Accellera SystemVerilog Assertions
Design Working Group - Working Proposal (Rev0.75)
11/21/02

Section 11 Assertions

This is a working proposal for SystemVerilog Assertions. There are still a number of issues to be settled, but this is an attempt to document what has been proposed and/or agreed to by the DWG. There are a number of outstanding issues still to be decided:

NOTE: In Rev0.6, the following list of issues may be (and probably is) incomplete.

- Mixing clock expressions in a single sequential expression
- Recognizing sequences from one clock domain in a different clock domain
- Formal semantics need to be defined
- Syntax for declaring properties
- Language/tool control for assume/restrict/cover directives
- Non-clocked sequences and assertions

11.1 Introduction (informative)

An assertion is a statement that a property must be true. There are two kinds of assertions: concurrent or immediate.

Immediate assertions follow event semantics for their execution. They get evaluated like a statement in a procedural block.Immediate assertions are intended to be used with simulation.

Concurrent assertions are based on clock semantics and use sampled values of variables. One of the goals of SystemVerilog assertions is to provide a common semantic meaning for assertions so that they may be used to drive various design and verification tools. Many tools, such as formal verification tools, evaluate circuit descriptions using a cycle-based semantic which typically relies on a clock signal or signals to drive the evaluation of the circuit. Any timing or event behavior between clock edges is abstracted away. Concurrent assertions incorporate this clock semantics. While this approach generally simplifies the evaluation of a circuit description, there are a number of scenarios under which this cycle-based evaluation provides different behavior from the standard event-based evaluation of SystemVerilog described elsewhere in this document.

This chapter describes both types of assertions.

11.2 Immediate assertions

The immediate assert statement is a test of an expression performed when the statement is executed in the procedural code. The expression is treated as a condition like in an if statement.

immediate_assertion ::=

[identifier **:**] **check (** expression **)**;

| [identifier **:**] **check (** expression **)** action_block

action_block ::=

do statement

| **else** statement

| **do** statement **else** statement

The statement associated with **do** is called pass statement, and is executed if the assertion succeeds, i.e. the expression evaluates to true. As with the **if** statement, if the expression evaluates to 'X', 'Z' or '0', then the assertion fails. The pass statement may, for example, record the number of successes for a coverage log, but may be omitted altogether. If the pass statement is omitted, then no action is taken if the assert expression is true. The statement associated with **else** is called fail statement, and is executed if the assertion fails (i.e. the expression does not evaluate to a known, non-zero value) and can be omitted. The optional assertion label (identifier and colon) creates a notional named block around the assertion statement (or any other SystemVerilog statement) and can be displayed using the %m format code.

```
assert_foo : check (foo) do $display("%m passed"); else $display("%m failed");
```

Since the assertion is a statement that something must be true, the failure of an assertion shall have a severity associated with it. By default, the severity of an assertion failure is "error". Other severity levels may be specified by including one of the following severity system tasks in the fail statement.

— **$fatal** is a run-time Fatal, which terminates the simulation with an error code. The first argument passed to $fatal shall be consistent with the argument to $finish.

— **$error** is a Run-time Error.

— **$warning** is a Run-time Warning, which can be suppressed in a tool-specific manner.

— **$info** indicates that the assertion failure carries no specific severity.

The syntax for these system tasks is shown in section 16.4 of System Verilog3.0 LRM.

All of these severity system tasks shall print a tool-specific message indicating the severity of the failure, and specific information about the specific failure, which shall include the following information:

— The file name and line number of the assertion statement,

— The hierarchical name of the assertion, if it is labeled, or the scope of the assertion if it is not labeled.

For simulation tools, these tasks shall also include the simulation run-time at which the severity system task is called.

Each system task can also include additional user-specified information using the same format as the Verilog **$display**.

If more than one of these system tasks is included in the **else** clause, then each shall be executed as specified.

If an assertion fails and no **else** clause is specified, the tool shall, by default, call **$error**, unless a tool-specific command-line option is enabled to suppress the failure.

If the severity system task is executed at a time other than when the assertion fails, the actual failure time of the assertion can be recorded and displayed programmatically. For example:

```
time t;
```

```
always @(posedge clk)
   if(state == REQ)
      check(req1 || req2)
      else begin
         t = $time;
         #5 $error("assert failed at time %0t",t);
      end
```

If the assertion fails at time 10, the error message will be printed at time 15, but the user-defined string printed will be “assert failed at time 10”.

The display of messages of warning and info types can be controlled by a tool-specific command-line option.

