4.5.1 Purpose of verification statements

4.5.1.1 assert Statement

The **assert** statement is used to enforce a property as a verifier. It can report an action based on the pass or failure of the assertions. The **assert** statement is one of the key directives in verification because it makes a statement about what the desired property should be. The tool is responsible for reporting errors and for maintaining statistics about the assertions. A user-defined error or success message or action (e.g., modifying a value of a variable) may be defined in the action block (Section 4.5.1.6). A default message is provided.

4.5.1.2 assume statement

During various design phases (e.g., micro-architecture definition, RTL coding etc.), architects and designers make a number of assumptions about the environment in which this design is expected to perform. Traditionally, these assumptions are captured using comments in the RTL code, or as text in requirement or verification planning documents. Assumptions are important because they define the limits and restrictions imposed on the design and supporting tools. In simulation, assumptions impose limits on scenarios. In formal verification, assumptions speedup the formal verification process and avoid false negatives (i.e., a false error reporting because the conditions causing the error is not expected to ever happen). For example, if parity mode is hard-wired to *no-parity* then any logic (and assertion) affected by parity being active can be ignored since this hardware never uses parity. Other examples where assumptions can be offered to a formal verification tool can be that the *test mode* is in the inactive state, and that the *reset* is active for 10 clocks and low forever. To support such features, SystemVerilog supports the **assume** statement.

The purpose of the **assume** statement is to allow properties to be considered as assumptions or constraints for formal analysis, as well as for dynamic simulation tools. When an expression or a property is assumed, the formal verification tools constrain the design inputs so that the property holds. For example, an asserted property is required to hold only along those paths that obey the assumption. In some cases, constraints can be interpreted as properties to be proved. For example an input constraint associated with one module can also be an output property of the module driving this input. Therefore, properties and constraints may be *dual* in nature.

The exact way in which an **assume** construct is handled depends partly on the verification methodology. A SystemVerilog aware simulator treats them as verifiers to monitor the input stimuli and will report a failure if the property in the **assume** statement is false.

The role of the **assume** construct is useful in the confirmation of the design environment. It is also more visible and valuable in formal verification. Formal verification tools consider all possible allowed input combinations when performing checks and working on a proof, and hence need to constrain the inputs to their legal behavior. Otherwise, the tool would either report false negatives or may not terminate at all.

Notes:

1. For formal analysis, there is no obligation to verify that the assumed properties hold.
2. An assumed property can be considered as a hypothesis to prove the asserted properties.
3. For simulation, the environment must be constrained such that the properties that are assumed shall hold. Thus, an **assert property**, or an **assumed property** must be checked and reported if it fails to hold. In simulation the **assume property** behaves like an **assert property**.
4. Biasing the inputs with the **dist** operator and the **assume property** provides a way to make random choices. Typically, this biasing is not used by formal verification tools. It could however be used by a power analysis tool where, based on a use distribution, the tool could calculate the power distribution consumption within the design; this analysis could identify regions of the design that needs power optimization. When a property with biasing is used in an assertion statement, the **dist** operator is equivalent to **inside** operator, and the weight specification is ignored. For example:

```
mp_req_distribution_1: assume property (@ (posedge clk)
        if (req dist {0:=40, 1:=60}) `true ); //40% req==0, 60% req==1
```

In simulation, `mp_req_distribution_1` the **dist** is interpreted as **inside**. This assumption will never fail. This is equivalent to:

```
req_distribution_2: assume property (@ (posedge clk)
        if (req  inside {0, 1}) `true);
```

The `req inside {0, 1}` states that `req` can take the values `0` and `1`, with no distribution.

4.5.1.2.1 `assert` and `assume` for same property: then what?

Having both the **`assume`** and the assert statement for the same property or for elements of the same properties seems contradictory because the assert directive is a requirement that the property must hold under all circumstances, yet the assume directive defines an environment under which the assertions are verified. However, for one block an assertion may need to be proven, while for another block that same assertion is an assumption. Generally formal tools have a way to allow changing an assertion to an assumption and vice-a-versa. There is no need to create both in the RTL.

4.5.1.2.2 Same inputs in antecedent and consequent

Inputs are sometimes used in both the antecedent and consequent of a property. They are often useful in simulation to ensure compliance to protocol by either the higher-level design or by the testbench. For example, the property pACK_START_VALID_DONE states that both the trigger condition (the antecedent) and the EFFECT sequence (consequent) rely on the input *ack_in*.

```
sequence qACTIVE_REQUEST;  req && !ack_in;
 property pACK_START_VALID_DONE;
   @ (posedge clk) disable iff (!reset_n)
   qACTIVE_REQUEST ##1  ack_in |=>
       start && !ack_in && valid ##1
        (!start && !ack_in && valid)[*3] ##1
        (!start && !valid && done);
  endproperty : pACK_START_VALID_DONE
 apACK_START_VALID_DONE  : assert property (pACK_START_VALID_DONE);
```

ack_in used in antecedent

input *ack_in* also used in consequent

When the same input signals are used in the antecedent and consequent side of a property, false negative errors in formal verification can be created because the formal verification tool can assume inputs to have any value at any cycle. This is not a design fault, but rather an error in the definition of the constraints on the inputs. An example of a constraint for the above case would be:

```
mpCONSTRAINT_ACK_START_VALID_DONE  : assume property(@ (posedge clk)
      qACTIVE_REQUEST ##1  ack_in |=> !ack_in[*1:$] ##1  (!start && !valid && done));
```

The `mpCONSTRAINT_ACK_START_VALID_DONE` assumption states that one clock after state `qACTIVE_REQUEST`, if `ack_in ==1`, then it must be zero starting from the next cycle until `(!start && !valid && done)`.

