing.

SVTB Part

Understand SVTB Design

1. Let's take a closer look on the SVTB part of the design
   - Bring up Testbench Brower (TBBR) from nTrace, Tools→Testbench Browser
   - Because testbench is software oriented, in this window will provide declaration based hierarchy tree to help user understand the structure of their testbench
2. Expand $root node
   - You can see there're several classes declared under global area of the testbench
   - DC on class "cpu_program", you can see there're functions and tasks. The inherited function/task will be marked with pink arrow to indicate it's declared from other class
   - On Inheritance View, you can realize the inheritance relationship from the selected class. The cpu_program is extended from vmm_xtractor class, so this program will inherit the function from it.
   - Click on the function "new", you can see the inheritance relation of functions
   - Click on "cpu_address_restrictions", this is a constraint in SVTB program. Switch to Constraint tab, it will show you the variables been constrained by the selected constraint.
   - Vise versa if you select variable from declaration tree we will show you the constraints which influence the selected variable. Select **start_write_addr** and switch back to Constraint tab, you can see this variable is actually constrained by 2 constraints.

SVTB simulation result visualization for logging

As we mentioned previously, testbench language is more like software, which means its hierarchy is not static. It changes over simulation time, and even data in testbench is dynamic. Dumping out simulation results for SVTB is very difficult. Today testbench designers use logging mechanism to print out necessary information for debugging. In fact, most of the commonly-used SVTB libraries, like VMM or OVM include built-in mechanisms for logging. However, generally, such logging provides text only information and the user needs to use a t ext-editor to see its content eyeball abnormalities manually. Then they have to manually correlate back to their design to see what's going on.

1. **vi demo_log.txt**
    - This is a typical text format log from SVTB simulation. The user usually inserts key word (like DEBUG) to monitor the status or problem from testbench in simulation. User then finds time and variable information from the resulting text log and goes back to check source code and HDL simulation result to see what's going wrong.
2. Verdi now provides a mechanism that allows users to dump out logging information directly into Verdi's FSDB database.
    - Bring up nWave, File→Restore Signal→svtb_demo_2009.rc. You can see the log messages are saved into Verdi's waveform format. It becomes easier to see the time information for the messages. (If you don't see overlapped messages, select all signals→Waveform→Message→Expand/Shrink Transaction Signals). From "do_execute_inst_loop", you can easily find out the status of CPU status from testbench.
    - There're several fields available inside of message signals. Select one of the messages from the **compare_map** stream and do, RMB→Properties…. The detailed information including all the fields with their value and call stack can be seen in this window. You can also change radix for the value here. Select the \pram_duv::main::data_compare attribute and change Radix to Hexadecimal. The value radix displayed in the waveform will also change.
    - Some fields in message boxes may be important for us and we need to monitor their value change along the simulation time. While there is rich information within the message box as attributes, it can be daunting to monitor details of a variable. Now, you can expand the important variables from a message similar to the way you expand bus waveform elements. Select the **compare_map** stream and do Waveform→Message→Create Attribute Signals. In the resulting pop-up form, enable (toggle) the box next to \pram_duv::main::data_compare and press OK. A new signal row will be created to represent the waveform for \pram_duv::main::data_compare. Because we cannot determine the exact value change of the "transitions" for \pram_duv::main::data_compare, we use green

triangle to indicate whenever a change is detected compared to the previously logged value rather than a typical value-change type transition vertical line.

- How do the messages get printed into FSDB? Verdi provides a new task "fsdbLog"; which allows user to print messages into waveform format. Select a"next instruction received" message in nWave and D&D to TBBR Source code where you can find the source code which produces the message.. The $fsdbLog command is actually very similar to $display, and wherever there's a $display, you can replace it with $fsdbLog. The resulting logging information will be printed into waveform. This will help to see the message clearly and also allows you to do analyze these messages further.
- Let's see how logging message can help you to debug testbench. Expand "CPU_data" group signals. (If you don't see overlapped messages, select all signals→Waveform→Message→Expand/Shrink Transaction Signals) Move cursor to 6800ns. (You can also DC on Start_tim, End_time marker and delete them to make screen clean). The signal "dataout" is actually output from testbench. We found there's an error transition 97→c. Actually, using a monitor scheme from testbench, the message "status" reports the comparison result from testbench. It did flag a fail status, which indicates there's something wrong.
- The message could contain more information than is practical to view in a waveform. Verdi's Message Analysis Window can help you to manage messages efficiently in a textual spreadsheet-like window. In nWave, do Tools→Message→Message Analysis Window. Click on Get Stream button and select "status", press OK. All the available information inside messages will be listed here similar to a spreadsheet. You can also configure the columns to focus on the information of interest. Do View→Column Configuration; keep Label, Msg, BeginTime and \pram_duv::main::data_compare in Show Column.
- In this case, if we only care about the "fail" messages, we can also filter out the rest of the messages. View→Filter, Column change to "Label" and Criteria "=" "fail"(Also turn on Sync to Waveform option to apply the setting to nWave) then press "Apply". The information in the table now only contains "fail" messages and the messages on the waveform also include only the "fail" parts. We can also sync the table with waveform, View→Sync Cursor Time and Sync Active Transition. Click on the table to show synchronization between table and waveform.