Since the fail statement, like the pass statement, is any legal SystemVerilog procedural statement, it can also be used to signal a failure to another part of the testbench.

```
check (myfunc(a,b)) do count1 = count + 1; else ->event1;
check (y == 0); else flag = 1;
```

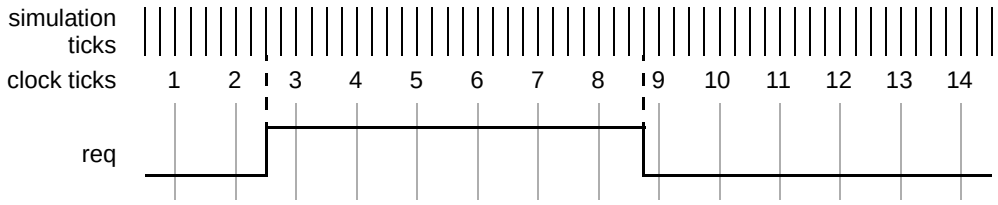
11.3 Concurrent assertions

Concurrent assertions describe behavior that spans over time. The evaluation model is based on a clock such that a concurrent assertion is evaluated only at the occurrence of a clock tick. The values of variables used in the evaluation are the sampled valued. This way, a predictable result can be obtained from the evaluation, regardless of the simulator’s internal mechanism of ordering events and evaluating events. This model of execution also corresponds to the synthesis model of hardware interpretation from an RTL description.

The timing model employed in concurrent assertion specification is based on clock ticks, and uses a generalized notion of clock cycles. The definition of a clock is explicitly specified by the user, and can vary from one expression to another. In addition, a user can choose to use the simulation time as a clock to express asynchronous events.

A clock tick is an atomic moment in time and implies that there is no duration of time in a clock tick. It is also given that a clock may tick only once at any simulation time. The value of a variable in an expression at a clock tick is sampled at the end of one simulation timestep (i.e. at read-only synchronization time, as defined by the PLI) before the clock tick. In an assertion, the sampled value is the only valid value of a variable at a clock tick. Figure 11-1 shows the values of a variable as the clock progresses. The value of signal `req` is low at clock ticks 1 and 2. At clock tick 3, the value is sampled as high and remains high until clock tick 9. The value of variable `req` at clock tick 9 is low and remains low.

Figure 11-1—Sampling a Variable on Simulation Ticks



The sampled value of a signal with respect to its clock is the value of the variable at the end of the simulation time (i.e. read-only sync) before the clock event occurs.

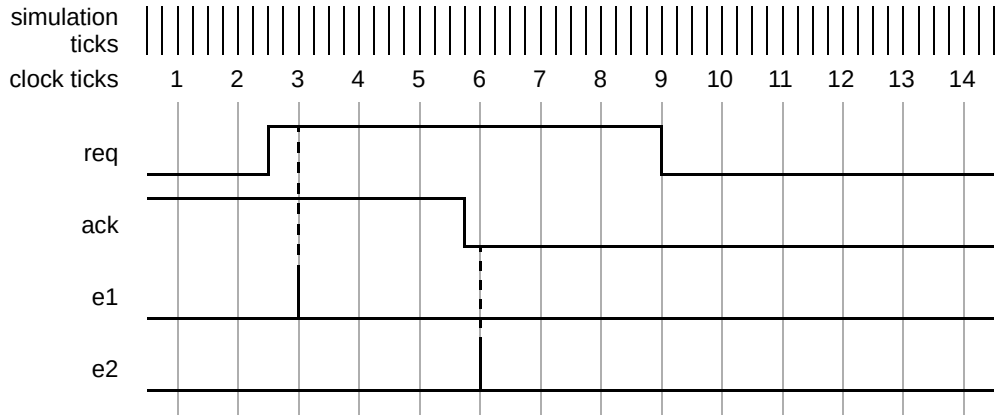
An expression is always tied to a clock definition. The values of variables are sampled only at clock ticks. These values are used to evaluate edge expressions (such as rose and fell) or boolean sub-expressions that are required to determine a match with respect to a sequence expression.

An edge expression at a clock tick changes the value of an expression from the value of that expression at the previous clock tick. Like boolean expressions, an edge expression evaluates to true if the change occurs, and to false if the change does not occur.

For example, when a signal changes its value from low to high (a rising edge), it is considered a rose edge. Figure 11-2 illustrates two examples of edges:

- edge expression `e1` is defined as `(rose req)`
- edge expression `e2` is defined as `(fell ack)`

Figure 11-2—Edge Expressions



The clock used for sampling the events is different than the simulation ticks. Assume, for now, that this clock is defined in this language elsewhere. At clock tick 3, edge `e1` occurs because the value of `req` at clock tick 2 was low and at clock tick 3, the value is high. Similarly, edge `e2` occurs at clock tick 6 because the value of `ack` was sampled as high at clock tick 5 and sampled as low at clock tick 6.

NOTE: A vertical bar, in figures like Figure 11-2, without an arrow on the top or the bottom of the bar indicates an occurrence of an edge expression.