4.5.1.3 restrict statement

🕮 **Rule:** *[1] In formal verification, for the tool to converge on a proof of a property or to initialize the design to a specific state, it is often necessary to constrain the state space. For this purpose, the assertion statement* **`restrict property`** *is introduced. It has the same semantics as* **`assume property`***, however, in contrast to that statement, the* **`restrict property`** *statement is not verified in simulation and has no action block.*

The difference between the **`restrict`** statement and the **`assume`** statement is that the **`restrict`** is used to limit scenarios to converge on a proof, whereas the **`assume`** defines legal input states. For example, suppose that a cache controller performs `behavior A` when there is a cache hit (*e.g., fetch data from the cache*), or performs `behavior B` when there is a cache miss (e.g., *invalidte cache entry, fetch data from main memory through an interface, store data into the cache, supply the needed value*). To simplify the problem for the formal verification tool, one could constrain or "restrict" the problem to one of the two cache modes. Thus, the following can be written in one set of scenarios:

```
restrict property (@(posedge clk) cache_hit == 1'b0);   // cache miss scenario
```

Another formal verification session could use this restriction:

```
restrict property (@(posedge clk) cache_hit == 1'b1);   // cache hit scenario
```

Another example of a **`restrict`** statement is the *reset* signal that can be classified as "one-shot" (i.e., one-time) that has a specific behavior during initialization. In this case, one needs to restrict the signal `reset_n` to low for 1 to 100 cycles, and then to high forever. Thus,:

```
initial
ap_reset_then_hi : assume property ( @ (posedge clk)
     !reset_n[*1:100] ##1 reset_n |=>  always (reset_n));
```

This is a good use of the **restrict** because it is never illegal for reset_n to go active, but it is a common scenario to limit possible scenarios.

In formal tools, you can typically re-qualify each assertion as **assume**, **assert**, or **cover** because the role of some assertion in a whole system may actually be different depending on what "part" of the system you are currently examining with formal. However, there is no need for the same re-qualification for the **restrict** statements - they mean one and only one thing, and the **restrict** should not be used as an **assert** or **assume** statement..

Guideline: Use **assume** to define legal input states that prevent false failures. Use **restrict** in formal verification when you are just trying to reduce the state space by limiting a test to one of several legal scenarios.

4.5.1.4 cover statement

There exist three categories of cover statements, **cover sequence** and **cover property,** immediate **cover**. The **cover sequence** statement specifies sequence coverage, while the **cover property** statement specifies property coverage. The syntax for the **cover** statement is:

statement_or_null is executed every time a property attempt succeeds nonvacuously

```
cover_property_statement::=
   cover property (property_spec) statement_or_null

cover_sequence_statement::=
   cover sequence (
    [clocking_event] [disable iff (expression_or_dist)]
                    sequence_expr) statement_or_null
simple_immediate_cover_statement ::=
   cover ( expression ) statement_or_null
deferred_immediate_cover_statement ::=
   cover #0 ( expression ) statement_or_null
 | cover final ( expression ) statement_or_null
```

statement_or_null is executed every time a sequence thread succeeds. Use **first_match** (seq_expression) if interested in one match

Note: During simulation, it is possible to detect that a property is covered by querying the **vpi_get()** function with the **vpiAssertSuccessCovered** argument (see 6.2.7.2). For simulation efficiency, one can then turn off the *assertion_identifier* for the coverage (e.g., **$assertoff(1, cp_q_abc)**) when coverage is reached.

Rule: The pass statement in the action block must not include any concurrent **assert**, **assume** or **cover** statement. Coverage results are divided into two: coverage for properties, coverage for sequences.

[1] The results of coverage statement for a property contain:

- *Number of times attempted*
- *Number of times succeeded*
- *Number of times succeeded because of vacuity*

```
default clocking @(posedge clk); endclocking
cp_ab: cover property(a |=> b);
```

The following files represent an example of the cover property and the resulting statistics: *ch4/implication.sv*, *implication.bmp, implication.jpg, implication_assertion_report.txt, implication_fcover_report.txt*.

[1] Results of coverage for a sequence include:

- *Number of times attempted*
- *Number of times matched (each attempt can generate multiple matches)* (see 1.3.2 and 2.3.2). *In addition, statement_or_null gets executed for every match. If there are multiple matches at the same time, the statement gets executed multiple times, one for each match.*

```
cq_ab: cover sequence(a |=> b);
```

[1] The immediate **cover** *statement specifies that successful evaluation of its expression is a coverage goal. The results of coverage for an immediate* **cover** *statement shall contain the following:*

- *Number of times evaluated*
- *Number of times succeeded*

```
c_ab: cover (a && b);
```

A pass statement for an immediate **cover** *may be specified in statement_or_null. The pass statement shall be executed if the expression evaluates to true. The pass statement shall be enabled to execute immediately after the evaluation of the expression of the immediate* **cover**.