System Verilog Coverage API

João Geada

Change Log

Version	Date	Authors	Description
0.1	01/20/2003	João	First draft proposal for incorporating coverage API donation into SV framework

Table of Contents

1 Requirements overview

1.1 SystemVerilog API

1. API should be similar for all coverages
   There are a wide number of coverage types available, with possibly different sets offered by different vendors. Maintaining a common interface across all the different types enhances portability and ease of use.
2. Support at minimum the following types of coverage
   a. statement coverage
   b. toggle coverage
   c. fsm coverage

        i. fsm states
        ii. fsm transitions
    d. assertion coverage
3. Set of coverages supported by API should be extensible in a transparent manner ie adding a new coverage type should not break any existing coverage usage.
4. API must provide means to obtain coverage information from only specific sub-hierarchies of the design without requiring full enumeration of all instances in those hierarchies by the user

1.2 VPI extensions API

1.3 Naming coventions

All elements added by this interface will confrom to the vpi interface naming conventions:

1. all names will be prefixed by vpi
2. type names will start with "vpi", followed by Capitalized words with no separators, eg vpiCoverageStmt
3. all function names will start with "vpi_" followed by all lowercase words separated by '_', eg vpi_control()

2 Coverage Definitions

Statement coverage: whether a statement has been executed or not, where statement is anything defined as a statement in the LRM. Covered means: executed at least once. Some implementations may also permit the execution count to be queried. Note that the granularity of statement coverage may be either per statement or per statement block (however defined) and neither approach will be favored by this standard.

Fsm coverage: the number of states in an FSM that have been reached by this simulation. Fsm automatic extraction is not required by this standard, but a standard mechanism to force specific extraction is available via pragmas.

Toggle coverage: for each bit of every signal (wire and register), whether that bit has had both a 0 value and a 1 value. Full coverage means both seen, otherwise partial coverage may be queried by some implementations. Some implementations may also permit the toggle count of each bit to be queried.

Assertion coverage: for each assertion, whether it has had at least one success. Implementations may permit further details to be queried, such as attempt counts, success counts, failure counts and failure coverage.

Note that the above defines the "primitives" for each coverage type. Over instances or blocks, the coverage number is merely the sum of all contained primitives in that instance or block (if/as appropriate).

3 SystemVerilog real-time coverage access

3.1 Predefined coverage constants in SystemVerilog

The following predefine 'defines will exist in SystemVerilog to represent basic real-time coverage capabilities accessible directly from SystemVerilog

Coverage control

```
`define SV_COV_START          0
`define SV_COV_STOP           1
`define SV_COV_RESET          2
`define SV_COV_CHECK          3
```

Scope definition (Hierarchy traversal/accumulation type)

```
`define SV_MODULE_COV         10
`define SV_HIER_COV           11
```

Coverage type identification

```
`define SV_ASSERTION_COV      20
`define SV_FSM_STATE_COV      21
`define SV_STATEMENT_COV      22
`define SV_TOGGLE_COV         23
```

Status results

```
`define SV_COV_ERROR          -1
`define SV_COV_NOCOV          0
`define SV_COV_OK             1
`define SV_COV_PARTIAL        2
```

3.2 Built-in coverage access system functions

3.3 $coverage_control

```
$coverage_control(control_constant,
                  coverage_type,
                  scope_def,
                  modules_or_instance)
```

Enables, disables, resets or queries the availability of coverage information for the specified portion of the hierarchy. The return value is an integer, with the value indicating the success of the action and will be one of

- SV_COV_OK
  Request successful. If starting or stopping or resetting this means the desired effect occurred, if querying, means coverage available. A successful reset clears all coverage (ie a …get() == 0 after a successful …reset())
- SV_COV_ERROR
  Call failed with no action, typically due to errors in the arguments, such as non-existing module or instance specifications
- SV_COV_NOCOV
  Coverage not available for the requested portion of the hierarchy

- SV_COV_PARTIAL
  Coverage only partially available in the requested portion of the hierarchy (ie some instances have the requested coverage information, some don't)

The hierarchy(ies) being controlled/queried are specified as follows:

`SV_MODULE_COV, "module name" :`
coverage of all instances of the given module (module name given as a string), but excluding any child instances in the instances of the given module

`SV_COV_HIER, "module name" :`
coverage of all instances of the given module including all the hierarchy below

`SV_MODULE_COV, instance_name :`
coverage of the one named instance. Instance is specified as a normal Verilog hierarchical path

`SV_COV_HIER, instance_name :`
coverage of the named instance plus all the hierarchy below.

3.4 $coverage_get_max

`$coverage_get_max(coverage_type, scope_def, modules_or_instance)`
Obtains the value representing 100% coverage for the specified coverage type over the specified portion of the hierarchy. This value will remain constant across the duration of the simulation. **Note:** This value is proportional to the design size and structure, so it should also be constant through multiple independent simulations and compilations of the same design, assuming that compilation options do not modify either coverage support or design structure. The return value is an integer, with the following meanings:

- -1 (SV_COV_ERROR): an error occurred (incorrect arguments)
- 0 (SV_COV_NOCOV): no coverage available for that coverage type on that hierarchy(ies)
- +ve: maximum coverage number, which is the sum of all coverable items of that type over the given hierarchy(ies).