The clock expression that controls evaluation of a sequence may be more complex than just a single signal name. An expression such as (clk && gate) could be used to represent a gated clock. Other more complex expressions are possible. In order to ensure proper behavior of the system and conform as closely as possible to truly cycle-based semantics, the signals in a clock expression must be glitch-free and may only transition once at any simulation time. The clock expressions must be evaluated with zero delay.

11.4 Declaring Clocks

To declare a clock (sample events) on which to sample the signals in an assertion, the standard event declaration syntax is extended:

event_declaration ::= **event** identifier [**=(**<event_expression>**)**] { **,** identifier [**=(**<event_expression>**)**] **;**

The event assignment is required for the event to be used as a clock in an assertion. If the event_expression is not specified, the event is used as a regular verilog event. Whenever the event_expression evaluates to true, the event gets triggered and is recognized as a sampling event in an assertion.

NOTE: The two words “clock tick” and “sampling event” are used synonymously in this document.

When an event is defined in this way by an explicit event expression, the event may not be explicitly triggered from anywhere in the code. For example the following code illustrates an illegal triggering of an event.

```
event myclk = (posedge clk1);
always @(posedge clk2)
```

```
if(foo)
   -> myclk; // illegal
```

A clock may also be specified directly along with an assertion without naming or declaring the clock as an event as shown above. The event declaration method, however, allows to identify a clock expression so that it can be commonly used for multiple assertions.

11.5 Sequences

A sequence is a list of SystemVerilog boolean expressions in a linear order of increasing time. These boolean expressions must be true at those specific points in time for the sequence to be true over time. A boolean expression at a point in time is a simple case of a sequence with time length of one unit.To determine a match of a sequence, the boolean expressions are evaluated at each successive sample point to satisfy the sequence. If all expressions are true, then a match of the sequence occurs.

A sequence expression describes one or more sequences by using *regular expressions* that concisely specify a range of possibilities of and repetitions of sequences. These sequential regular expressions can actually describe a set of one or more sequences that satisfy the sequential expression.

A sequential regular expression is a semicolon-delimited list of boolean expressions, each of which is evaluated on the specified sample events.

sequence_concatenation::=

[repeat_range] sequence_expr { **;** [repeat_range] sequence_expr }

repeat_range ::=

[*constant*_expression [**:** *constant*_expression] **]** // *constant*_expression must be 0 or greater

[*constant*_expression **: inf]**

Thus, the sequence

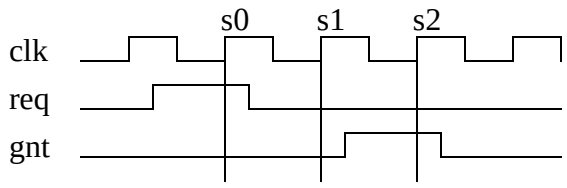
```
req;[1]gnt;[1]!req
```

specifies that **req** be true on the current sample, **gnt** will be true on the first subsequent sample and **req** will be false on the next sample after that. The ‘;’ operator specifies the separation of sampling events. The number of samples is prepended to the expression in the sequence, as in

```
req;[2]gnt
```

This specifies that req will be true on the current sample, and gnt will be true on the second subsequence sample, as shown in figure Figure 11-3.

Figure 11-3—Concatenation



To specify that ‘b’ will be true on the Nth sample after ‘a’, the following two sequences are equivalent:

```
a;[N]b      // check b on the Nth sample
```

To specify concatenation of overlapped sequences, where the end point of one sequence coincides with the start of the next sequence, a value of 0 samples is used as shown below.

```
a ;[1]b ;[1]c // first sequence seq1
d ;[1]e ;[1]f // second sequence seq2
seq1 ;[0] seq2 // overlapped concatenation
```

In the above example, c is the endpoint of sequence seq1, and d is the start of sequence seq2. When concatenated with [0] sampling , c and d must occur at the same time, resulting in the concatenated sequence being is equivalent to:

```
a;[1]b ;[1]c&&d ;[1]e ;[1]f
```

In cases where the concatenation can occur anytime between two points in time, a time window can be specified as

```
req;[4:32] gnt
```

In the above case, signal gnt must be true at some sampling event between sampling events ranging from 4 to 32 after the current sample.

The time window can extend to the end of simulation in the example below.

```
req;[4:inf] gnt
```

Keyword **inf** is used to indicate the end of simulation. For formal verification tools, **inf** is interpreted as infinity.

A sequence can be unconditionally extended by using **true**. **true** unconditionally evalues to true boolean value.

```
a ;[1]b ;[1]c ;[3]true
```

After signal c, the signal length is extended by 3 sample events. Such adjustments in the length of sequences are required when complex sequences constructed by combining simpler sequences.

11.6 Declaring Sequences

Sequences can be reused by declaring them as objects of type **seq** with optional parameters:

named_seq ::=
name [**(**formal_args**)**] = **(**sequence_expr**)**
seq_decl ::= **seq** @**(** event_expression **)** named_seq [{ **,** named_seq}] **;**
| **seq** @**(** event_identifier **)** named_seq [{ **,** named_seq}] **;**

The event_expression or event_identifier specifies the clock for the sequence.