SVTB interactive simulation

One of the most commonly used debugging techniques for testbench is simulation interactive debugging. Verdi also provides seamlessly integration with simulator. Once you've setup the simulation environment for SVTB, you can also easily bring

up the interactive simulation into Verdi's environment. With Verdi's environment you will have better visualization and also easier control for the simulation.

1. Bring up interactive simulation on TBBR, Tools→Run Interactive Debug.
2. One of commonly asked question for interactive debug is: How do I decide where should I set a break point for debug? In this demo case, as we mentioned earlier, logging message is where you can find out the status inside of your SVTB simulation. We know there's a "fail" flag, we can use this as a start point for break point. We can also color the message printed for CPU instructions, select \$unit::\cpu_instr::print_instr, then Waveform→Message→Filter Colorize…. Press the Append… button, select instr_color.flt then press OK. You can see the message for CPU instructions are colored according to its label name. This helps us easily to distinguish what message is important.
   - Select fail (Select on fail text) message @6800ns, DnD from nWave to TBBR source code. DC on line 104 to set break point.
   - Because this is a CPU design, we might also need to see if there's anything wrong with the CPU instruction. Select message ADDA_ (@6450), DnD to TBBR. DC on line 214 to set another break point.
3. Click go! button to start simulation. You can see simulation stopped at first breakpoint at line 214, the center of the window shows the callstack of the stopped point.
   - Switch to Local tab, you can see all the local variables at current class where the breakpoint stops. You can also observe the current values of the variables
   - DC on #8 index of call stack, Verdi will help you to stack up in the call stack. DC on #4 index of call stack, you can stack up further. This will help you to understand the calling sequence to current break point
   - Click go! button again to continue simulation. This time, simulation stopped at line 208 again. Now we want to manually to pick some important variables to monitor the value, firstly check this case statement. DnD "mode" (@line197), "operand" (@line205) and "size" (@line 243) into variable watch window.
   - Continue to run simulation, you can find the value is changing according to simulation.
   - Now, if we determine that there is nothing wrong at this breakpoint, we can disable the break point on line 208 (DC on 214) and continue to run simulation. The simulation now stopped at line 104, the simulation time is now @3200ns. We're now at the first fail. You can see the callstack has changed. We know we're now at pram_duv class; this is where testbench produces memory data and drive to DUT. We can try to monitor memory value and the address. Scroll up to line31 and line 32, DnD "mem" and "addr" into variable watch window.

- Press “Go!” button, the simulation will stop @ 4700ns. Continue to run simulation. Press “go! Button again. Now it comes to 6800ns, this is the place where we started to debug. The address now is “151”; let’s check the memory value of address 151. You can clearly see the memory now is “12” which is “c” in hex. This is exactly the value we found from the design signal. The error value is indeed produced from testbench. The memory content is produced throughout testbench, we will need to further check if there’s any error constraint or other problem from testbench. Verdi provides all the needed capabilities for you to work seamlessly with simulator.

Summary

For SVTB, Verdi extends the capability from HDL to SVTB world. Building on Verdi’s straight-forward display and use-model, the user can now print SVTB messages into Verdi’s waveform database. User can not only see the message information from the waveform, but also the time and sequence information of the message. Plus Verdi’s message analysis capability allows you to easily filter out unnecessary information and hence focus on the relevant message for debugging. Messages are integrated into Verdi’s tightly integrated window infrastructure, thus allowing you to check where a message came from in the source code by simple drag-and-drop. Messages also serve as a good starting point for debugging your testbench. Interactive debugging is also one of commonly used debugging technique for actual testbench code. Verdi provides seamless integration with the simulator, allowing user to debug interactively through Verdi’s platform.