6.6 RTL Design

Assertions, generated during the architectural planning phases, greatly facilitate the writing of the RTL implementation because they help the designers to fully understand the architectural and interface requirements. The assertion statements do contain rules and restrictions that can easily be translated into RTL. Assertions also cause the designers to think about corner cases upfront in the process. For instance, in a FIFO controller design, a potential corner case is simultaneous *Read-and-Write* of an empty FIFO in which a "bypass" mode is required. The designer can make sure that the verification team exercises the corner cases through assertions.

During the coding of the RTL, it is recommended that designers add assertions and assumptions as executable documentation to be checked during the verification process. Note that assertions are not an equivalent representation of RTL code. They define a higher level of abstraction of the design. Assertions represent the expected behavior per requirements, as opposed to RTL that deals with detailed FSM architecture and timing. Thus, assertions cross-correlate implementation and provide another level of design verification. It is recommended that assertions be written in checkers (See Section 5 for guidelines and rationals). Such assertions significantly help in debugging failures because embedded assertions are very close to the source of error.

6.7 Testbench design

With the addition of SystemVerilog Assertions throughout the design cycle, the testbench design is greatly simplified because the verification of the interface and the requirement rules are defined in an executable assertion language. That greatly eases the need and the design of an automatic verifier. Errors in protocol or design can be quickly detected. The testbench consists of the instantiation of modules in the design, resets, clocks, transactors to define the test scenarios through tasks to be performed (e.g., WRITEs, READs), and drivers to translate the transactions into low-level signals. The testbench may possibly include a simpler automatic verifier.

6.8 Functional coverage in verification

With the recent advances in coverage technologies, functional coverage and assertion coverage have become quite significant in the verification process. Functional coverage is recognized as a means to ensure high quality of the design to be released to manufacturing. Testimonies on the values of functional coverage can be found in many books and papers. For example, in the verification of a *Frequency-Programmable Switch Chip*, the IBM design team experienced that "the implementation of coverage models provided a high level of confidence upon completion of the verification efforts and helped quantify the stress on the logic designs".[53]

Another interesting experience by a customer who used SystemVerilog Assertions in a battery-powered design was the benefit that coverage provided in the performance of the product. By the end of the project the top-level testbench did not show any functional bugs using the assertions. However, functional coverage revealed that one portion of the logic was performing unnecessary read/write operations to memory. Specifically, that portion of the logic had a high number of non-vacuous completions of memory accesses as compared to the non-vacuous completions of the logic that made use of those accesses. A correction to the logic to eliminate those unnecessary memory accesses yielded a 10% power efficiency increase. Considering that this design was battery powered, that 10% power efficiency was significant. Assertions with coverage helped in detecting the issues.

---

[53] http://www.research.ibm.com/journal/rd/483/hoppe.html or **http://tinyurl.com/ckye99** *Functional verification of a frequency-programmable switch chip with asynchronous clock sections*

Functional coverage can extend to several coverage domains. For example, in the functional verification of the *Z990 Superscalar* and *Multibook Microprocessor Complex*, the IBM design team identified several coverage domains.[54] These included, among other things the following:

- Program event recording coverage to track coverage events related to a specific domain or logic function, such as branch instructions, store instructions, and exceptions.
- Instruction coverage to verify that all instructions had been executed in all valid addressing, translation, and architectural modes.
- Architected features, such as all possible values for control registers, floating-point control register, and program status word.
- Superscalar grouping that defined which instructions could be executed simultaneously.

Though functional coverage has been well adopted in the industry, advanced users are soon realizing that having all functionalities covered (in addition to the usual code coverage) doesn't necessarily mean that all those functionalities are checked for correctness. Design Managers tend to think that functional coverage can find design errors; rather it is a measure of how much verification is done and not necessarily how "well" it is done. According to this referred IBM article, *the sole purpose of coverage is to measure test completeness and efficiency. Test correctness falls under functional verification*. In that project, the failing test cases were all screened, defects were opened on them, and fixes were eventually provided and verified. As a result, all test cases ran successfully.