The declaration can optionally include arguments that allow the same sequence to be instantiated multiple times with different argument values.

Note that variables referenced within a seq that are not formal arguments to the sequence are resolved hierarchically from the scope in which the seq is instantiated.

```
event clkev = (posedge clk);
seq @clkev s1 = (a;b;c), s2 = (d;e;f);
seq @(nededge clk) s3 = (g;h;i);
```

In this example, sequences s1 and s2 are sampled on each successive posedge clk. The sequence s3 is sampled on negedge clk.

11.7 Sequence Operations

11.7.1 Repetition in Sequences

To specify the repetition of a boolean expression within a sequence, the boolean may simply be repeated, as:

```
a;[1]b;[1]b;[1]b;[1]c
```

or the number of repetitions may be specified with a trailing “*[N]”, as:

```
a;b*[3];c
```

sequence_repeat ::=

sequence_expr ***** [repeat_range]

| sequence_expr **=*** repeat_range

If it can be repeated a minimum or maximum number of times, this can be expressed with a trailing *[min:max]. These repetition counts must also be literals or constant expressions.

```
(a ;[1] b)*[5] // a;[1]b;[1]a;[1]b;[1]a;[1]b;[1]a;[1]b;[1]a;[1]b
(a*[0:3];[1]b;[1]c) // equivalent to(b;[1]c) or (a;[1]b;[1]c) or
                    // (a;[1]a;[1]b;[1]c) or (a;[1]a;[1]a;[1]b;[1]c).
```

This means that a sequence `a;[1]a&&b;[1]a;[1]b;[1]c;` will pass. However, the expression sequence is not equivalent to `((a && !b)* [0:3];[1]b;[1]c)`, which would fail the same sequence.

In addition, the keyword **inf** is introduced to specify a potentially infinite maximum number of repetitions. So,

```
a;[1] b*[1:inf];[1]c
```

means ‘a’ is true on the current sample, then ‘b’ will be true on every subsequent sample until ‘c’ is true. On the sample in which ‘c’ is true, ‘b’ does not have to be true.

The “*[N]” notation indicates consecutive repetition of an expression. It is also possible to specify non-consecutive repetition of a *boolean* expression with

```
a;b*=[min:max];c
```

This is equivalent to

```
a;[1]((!b*[0:inf];b))*[min:max]);[1]c
```

Adding the range specification to this allows the construction of useful sequences containing a boolean expression that is true for at most N samples:

```
a;[1]b*=[0:N];[1]c // a followed by at most N occurrences of b, followed by c
```

The rules for specifying repeat counts are summarized as:

— Each form of repeat count specifies a minimum and maximum number of occurrences

— expr*[n:m], where n is the minimum, m is the maximum

— expr*[n] is the same as expr*[n:n]

— The sum of the minimum repeat counts for all terms in a sequence must be greater than 0

— The sequence as a whole cannot be empty

— If n is 0, then there must be either a prefix, or a post fix concatenation term

The delay and repetition syntax for sequences is summarized in the following table:

Table 11-1Sequence Concatenation and Repetition Summary

Expression	Meaning	Comment
a;[1]b	'a' followed by 'b' on the next clock tick	
a;[N]b	'a' followed by 'b' on the Nth clock tick	N may be 0 or greater
a;[N:M]b	'a' followed by 'b' on any of the Nth to Mth clock ticks	
a;[1]b*[N];[1]c	'a' followed by N consecutive samples of 'b', followed immediately by 'c'	
a;[1]b*[N:M];[1]c	'a' followed by at least N and at most M consecutive samples of 'b', followed immediately by 'c'.	
a;[1]b*=[N];[1]c	'a' followed by N non-consecutive samples of 'b', followed immediately by 'c'.	
a;[1]b*=[N:M];[1]c	'a' followed by at least N and most M non-consecutive samples of 'b', followed immediately by 'c'.	
a;[1]b*[1:inf];[1]c	'a' followed by an infinite number of samples of 'b', followed by 'c'.	Only fails in simulation if 'b' does not occur after 'a'.

Note that the ';' operator has higher precedence than the repetition operators, so

```
a;[1]b*[2]
```

matches a;b;b, whereas

```
(a;[1]b)*[2]
```

matches a;b;a;b. Also note that the repetition operators associate left-to-right, so

```
a;[1] (b*[2] *= [3]) ;[1]c
```

is equivalent to

```
(((a;[1]b)*[2])*=[3]);[1]c
```

11.7.2 AND operation

The binary operator **and** is used when both operand expressions are expected to succeed, but the end times of the operand expressions may be different.

sequence_and ::=
sequence_expr **and** sequence_expr

The two operands of **and** are sequence expressions. The requirement for the success of the **and** operation is that both the operand expressions must succeed. When one of the operand expressions succeeds, it waits for the other to succeed. The end time of the composite expression is the end time of the operand expression that completes last.

