# Adding Last-Minute Assertions to a Design and Verification Project: Lessons Learned (a little late) about Designing for Verification

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

***Abstract*—The design team was finished, and confident the design was correct. Verification of the design was well underway and all tests were passing. As a consultant, my task was to add coverage assertions to help ensure the verification code was testing key aspects of the design. Was it effective to add assertions at the last minute? Were any bugs found? What did assertions reveal about the verification coverage? How difficult was it to add assertions so late in the design cycle? The answers to these questions just might change how you verify your next design.**

## I. Introduction

I was recently hired as a consultant to assist a verification team that was implementing a constrained random testbench using SystemVerilog[1] and VMM[2]. My portion of the work was to add coverage to the testbench in order to determine if the randomized tests were exercising critical portions of the design.

Several aspects of the design involved communication protocols that could span dozens of clock cycles. This seemed like a perfect fit for SystemVerilog assertions combined with SystemVerilog coverage — or so I thought.

## II. Overview of the Design

This section provides a high level description of the design to be verified. The actual design is proprietary, and is for a product in a highly competitive market. To protect the interests of company that hired me, this high-level description is purposefully vague, but contains enough detail in order to understand how and why assertions were used for verification coverage.

The design to be verified was a small System on Chip (SoC) design, comprising an analog section, a digital section, and a EEPROM. Figure 1 shows a high-level block diagram of the design. A brief description of the blocks where coverage was of primary concern follows the block diagram.

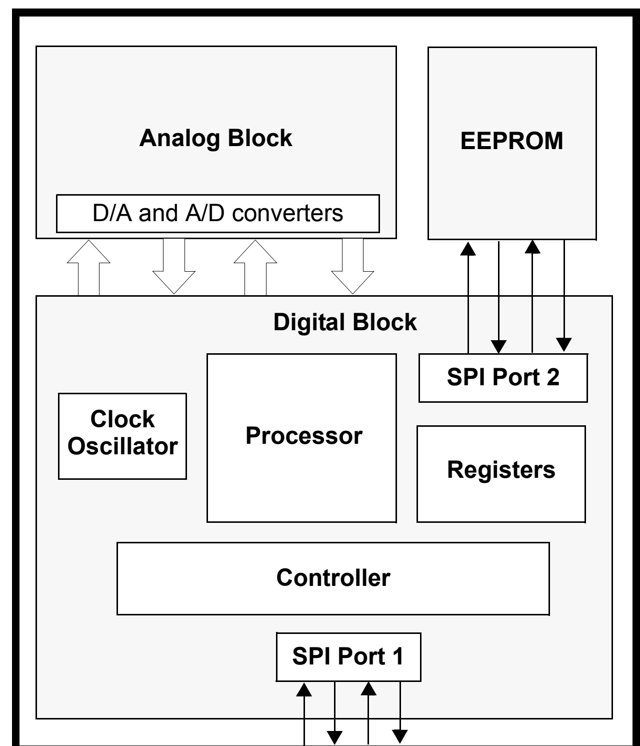


Figure 1. Block diagram of the design that was verified

### *A. The external interface*

The only communication to the outside world is through a proprietary Serial Peripheral Interface (SPI) port, which consists of 4 primary signals. The design is always the slave on this SPI port. Some external device is the master of the port; all communication is initiated and controlled by this external device.

Each transfer on this external SPI port consists of two 16-bit words. The first word is a command being sent from the external device. The second word is an internally generated response to a command. The command set is proprietary, and is only described at an abstract level in this paper. In general, there are three types of commands: (a) commands to set internal registers, (b) commands to read

internal registers, and (c) test commands (which include programming the EEPROM). Most commands are autonomous, and can be processed independent of other commands. This means that commands can be received in any order.

Each command received results in a response, which is generated by the Controller. Since the external device is the SPI master, the design cannot send the response as soon as it is ready (this would require that the design initiate a SPI transfer, which only the external master can do). Instead, the response to a command is returned when the external device sends a new command. Figure 2 illustrates this command/response relationship.

