sequence q_abc2012; a ##1 b ##1 c; endsequence : q_abc2012
ap_qabc_de2012: assert property(@ (posedge clk)
    if (q_abc2012.triggered) d|=> e); // if end of sequence q_abc2012, then
// if d, e should occur at next cycle

Figure 1.3.2 Application of sequences (ch1/seq1.sv)

1.4.3 Assertion states

An assertion is a directive that a given property is required to hold. An assertion can be in several
states, and it is symbolically represented in waveforms for clearer debugging. The convention
used in QuestaSim (Mentor Graphics) is explained below:

- **Attempted:** When the assertion is attempted at its leading clocking event, it is
  considered as a start event, represented as a blue square. In essence, an attempt
  is the start of an evaluation of an assertion (see 2.3.1 for more information on attempts).
  Assertion statements are started (attempted) at every leading clocking event, unless they
  are disabled or if they are turned off. **If the attempt succeeds, one or more threads
  for that successful attempt are started, and those threads are independent from
  previous or future attempts.** If the attempt fails, that assertion for that attempt is either
  vacuous if the property has an implication operator, or it fails if there is no implication
  in the property.

- **Failed:** An assertion fails when it does not meet the requirements of the property. If the
  assertion fails, it is represented as a red inverted triangle at its time of failure.

- **Pass:** An assertion succeeds or passes if it meets the requirements of the property. A
  pass can be either vacuous or nonvacuous. If the assertion passes nonvacuously (i.e., *it
  has significance in that it met the requirements*), it is represented as a green equilateral
  triangle at its end point, the cycle of the pass. A successful thread of an assertion is
  represented by a yellow triangular flag. A successful nonvacuous pass of an assertion,
  contributes to the pass statistics. An assertion can also succeed vacuously because its
  result is of little consequence or interest. For example, if we’re interested in observing
  an acknowledge if request is true, but we don’t care about the value of the acknowledge if
  request is false; thus, an assertion with a false request is considered vacuous. In that
  case, the simulator displays no symbol at its end point, the activity (green trace) stops,
  and no statistics are collected.

- **Active:** During the progression of a started assertion, the activity of the assertion is
  represented with a green trace from its start point until success or failure or vacuity.
  QuestaSim use a separate thread viewer to show the various threads within an attempt.
  At the end of a simulation run (e.g., with a $finish) there could be some pending,
  unterminated, assertions. 1 [1] For any assertion, the number of attempts that have not
  yet reached any conclusion (success, failure, disabled, or killed) can be derived from the
  formula:

  \[
  \text{in progress} = \text{attempts} - (\text{successes} + \text{vacuous success} + \text{disabled} + \text{killed} + \text{failures})
  \]

  “In progress” is often referred as “unfinished” at the end of simulation.

- **Inactive:** An assertion is inactive if its attempt is vacuous at the leading clocking event,
  or if it was never attempted because the leading clocking event for the assertion has not
  yet occurred. An assertion is demonstrated as inactive using a blue trace (in the
  assertion trace) or no trace (in the active count).

**IMPORTANT CONCEPTS**
Outcomes: An assertion can have one of the following outcomes after it was started with a successful attempt:

1. **Success**: An assertion succeeds when the property of the assertion holds. An assertion can succeed vacuously or non-vacuously. The success of a thread of an assertion does not necessarily indicate the successful termination of the attempt because there may be a need for all the threads to be exercised to declare the success (i.e., pass) of the assertion (see 2.3.2, and Appendix B for definition of a thread). A property will also succeed vacuously when the condition on the `accept_on` operator is true (see 3.9.2.14). As mentioned above, a property that evaluates vacuously in simulation does not contribute to the pass/fail counters.

2. **Failure**: An assertion fails when the property of the assertion does not hold. A property will also fail when the condition on the `reject_on` operator is true (see 3.10.9).

3. **Disabled**: An assertion can be disabled asynchronously with the `disable iff` when the disabling condition is true (see 3.8). A disabled assertion does not contribute to the statistics.

4. **Killed**: An assertion can be aborted and stopped with the system function `$assertkill` (see 4.2.4.1). A killed assertion does not contribute to the statistics.