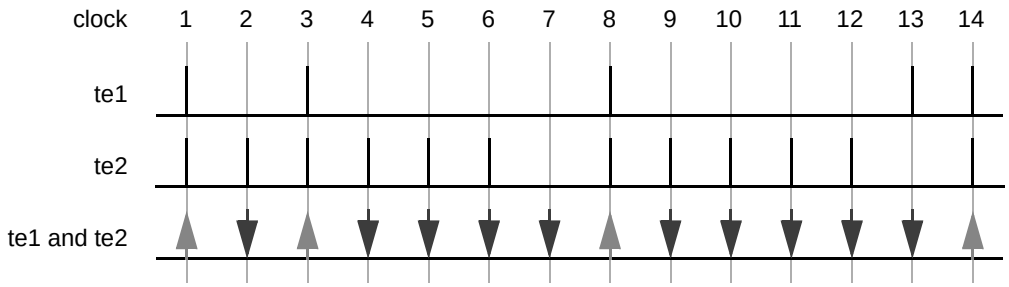
For the expression:

```
te1 and te2
```

If `te1` and `te2` are sampled booleans (not sequences), the expression succeeds if `te1` and `te2` are both evaluated to be true.

An example is illustrated in Figure 11-4 to show the results for attempt at every clock tick. The expression matches at clock tick 1, 3 and 8 because both `te1` and `te2` are simultaneously true. At all other clock ticks, the **and** operation fails because either `te1` or `te2` is false.

Figure 11-4—ANDing (and) Two Sequences



When `te1` and `te2` are sequences, then the expression:

```
te1 and te2
```

— Succeeds if `te1` and `te2` succeed.

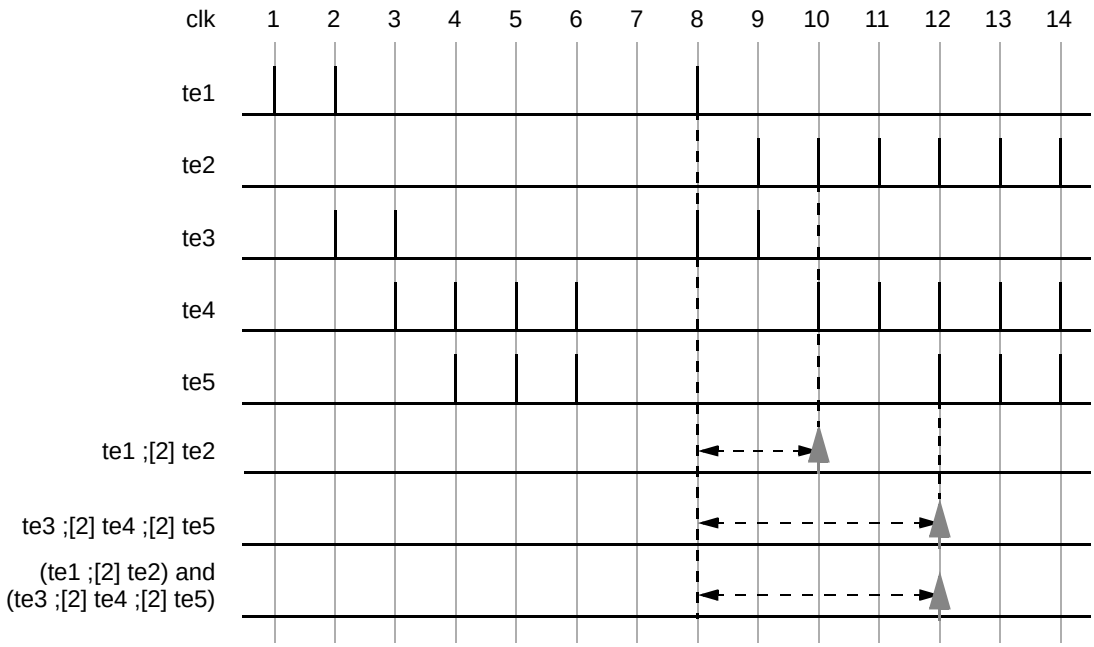
— The end time is the end time of either `te1` or `te2`, whichever terminates last.

First, let us consider the case when both operands are single sequence evaluations.

An example is illustrated in Figure 11-5. Consider the following expression with operator ***and*** where the two operands are sequences.

```
(te1 ;[2] te2) and (te3 ;[2] te4 ;[2] te5)
```

Figure 11-5—ANDing (and) Two Sequences



Here, the two operand sequences are `(te1 ;[2] te2)` and `(te3 ;[2] te4 ;[2] te5)`. The first operand sequence requires that first `te1` evaluates to true followed by `te2` two clock ticks later. The second sequence requires that first `te3` evaluates to true followed by `te4` two clock ticks later, followed by `te5` two clock ticks later. Figure 11-5 shows the evaluation attempt at clock tick 8.