Scope specified as per $coverage_control

3.5 $coverage_get

`$coverage_get(coverage_type, scope_def, modules_or_instance)`
Obtains the current coverage value for the given coverage type over the given portion of the hierarchy. This number can be converted to a coverage percentage by use of the equation $\mathrm{cov}\,erage\% = \frac{\mathrm{cov}\,erage\_get()}{\mathrm{cov}\,erage\_get\_\max()} * 100$. The return value follows the same pattern as …get_max(), but with the +ve number representing the current coverage level ie the number of the coverable items that have been covered in this hierarchy(ies)
Scope specified as per $coverage_control

3.6 $coverage_merge

$coverage_merge(coverage_type, "filename")

Loads and merges coverage data for the specified coverage into the simulator. The filename must not contain any directory specification or extensions, as these will dictated/assigned by the tool and may be implementation specific[1]. If "filename" does not exist or does not correspond to a coverage database from the same design an error will occur. If an error occurs during loading, the coverage numbers generated by this simulation might not be meaningful.

3.7 $coverage_save

$coverage_save(coverage_type, "filename")
Saves the current state of coverage to the tool's coverage database and associated with the name "filename". This filename must not contain any directory specification or extensions. Data saved to the database must be able to be retrieved later by the …merge function supplied the same name. Saving coverage must not have any effect on the state of coverage in this simulation.

4 FSM recognition

It is assumed that coverage tools will have automatic recognition of many of the common FSM coding idioms in Verilog/SystemVerilog. The standard will not attempt to describe or require any specific automatic FSM recognition mechanisms.
However, the standard will prescribe a means by which non-automatic FSM extraction will occur. The presence of any of these standard FSM description additions *must* override the tool's default extraction mechanism.
Identification of an FSM consists of identifying the following items

1. the state register (or expression)
2. the next state register (this is optional)
3. the legal states
4. the legal transitions between states

4.1 FSM pragmas[2]

FSM pragmas directly identify the state register, next state register (if any) and the valid states. From this information it is straightforward to identify the valid transitions from analysis of the Verilog source code.

State variable or expression identification

```
/* SV FSM state_vector <stateVarName> [<FSMName>] [enum <enumName>]*/
/* SV FSM state_vector <stateVarPartSel> <FSMName> [enum <enumName>] */
/* SV FSM state_vector <stateVarConcat> <FSMName> [enum <enumName>] */
```

Associate a reg, wire or parameter definition with a specific FSM

```
/* SV FSM enum <enumName> */
```

[1] ie tools are allowed to store coverage files any place they want with any extension they want *as long* as the user can retrieve the information by asking for a specific saved name from that coverage database
[2] full details given in an attached extract from VCS Coverage Metrics (VCM) documentation

5 VPI coverage extensions

5.1 Extensions to VPI enumerations

Coverage control

- #define vpiCoverageStart
- #define vpiCoverageStop
- #define vpiCoverageReset
- #define vpiCoverageCheck
- #define vpiCoverageMerge
- #define vpiCoverageSave

Coverage types

- #define vpiAssertCoverage
- #define vpiFsmStateCoverage
- #define vpiStatementCoverage
- #define vpiToggleCoverage

FSM types

- #define vpiFsm
- #define vpiFsmStates
- #define vpiFsmStateExpression
- #define vpiFsmStateValue

Coverage properties

- #define vpiCovered
- #define vpiCoverMax
- #define vpiCoveredCount

Assertion specific coverage properties

- #define vpiAssertAttemptCovered
- #define vpiAssertSuccessCovered
- #define vpiAssertFailureCovered

5.2 Obtaining coverage information

All use vpi_get() with the appropriate properties and object handles
coverage type, instance -> number of covered items in the given instance

vpiCovered, handle -> number of items of handle type covered. Only applicable to: statement handles, signal (wire/reg) handles, assertion handles, fsm handles

vpiCoveredCount, handle -> number of times each item of handle type covered. Only easily interpretable when handle points to a unique coverable item (otherwise sum of counts of all contained items)

vpiCoveredMax, handle -> total possible coverable items in the given handle. Handle types limited as per above. Note that this is only really useful when handle is a handle to an object possibly containing more than 1 coverable item.

Use vpi_iterate(vpiFsm, instance-handle) to get iterator to all fsms in an instance
Use vpi_get(vpiFsmStateExpression, fsm-handle) to get handle to the signal/expression encoding the Fsm state.
Use vpi_iterate(vpiFsmStates, fsm-handle) to get iterator to all states of an fsm
Use vpi_get(vpiFsmStateValue, state-handle) to get value of a state

5.3 Controlling coverage

vpi_control()
3 arguments: coverage control (start, stop, reset, check), coverage type, and handle to appropriate instance or assertion. Note that statement, toggle and fsm coverage are not individually controllable (ie controllable only at the instance level). Action as per the equivalent system function ($coverage_control)

vpi_control()
3 arguments: coverage control (merge, save), coverage type, “filename”. Merges coverage into the current simulation. Action as per the equivalent system functions ($coverage_merge, $coverage_save)