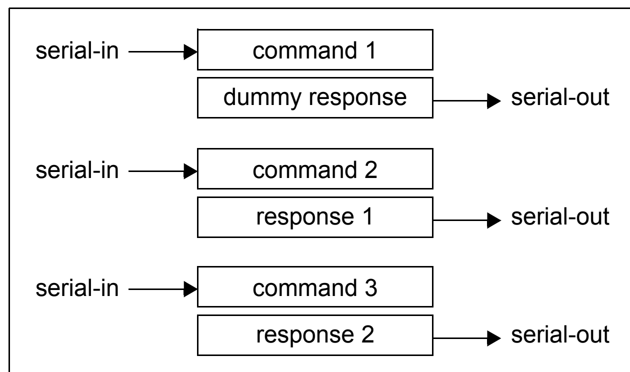


Figure 2. Command/response relationship

### B. *The EEPROM interface*

The EEPROM stores 8-bit words. Communication to the EEPROM is also serial, using a second Serial Peripheral Interface (SPI) port. On this port, the EEPROM is the slave, and the digital block is the master that controls the communication.

The protocol for the EEPROM SPI port is more complex than the external SPI port. A read or write transfer can be a single 8-bit word, or a burst of 16 8-bit words. A read or write transfer comprises an 8-bit control word followed by an 8-bit address, followed by 1 or 16 8-bit data words. The EEPROM also contains an internal status register. A status register read or write transfer comprises an 8-bit control word followed by an 8-bit data word. Finally, writing to the EEPROM can be enabled or disabled. This transfer comprises of a single 8-bit control word.

The EEPROM status register contains a code indicating if the previous operation was successful. Reading or writing to a non-existent address, or an invalid parity are examples that would result in a status that indicates a problem with the transfer.

Transaction-level verification would suffice to check that data written to the EEPROM could be read back again without corruption. Verifying the transfer protocol is more involved, however, since a transfer needs to have exactly 8, 16, 24, or 144 `spi_clock` cycles, depending on the type of operation. SystemVerilog assertions make it much easier to verify functionality that spans many clock cycles.

### C. *The controller*

The Controller module performs three primary functions. It decodes commands received from the external SPI port, causes the appropriate actions to occur within the design, and generates the response word to be sent on the next external SPI port transfer. Many of the actions generated by the controller can take hundreds of clock cycles to complete, and are controlled using sequences of handshake signals within the design. Once again, assertions are ideal for verifying handshake sequences that span multiple clock cycles.

### D. *Clocks*

The digital block of the design runs on a single master clock. The SPI port to the EEPROM uses a separate clock that is gated—the clock only runs when data is being transferred to or from the EEPROM. This gated clock is derived from the master clock. The external SPI port also uses a gated clock that only runs when data is being transferred, but this clock is supplied externally. This external clock is not associated with, or synchronized with, the internal master clock in any way.

## III. Overview of the Verification

The focus of the verification was on the digital sections of the design. The digital blocks were modeled at the synthesizable RTL level using Verilog-2001[3]. The EEPROM was a bus-functional behavioral IP model supplied by the maker of the EEPROM, and was also modeled using Verilog-2001. The analog section was not represented as simulation models for this verification project. Instead, the digital inputs from analog section were generated by the testbench.

A SystemVerilog VMM[2] testbench was under development when I began work on the project. The portion of the testbench that was complete was the driving stimulus into the design. A directed test loads the EEPROM with valid data using a back door (the Verilog $readmemb command). Once the EEPROM is loaded, a transaction generator generates random commands. Each transaction consists of both a command to be processed and digital inputs that would normally come from the

analog block of the design.

The next phase of the VMM testbench, which was not yet completed when I was brought in, was to add functional coverage, in order to ensure that the randomly generated stimulus was exercising all possible commands with meaningful range of values on the digital inputs from the analog block. A VMM scoreboard that would track the responses received from the design, and verify the correctness of each response, was also to be added.

Functional coverage is primarily based on values. For this specific design, functional coverage could be used to report how many times each possible input command occurred during simulation, and how many times each type of response occurred. When I examined the design specification to determine where functional coverage needed to be specified, it became apparent that it would be difficult to use SystemVerilog's value-based functional coverage to determine if the internal handshake sequences and the two SPI port protocols had been fully exercised.