While the authors believe strongly that use of functional coverage metrics is a must, they also insist that a vital link between functional coverage and correctness is even more important. One of the advantages of assertion-based coverage measurement is that the correctness is built-in into the assertions. In other words, with assertion-based coverage, functional correctness checking is the main product and functional coverage is a very valuable byproduct. But assertions, as in use today, tend to express localized behavior rather than end-to-end functionality. Hence the authors believe that assertion coverage and functional coverage will co-exist in verification.

One of the increasingly painful problems is the extraction of meaningful information out of these functional coverage statistics, and the determination of "how much verification is done". Traditionally, such an analysis is considered a tool issue. However, with recent advances in reactive testbench techniques, it is becoming more and more important to measure the coverage progress on the fly and to react quickly to the outcome. It is also important to be tool independent and yet achieve this goal. SystemVerilog Assertions provide a set of Application Programming Interface (API) routines that can be used to access the status of assertion evaluations within the verification environment. The API use model is presented in the next section with a small example.

6.8.1 SystemVerilog Assertions API

While SystemVerilog Assertions are very powerful in expressing design behavior, users may need more flexibility in the analysis and interaction of the verification environment based on results of the assertions. Examples of such applications include:

- Grouping "functionally" related `assert/cover` statements into functional categories and analyzing them in different perspectives.
- Building an advanced temporal debugger as an add-on to the SystemVerilog aware simulator.
- Reusing internal debugging tools originally based on proprietary assertion languages, thus permitting the smooth migration to SystemVerilog, a standard language.

---

[54] http://www.research.ibm.com/journal/rd/483/bair.html or http://tinyurl.com/dmexgw
Functional verification of the z990 superscalar, multibook microprocessor complex

- Building more automated testbenches using custom C-routines to interact with the assertion related events. For example, in an image processing application, a custom C-routine can be called to predict the output frame when an assertion evaluation starts.

Verilog has a standard API named "Programming Language Interface" (PLI). The recent PLI version 2.0 is referred to as "Verilog Procedural Interface (VPI)". SystemVerilog defines a Direct Programming Interface (DPI) that allows for calling C routines from SystemVerilog, and SystemVerilog routines from C. Many vendors have extrapolated this to C++ as well. It is much faster than PLI/VPI at the expense of error checking. SystemVerilog also defines a set of API routines to interact with the SystemVerilog Assertions and is called "SystemVerilog Assertions API".

While the exact functional prototypes of the SystemVerilog Assertions API itself are beyond the scope of this book, this section presents a brief overview of the functions and use model. Readers are encouraged to refer to the LRM for further details.

Since SystemVerilog Assertions API is an extension above the VPI, we will hereafter refer to it as VPI. The general flow of using VPI to extract information is a two-step process:

1. Obtain a "handle" to the assertions.[55]
2. Obtain the properties of assertions.

Calls to standard VPI routines, such as **`vpi_iterate`** and **`vpi_handle_by_name`**, can derive a list of assertions in the entire design or in a specific module. An analysis of assertions statistics will involve:

- Sorting the assertions based on their static characteristics such as name, instance name, module containing the assertions etc. For debug one might also be interested in the file name, line number etc. Such information is "static" in the sense that they remain unchanged throughout the simulation phase.
- Sorting the assertions based on their dynamic characteristics such as attempt started, attempt ended, attempt aborted, attempt stopped etc.

The static information about assertions can be retrieved using a single call to the API routine **`vpi_get_assertion_info`**`()` from an assertion handle. To retrieve the dynamic characteristics, the API extends the VPI call back routines to accommodate assertions and register call-backs on specific events, such as start, stop etc.

Building an Assertion Coverage Extractor using SVA API

The Assertion API capabilities are demonstrated with a simple example. The shown C-code is incomplete and represents pseudo-code to show intent.

For assertions, the coverage status property **`vpiCovered`** implies that the assertion has been attempted, has succeeded at least once, and has never failed. More detailed coverage information can be obtained for assertions by using different arguments to the **`vpi_get`** function. Table 6.8.1.1-1 lists various **`vpi_get`** arguments related to assertions.

[55] NOTES
1—As with all VPI handles, assertion handles are handles to a specific instance of a specific assertion.
2—Unnamed assertions cannot be found by name.

Table 6.8.1-1 vpi_get() Arguments and Usage

Argument to vpi_get() function	Usage
`vpiAssertAttemptCovered`	Extract the number of assertion attempts.
`vpiAssertSuccessCovered`	Extract the number of true (non-vacuous) successes.
`vpiAssertVacuousSuccessCovered`	Extract the number of vacuous successes.
`vpiAssertFailureCovered`	Extract the number of assertion failures.