The following is an example of a sequence declaration and an assertion of this sequence. Note that assertions of properties that are strictly sequences are rare; it is shown here for demonstration purposes:

```verilog
class qAB2C;
  $rose(a) ##1 b ##2 c; // single thread sequence (see 2.3.1 for definition)
endclass

class ap_qAB2C;
  assert property(@(posedge clk) qAB2C);
endclass
```

The evaluation of `ap_qAB2C` assertion is demonstrated with a timing model in Figure 1.4.3-1. At every positive edge of a clock cycle a clocking event occurs (called the leading clocking event) and a new attempt is started. Within an attempt, the following evaluations occur:

1. If `a` did not rise (i.e., `$rose(a)==0`), then the attempt for the assertion fails in that cycle.\(^{18}\)
   
   The assertion fails in this case because the property is a sequence, and for a thread of an assertion to pass all terms in the sequence for that thread must succeed at their evaluation points. If the first term fails, then obviously the assertion fails. An multithreaded assertion of a sequence is considered passing and covered when one of the threads succeeds (See 2.3.1, `ch1/mabc.sv, ch1/mabc.jpg`).

2. For each attempt, if `a` rose to true, then a single thread (in this case) is started, and the evaluation of the assertion is active and continues.

3. At the next cycle within that thread, if `b` is true the evaluation of the assertion stays active and continues. Otherwise, that attempt of the assertion fails.

4. The next cycle within that thread is true because `##2` is equivalent to `##1 1'b1 ##1`. If in the cycle after that `c==1'b1`, then the attempt for that single-threaded sequence passes, otherwise that attempt of the assertion fails.

---

\(^{18}\) `$rose(a)` is true if `a` was `1'b0` at the previous clocking event and is `1'b1` in the current cycle.
Every asserted / assumed / or covered property or covered sequence is checked at every leading clocking event (such as @ (posedge clk)). Many applications have a triggering sequence that causes the attempted assertion to start and to continue its evaluation. For example, if a bus_request occurs (i.e., a sequence of one cycle duration), then an acknowledge should occur at the next cycle. If the triggering sequence does not occur (i.e., bus_request is false), then there is no need to further evaluate the attempt (i.e., the rest of the property for that cycle) because it is not an interesting case worth considering. In that case the property is evaluated as vacuous success (also called true, vacuous). To achieve this cause-effect of thread evaluations, SystemVerilog introduces the concept of antecedent and consequent connected with implication operators “|->” (overlapped, meaning that the next sequence starts in the same cycle as the end point of the causing sequence), or “|=>” (non-overlapped, meaning that the next sequence starts in the cycle following the end point of the causing sequence). For example,

```
property p_handshake; // See 8.1 for naming convention
  Antecedent (or cause). If FALSE then property succeeds vacuously. If true, then the evaluation of the property continues.
  request |=>
  Consequent (or effect). The sequence is evaluated if antecedent succeeds.
  acknowledge #1 data_enable #1 done;
endproperty : p_handshake
ap_handshake : assert property (@ (posedge clk) p_handshake);
```

Property p_handshake states that if the antecedent sequence request (one-cycle sequence in this case) is true then the consequent is required to be true, otherwise the property fails. The attempt is started at every leading clocking event or the posedge clk in this case. If the antecedent fails (i.e., evaluates to 1'b0), then the assertion attempt succeeds vacuously. If the antecedent succeeds, then the attempt is further evaluated at the next cycle until either the consequent succeeds or fails. If the consequent succeeds at its completion, then the assertion passes nonvacuously. Figure 1.4.3-2 demonstrates two active attempts for this assertion.
Figure 1.4.3-2 Two active threads for the Handshake Property
(ch1/reqack.sv, reqack.bmp (wave), REQACK1.jpg, REQACK2.jpg (thread viewer))
Thread T1 of attempt at 100 ns Completes at Time 250 ns,
Thread T2 of attempt at 150 ns Fails at Time 250 ns (data_enable==1'b0, expected a 1'b1)

Note: An assertion of a property that is only a sequence is rarely used because of overlapping issues, thus causing unexpected errors. To avoid such errors, sequences are typically used in properties with implication operators, as shown in the previous example. However, sequences are used for functional coverage using the cover sequence that counts all matches, or cover property that counts only one match; These cover statements are used as a measurement that the verification environment created a test case representing such sequences.

1.5 Assertion-based verification

1.5.1 Specification and verification
A specification is a set of properties that may be executable. From a SystemVerilog Assertions viewpoint, verification is the confirmation that, for a given design and a given set of constraints (e.g., assumptions), a property that is required to hold in that design actually does hold under those constraints

Assertion-Based Verification (ABV) is a verification and documentation methodology. It instruments system requirements, interfaces, design characteristics, and verification environments with assertions. SystemVerilog Assertions is an assertion language tightly coupled to SystemVerilog for the definition, declaration, and verification of properties. Verification is the process used by the verification tool, such as simulator or formal verification tool (see Chapter 7), to insure that the design complies with the declared assertion directives of the properties. Thus, a declaration (e.g., named property) does not cause the verification tool to check the property. An assertion statement, also called directive, is needed for that (see 1.3, 4.5, 4.6).