For example, a command/response transfer on the external SPI port requires exactly 32 clock cycles to complete the transfer. If too few or too many clock cycles occur, then the transfer is invalid, and the design generates an error response. Several other conditions can occur as commands are executed that can also generate an error response. Value-based functional coverage can be used to determine how many times an error response occurred, but not *why* the error response occurred.

There was no straight-forward way for SystemVerilog's functional coverage to detect an incorrect number of clock cycles, and report how many times a SPI transfer had too few or too many clock cycles. SystemVerilog Assertions (SVA), on the other hand, allow for checking on sequences of signal changes that span multiple clock cycles. SystemVerilog Assertions can easily check for errors such as a SPI transfer not having the correct number of clock cycles.

Functional coverage has another limitation. It is useful for determining if critical parts of a design have been exercised during simulation, but functional coverage does not verify whether that stimulus results in unexpected or unintended design behavior. Assertions, on the other hand, can do both. Assertions can determine if some aspect of the design has been exercised (coverage), and whether that aspect of the design functioned as intended (verification).

For example, the EEPROM requires that a chip-select control line remain active throughout a SPI transfer to load values into the EEPROM. Functional coverage will report how many EEPROM operations occurred, but not whether those operation control line was set correctly during those write operations. An assertion can also report how many times a write operation occurred, but at the same time generate error messages should the chip-select control line not be set correctly during the operation.

## IV. Using Assertions for Coverage

SystemVerilog assertions can also be used for verification coverage—determining if some aspect of a design has been properly exercised during simulation. Three ways in which this can be accomplished are:

- Using the pass/fail reports of verification assertions (**`assert property`** statements)
- Using coverage assertions (**`cover property`** statements)
- Using functional coverage on assertion action blocks (**`covergroup`** and **`coverpoint`** definitions)

### *A. Using verification assertions*

The typical usage of assertions is to verify functionality in a design. With SystemVerilog assertions, the expected functionality is represented as a *property* specification. A simple example is:

```
property request_acknowledge;
   @(posedge master_clock)
   enable |-> request ##[1:3] acknowledge;
endproperty
```

This property states that if `enable` is true, then `request` must be true, followed 1 to 3 cycles later by `acknowledge` being true. The signals listed in the property are to be sampled on a positive edge of the `master_clock`, and a *cycle* is from one positive edge of the `master_clock` to the next positive edge.

A verification assertion statement then uses the property to check that the specified functionality holds true. An example verification assertion is:

```
assert property (request_acknowledge);
```

With this verification assertion, every time a `request` is followed by an `acknowledge` within 3 clock cycles, the assertion will pass. Should an `acknowledge` not occur within three cycles, the assertion will fail.

A verification assertion can be used for coverage information by checking the simulator report for how many times each assertion passed or failed. If the assertion passed at least once, then the verification stimulus caused that property sequence to occur.

The formatting for this type of reporting, or even how

the report is generated, is not part of the SystemVerilog standard. Each software tool that can execute SystemVerilog Assertions has its own proprietary way of reporting assertion pass/fail information. The Synopsys VCS simulator was used on this project, and the assertion pass/fail report was viewed as a table in the DVE graphical output display.

### B. *Using coverage assertions*

SystemVerilog coverage assertion is specified as:

```
cover property (request_acknowledge);
```

The difference between the **`assert property`** and **`cover property`** statements is that **`cover property`** only reports how many times the property passed. For example, it is not a coverage failure if a `request` is not followed by an `acknowldge` within three cycles. It only means that the specified sequence did not occur (it was not covered by the test stimulus).

### C. *Using functional coverage on assertions*

SystemVerilog's functional coverage provides a way to count how many times specific values occurred, or combinations of values occurred, during simulation. Functional coverage is reported separately and differently that coverage assertions.

It is possible to have the functional coverage report include how many times a verification assertion passed or failed. SystemVerilog assertions can specify two *action blocks*. One action block specifies statements to be executed when an assertion succeeds, and the other action block specifies statements to be executed each time an assertion fails. Using this, an assertion can change the value of a variable each time the assertion passes, and a different variable each time the assertion fails. These variables can then be monitored using SystemVerilog functional coverage. How many times that assertion passed or failed can then be included in the functional coverage report, along with other functional coverage points.