Figure 6.8.1-1 represents SystemVerilog Assertions API C-code for an assertion statistics extractor. ***/ch6/6.8/vpi_sect6_2_7.c***

```
#include <stdio.h>
#include <vpi_user.h>
#ifdef VCS
  #include <vcs_vpi_user.h>
  #include <vcsuser.h>
#endif
#include <sv_vpi_user.h>
void assert_cov (int data, int reason)
{  vpiHandle assertion_h, itr_h;
  char * a_name;
  int a_attempts, a_true_success, a_vac_success, a_failures;
  itr_h = vpi_iterate(vpiAssertion, NULL);
  vpi_printf ("SystemVerilog Assertions Statistics - demo from SVA Handbook\n");
  vpi_printf ("Assertion Name: \t \t No. of True Success \t No. of Vacuous Success \t
                    No. of Failures \t No. of total attempts \n");
      while (assertion_h = vpi_scan(itr_h)) {
        a_name = vpi_get_str(vpiFullName, assertion_h);
        a_attempts = vpi_get(vpiAssertAttemptCovered, assertion_h);
        a_true_success = vpi_get(vpiAssertSuccessCovered, assertion_h);
        a_vac_success = vpi_get(vpiAssertVacuousSuccessCovered, assertion_h);
        a_failures = vpi_get(vpiAssertFailureCovered, assertion_h);
        vpi_printf ("%0s  %0d \t %0d \t %0d \t %0d \t %0d \n",
              a_name, a_true_success, a_vac_success, a_failures, a_attempts);
    } /* while */
} /* assert_cov */
```

Figure 6.8.1 SystemVerilog assertions API C-code for assertion statistics extractor.

A sample SystemVerilog model making use of the above C function is shown in Figure 6.2.7.2.1-2. This model does not show any RTL or assertions. However, types of results produced using these methodologies are shown in Figure 6.8.1-3.

/ch6/6.8/vpi_sect6_2_7.sv

```
module cov_api_test ();
  // Design has 3 assertions named as follows:
  // ap_rst_checks, ap_arm_arith_instructions, ap_arm_logical_instructions
  // The definitions of those assertions are not shown here as the goal of
  // this example is to demonstrate SVA API call.
  // Coverage collection,   other option could be to use vpiCallBack
  final
    begin : call_api
      …
      $assert_cov;  // call to API C-code function
  end
endmodule : cov_api_test
```

Figure 6.8.1-2 Sample SystemVerilog example for use of the coverage extractor

Table 6.8.1-2 shows arbitrary results that can be produced after simulation of the model.

Table 6.8.1-2 Sample Assertion Statistics

Assertion Name	No. of True Success	No. of Vacuous Success	No. of Failures	No. of total attempts
cov_api_test.ap_rst_checks_	**5**	**0**	**3**	**8**
cov_api_test.ap_arm_arith_instructions	**9**	**5**	**2**	**16**
cov_api_test.ap_arm_logical_instructions	**0**	**4**	**7**	**11**

6.8.2 Formal verification (FV)

If formal verification is available, the design can be exercised through the tool using the SystemVerilog assertions and assumptions already defined. Chapter 7 addresses the topic of formal verification.

6.9 Case study - synchronous FIFO

This section provides a definition of the requirements for a synchronous FIFO used as IP. This section demonstrates by example how properties unambiguously clarify requirements.[56] The RTL design with properties is also presented. A VMM testbench model is provided as a reference because it is derived from previous work .[57] An explanation of how the various pieces fit together is also demonstrated.

The numbering scheme for this specification and verification plan is separate from this chapter-numbering scheme, and thus, each document starts at the number ONE.

6.9.1 Synchronous FIFO Requirements

The requirements for a synchronous FIFO are provided in the following pages.

[56] Format for this specification is from the book *Component Design by Example*, 2001 ISBN 0-9705394-0-1, Ben Cohen

[57] The book *A Pragmatic Approach to VMM Adoption* 2006 ISBN 0-9705394-9-5 defines the use of the VMM methodology along SystemVerilog Assertions for the verification of the FIFO model.