This attempt results in a match since both operand sequences match. The end times of matches for the individual sequences are clock ticks 10 and 12. The end time for the entire expression is the last of the two end times, so a match is recognized for the expression at clock tick 12.

Now, consider an example where an operand sequence is associated with a range of time specification, such as:

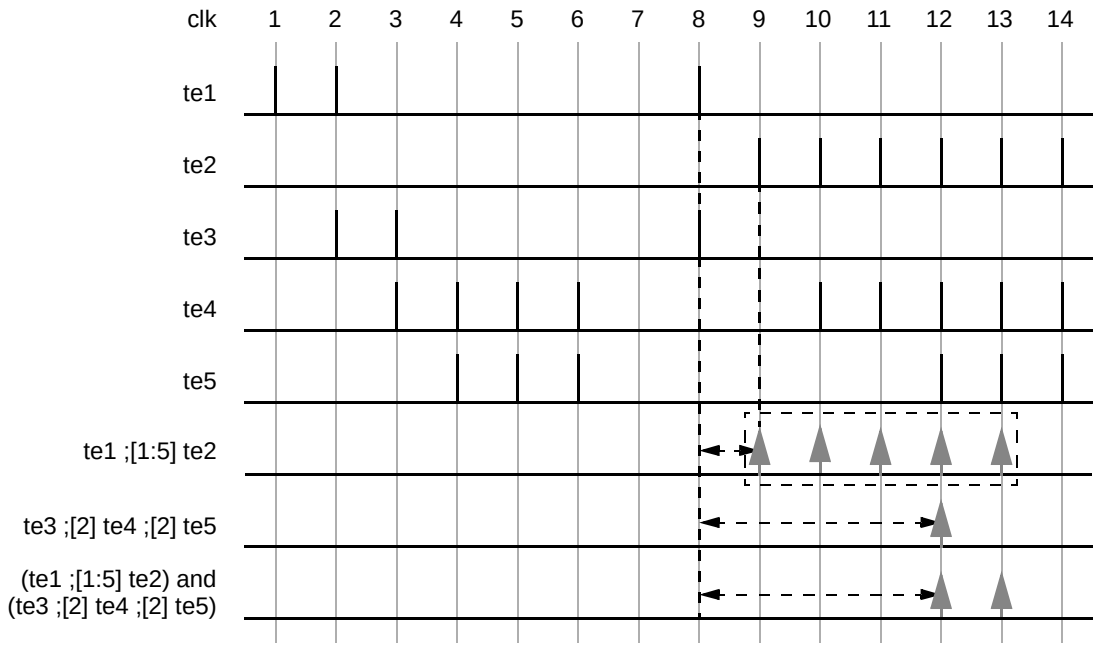
```
(te1 ;[1:5] te2) and (te3 ;[2] te4 ;[2] te5)
```

The first operand sequence consists of an expression with a time range from 1 to 5 and implies that when `te1` evaluates to true, `te2` must follow 1, 2, 3, 4, or 5 clock ticks later. The second operand sequence is the same as in the previous example. To consider all possibilities of a match, following steps are taken:

- The first operand sequence starts five sequences of evaluation.
- The second operand sequence has only one possibility of match, so only one sequence is started.
- Figure 11-6 shows the attempt to examine at clock tick 8 when both operand sequences start and succeed. All five sequences for the first operand sequence match, as shown in a time window, at clock ticks 9, 10, 11, 12 and 13 respectively. The second operand sequence matches at clock tick 12.
- To compute the result for the composite expression, each successful sequence from the first operand sequence is matched against the second operand sequence according to the rules of the **and** operation to determine the end time for each match.

The result of this computation is five successes, four of them ending at clock ticks 12, and the fifth ends at clock tick 13. Figure 11-6 shows the two unique successes at clock ticks 12 and 13.

Figure 11-6—ANDing (and) Two Sequences Including a Time Range



11.7.3 Intersection (AND with length restriction)

The binary operator **intersect** is used when both operand expressions are expected to succeed, and the end times of the operand expressions must be the same.

sequence_intersect ::=

sequence_expr **intersect** sequence_expr

The two operands of **intersect** are sequence expressions. The requirements for the success of the ***intersect*** operation are:

- Both the operand expressions must succeed.
- The length of the two operand sequences must be the same.

The additional requirement on the length of the sequences is the basic difference between **and** and **intersect.**

When there are multiple matches for each operand sequence expression, the results are computed as follows.

- A match from the first operand is paired with a match from the second operand with the same length (end time).
- If no such pair is found, the result of **intersect** is no match.
- If such pairs are found, then the result consists of matched sequences, one for each pair. The end time of each match is determined by the pair.