There is a limitation to reporting assertion passes using functional coverage in the current SystemVerilog 2005 standard, however. SystemVerilog Assertions have three types of outcomes: pass, fail, and vacuous success. A vacuous success occurs when an assertion triggers, but does not run to completion due to some guard condition not being true. Most assertions will have such a guard condition. For example, an assertion for a SPI transfer might have a guard that the assertion only run to completion if a SPI transfer has been started. If no transfer has started, the assertion immediately exits with a vacuous success, indicating the assertion neither passed nor failed.

The current limitation of trying to report how many times an assertion passes and fails as part of functional coverage is that even though an assertion can have three outcomes, there are only two action blocks: pass and fail. In SystemVerilog 2005, the passing action block will be executed both when the assertion passes *and* for each vacuous success. This will result in an incorrect functional coverage count of how many times the assertion actually passed. The next version of the SystemVerilog standard, which is currently being voted on for approval by the IEEE, will have a way to specify that only actual assertion passes should execute the passing action block.

## V. The Good Part of Adding Last Minute Assertions

The purpose of using assertions in this project was to check that the randomly generated test transactions effectively exercised all critical aspects of the design. Coverage assertions, as described above, were used to help evaluate the test coverage.

The coverage assertions that were added were effective. They found that the random transactions were not testing many critical portions of the SPI ports and internal handshaking. Additional randomization constraints and special directed tests were added, and 100% of the assertions were able to be triggered and pass multiple times.

The same properties used for coverage assertions were also used as verification assertions (there was a `cover property` and an `assert property` statement for each property specification). Once the appropriate randomization constraints were added, one of the verification assertions would occasionally fail, even though the functional values of the design responses seemed to be correct. It was found that there was a bug in a state machine that controlled the EEPROM SPI port. This design bug would likely not have been detected using just functional coverage. Had it been detected by functional verification, the bug might have been difficult to debug based solely on an incorrect response that was not generated until hundreds of clock cycles later.

The additional coverage reporting, verification, and debugging insight made adding the last-minute assertions successful and worthwhile.

## VI. The Bad Part of Adding Last Minute Assertions

A number of obstacles were encountered as coverage assertions were added to the design late in the design and verification cycle. These obstacles fall into four general categories: (a) incomplete or inaccurate design

specifications, (b) inconsistent naming conventions, (c) awkward design partitioning, and (d) gated clocks.

### *A. Incomplete or inaccurate design specifications*

The first task I undertook when I was brought in as a consultant was to review the design specification documents to see where assertions would be useful for verification coverage. One of the most difficult challenges that arose with adding assertions late in the design and verification process was incomplete design specifications. For example, the design specification for the external SPI port stated that:

> *`chip_select` is an active low signal, and is used to enable a transfer on the SPI port.*

This is a very vague and ambiguous specification! It does not state whether `chip_select` needs to remain low (active) throughout the SPI transfer. It also does not specify whether `chip_select` must go high (inactive) after each transfer, or if it can remain low (active), allowing multiple back-to-back transfers.

The philosophy at this company was that it was OK for the design specification to be vague. The ambiguity gave the design team more freedom on how to implement the design. The only requirement was that the design behavior match the big picture documented in the specification. The problem with ambiguous design specifications, however, is that it makes verification much more difficult. How can the verification team prove that design behavior is correct, if the specification does not define what is correct?

The project manager and the verification team for this project were located in the United States, and the design team was in India. Neither the verification team nor the project manager knew the answers to these questions. I had to send the design team the questions via e-mail, and wait until the next day to receive an answer. Two simple assertions that should have taken minutes to write ended up taking almost two days, instead.

The answer to these two questions was that `chip_select` had to remain low during the entire transfer; The designers used the `chip_select` input as an enable for an internal counter within design. The `chip_select` also had to go high (inactive) at the end of each transfer; The designers used the falling edge of `chip_select` to transition an internal state machine out of its wait state.

This same problem of ambiguous design specifications arose time and time again as I developed the assertion plan. Each time, precious and expensive engineering resources were lost determining what the intended design behavior should be.

My incessant questions regarding the specification did reveal flaws in the design that was supposed to be complete. In one case, an engineer on the design team had implemented an ambiguous design behavior in one way. My questions caused other engineers to consider what the intended behavior should be, and it was decided that the behavior needed to be different than the way it had been implemented.

The incomplete specification of `chip_select` (and hundreds of other ambiguous specifications) would have been discovered very early in the design process had the use of assertions been planned for, instead of being a last minute addition to the design verification.

### *B. Problems with naming conventions*

Inconsistent naming conventions for signals within the design was a frustrating problem. Each major module in the design was modeled by a different engineer. Some of the modules were re-used from previous designs. This resulted in signal names that were not consistent across module boundaries.

The design specification was just as inconsistent. The description different blocks of the design would refer to the same signal by different names. Often these names were different than what the engineers used in actual Verilog models.

The `chip_select` signal in the external SPI port, for example, had seven different names. The specification of the chip pinout called it `CS-`. Various paragraphs in the specification of the SPI port called it `chip select`, `chip-select`, `chip_select` and `I2C`. The actual RTL model of the SPI port called the signal `o_chip_select`, but the top-level netlist called it `cs_`.

The master clock in the design was called by at least a dozen different names. Had this been a multi-clock design, I might never have figured out which clock was being used where.

This inconsistency made it difficult to add assertions late in the design and verification process. A great deal of engineering time was wasted trying to figure out the actual names of signals when the names did not match the design specification, and the specification itself was not consistent. The solution for this problem would have been to select a naming convention very early in the specification of the design, and to enforce that convention in design reviews.

### C. Awkward design partitioning

The partitioning of the design into Verilog modules followed, for the most part, the functional block diagram in the design specification. The implementation of the specification, however, did not always put functionality in the same partition in which it was described in the specification.

For example, the external SPI port specification described how the command and response words were to be transferred serially. The specification of the Controller described how commands were decoded and responses generated. In the RTL models, however, the external SPI port did more than just transfer commands and responses. The SPI port also decoded some of the commands and set a flag if it was a test command. The decoding of the all other commands, however, was handled in the Controller module. Similarly, the generation of some error responses was done in the SPI port, but the generation of other error responses was done in the Controller.

The splitting of the command decoding and response generation into two modules made it difficult to write assertions. The assertions for each module did not appear to match the design specification, and it looked like some modules did not have coverage for all the conditions described in the specification. Similar situations occurred several times, where the description of a particular functionality was implemented in part or in whole in some other design module.

### D. Gated clocks

SystemVerilog concurrent assertions are cycle based. They must have a clock that triggers sampling values and evaluating expressions. The two SPI ports use gated clocks that are only running during an actual transfer. This made many of the assertions in the test plan difficult to write. For example, the external SPI protocol requires that an external device start a transfer by setting the active-low `chip_select` to zero, and remain low during the command/response transfer (a total of 32 bits). At the end of 32 `spi_clock` cycles, the `chip_select` must be set back to one. The `spi_clock` comes from the external SPI master, and only runs while `chip_select` is low. It is completely asynchronous to the internal `master_clock` used by the design, and might be faster or slower than the `master_clock`.

Part of a property definition is the assertion's cycle specification, which defines when values are to be sampled. The `spi_clock` cannot be used as the assertion cycle specification, since that clock is not running at the time that `chip_select` goes low. If the clock is not running, then the assertion would sample `chip_select`, and not detect when it when goes low.

The `master_clock` also cannot be used to verify this SPI port protocol. The `spi_clock` is asynchronous to, and perhaps faster than, the `master_clock`. If the `chip_select` is sampled on a `master_clock` cycle, one or more `spi_clock` cycles might have already occurred before the property sensed that `chip_select` was low. The property would also not correctly count how many `spi_clock` cycles occurred. There are other problems with using the `master_clock` to sample how many cycles occur on the asynchronous `spi_clock`, as well.