11.7.4 OR operation

The operator **or** is used when at least one of the two operand sequences is expected to match.

sequence_or ::=

sequence_expr **or** sequence_expr

The two operands of **or** are sequence expressions.

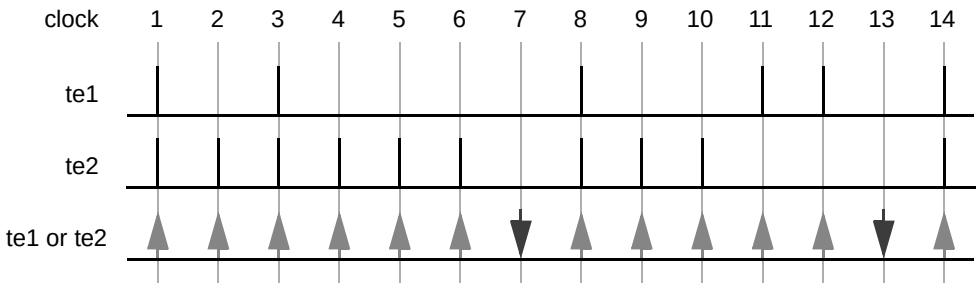
Let us consider these operand expressions as values, events and sequences separately to illustrate the details of **or** operations. For the expression

```
te1 or te2
```

when the operand expressions `te1` and `te2` are events or values, the expression matches whenever at least one of two operands `te1` and `te2` is evaluated to true.

Figure 11-7 illustrates **or** operation using `te1` and `te2` as simple values. The expression does not match at clock ticks 7 and 13 because `te1` and `te2` are both false at those times. At all other times, the expression matches, as at least one of the two operands is true.

Figure 11-7—ORing (or) Two Sequences



When `te1` and `te2` are sequences, then the expression

```
te1 or te2
```

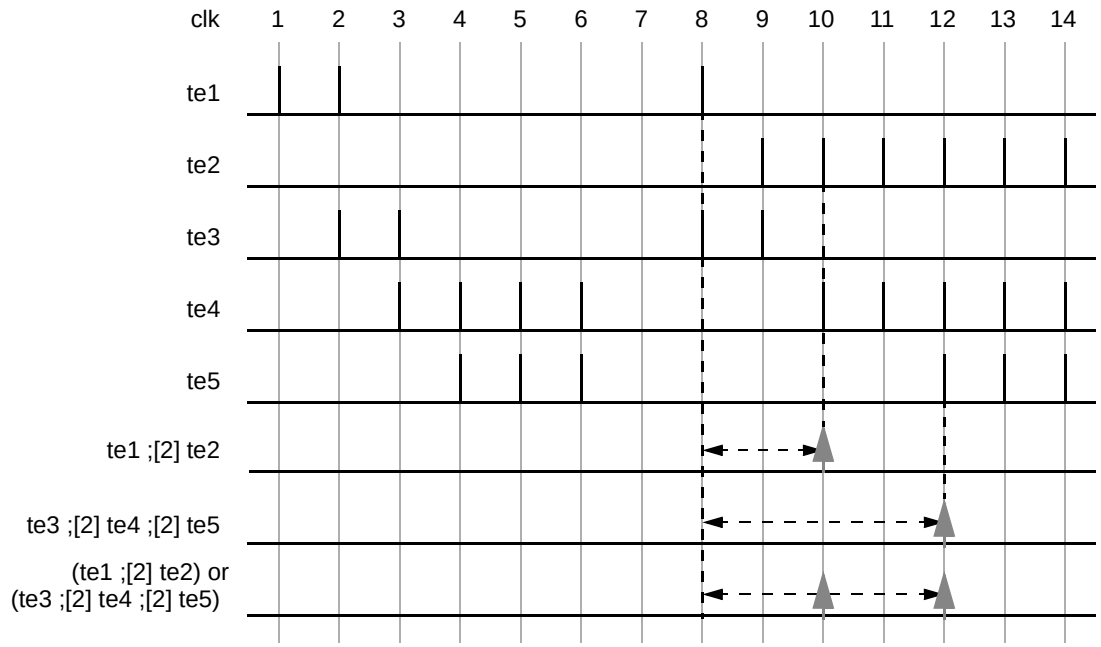
matches if at least one of the two operand sequences `te1` and `te2` match. To evaluate this expression, first,

the successfully matched sequences of each operand are calculated and assigned to a group. Then, the union of the two groups is computed. The result of the union provides the result of the expression. The end time of a match is the end time of any sequence that matched.