To work around this problem, the assertion code generated its own cycle clock, that ran at a higher frequency than any anticipated external `spi_clock` frequency.

## VII. Lessons Learned (a little too late)

**Lesson 1:** I reminded the project manager of this project a number of times that:

*"I can write the assertion to match how the design works, but all that does is prove that the design does what the design does. It does not prove that the design does what it is intended to do."*

An advantage of developing an assertion plan *before* the design is implemented is that it forces the verification team and the design team to read—*and think about*—the design specification. Ambiguities in the specification must be resolved order to specify the assertions. Adding assertions as an afterthought forced resolving ambiguities in the design specification, but most often that resolution was to change the specification to match how the design team had implemented the ambiguity in hardware.

**Lesson 2:** Inconsistency in signal naming conventions does not affect simulation or synthesis, but it does impact verification. Inconsistencies make it more difficult to add assertions to a design and could lead to incorrect or incomplete verification.

**Lesson 3:** Design partitioning needs to make sense for both synthesis and verification. Just because the RTL code synthesizes cleanly, does not mean the design can be verified cleanly. Developing an assertions plan early in the design cycle, instead of at the end, will help ensure that the design partitions are effective for both synthesis and verification.

**Lesson 4:** Gated clocks and cycle-based assertions are not compatible. There are ways to work around the

difficulties of using assertions when the clock is not always there, but it requires time, effort, and awkward verification kludges. Planning assertions early in the design process instead of at the end will at least make it evident that there will be some verification challenges, and might impact the decision as to when and where gated clocks should be used.

**Lesson 5:** Adding last minute assertions is better than not using assertions at all. When I was brought into the project late in the process, the design was complete and working (supposedly) and much of the VMM-based randomized transaction testbench had been developed. Coverage needed to be added to ensure that all critical aspects of the design had been verified. It was an afterthought to use assertions as one way to measure coverage. It was much more difficult to add assertions where there had been no plan for them. But is was still worth the time and effort. The assertions proved that more testing was needed, and revealed problems in the design.

## VIII. Conclusions

Assertions are a vital part of fully verifying a design. Some of the key benefits that came from adding assertions in this project were:

- Ambiguities in the design specification had to be resolved (which revealed some design flaws).
- Bugs in the design were discovered, even though the functional values appeared to be correct.
- Debugging was simplified; Assertions indicated exactly when and where a design bug occurred, which functional verification would not have been able to detect until many hundreds of clock cycles later.
- Greater coverage accuracy was achieved. Coverage assertions made it possible to determine if the randomized test transactions were exercising input error conditions which the design needed to handle.

Several challenges were encountered when adding assertions to the verification environment near the end of the design and verification process. Many of these challenges could have been avoided with forethought. Neither the design engineers nor the verification engineers had planned on using assertions, and therefore made many design choices that impacted the ability to effectively use assertions. Adding the last minute assertions took much longer (and was therefore more costly) than it would have been to plan for assertions early in the design cycle.

The mottos of “design for test” and “design for manufacturing” should be extended to verification, as well. All projects should have the motto of:

*Design for Verification!*

## References


[1] *“1800-2005 IEEE Standard for System Verilog: Unified Hardware Design, Specification and Verification Language”*, IEEE, Pascataway, New Jersey. Copyright 2005. ISBN: 0-7381-4811-3.

[2] *“Verification Methodology Manual for SystemVerilog”* (VMM), Bergeron, Cerny, Hunter, and Nightingale. Springer, New York, New York. Copyright 2006. ISBN: 0-387-25538-9.

[3] *“1364-2001 IEEE Standard Verilog Hardware Description Language”*, IEEE, Pascataway, New Jersey. Copyright 2005. ISBN:0-7381-2827-9.


## About the author


Stuart Sutherland is a member of the IEEE 1800 SystemVerilog standards committee, and is the technical editor of the 1800 SystemVerilog Reference Manual. Mr. Sutherland founded Sutherland HDL, Inc. in 1992. His company specializes in providing expert training and consulting on the proper usage of SystemVerilog. You can contact Mr. Sutherland at stuart@sutherland-hdl.com, or by calling 503-692-0898.