An example is illustrated in Figure 11-8. Consider an expression with **or** operator where the two operands are sequences.

```
(te1 ;[2] te2) or (te3 ;[2] te4 ;[2] te5)
```

Figure 11-8—ORing (or) Two Sequences



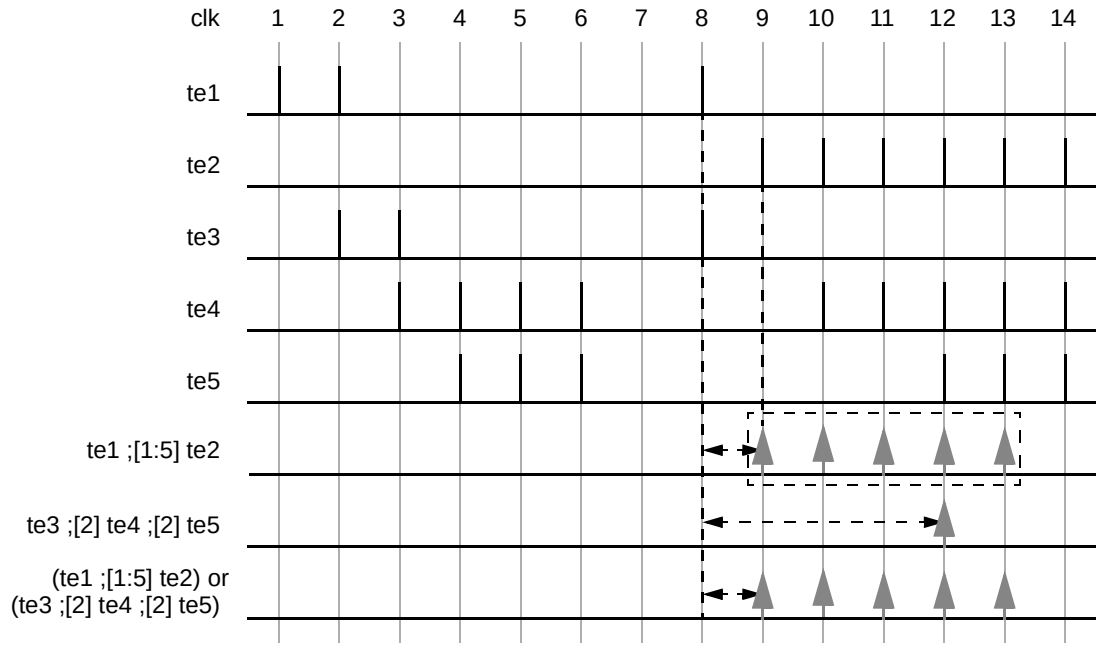
Here, the two operand sequences are: `(te1 ;[2] te2)` and `(te3 ;[2] te4 ;[2] te5)`. The first sequence requires that `te1` first evaluates to true, followed by `te2` two clock ticks later. The second sequence requires that `te3` evaluates to true, followed by `te4` two clock ticks later, followed by `te5` two clock ticks later. In Figure 11-8, the evaluation attempt for clock tick 8 is shown. The first sequence matches at clock tick 10 and the second sequence matches at clock tick 12. So, two matches for the expression are recognized.

Consider an example where an operand sequence is associated with time range specification, such as:

```
(te1 ;[1:5] te2) or (te3 ;[2] te4 ;[2] te5)
```

The first operand sequence consists of an expression with a time range from 1 to 5 and specifies that when `te1` evaluates to true, `te2` must be true 1, 2, 3, 4 or 5 clock ticks later. The sequences from the second operand require that first `te3` must be true followed by `te4` being true two clock ticks later, followed by `te5` being true two clock ticks later. At any clock tick if an operand sequence succeeds, then the composite expressions succeeds. As shown in Figure 11-9, for the attempt at clock tick 8, the first operand sequence matches at clock ticks 9, 10, 11, 12, and 13, while the second operand matches at clock ticks 12. The match of the composite expression is computed as a union of the matches of the two operand sequences, which results in matches at clock ticks 9, 10, 11, 12, and 13.

Figure 11-9—ORing (or) Two Sequences Including a Time Range



11.7.5 first_match operation

Use the **first_match** operator when you are interested in only the first match from a sequence expression that can possibly result in multiple matches. This allows you to discard all subsequent matches from consideration. In particular, when the sequence expression is a sub-expression of a larger expression, then applying the **first_match** operator has significant effect on the evaluation of the embedding expression.

sequence_first_match ::=
first_match (sequence_expr)

The operand expression can be a sequence expression. `sequence_expr` is evaluated to determine the match for the (**first_match** (`sequence_expr`)) expression. For a given evaluation attempt, the composite expression matches if `sequence_expr` results in at least one match of a sequence, and fails to match if none of the sequences from the expression result in a match. Following the first successful match for the attempt, the **first_match** operator stops matching subsequent sequences for `sequence_expr`. For an attempt, if there are multiple matches with the same end time as the first detected match, then all those matches are considered as the result of the expression.

Please note that **first_match** applies to each attempt for the sequence individually.

Consider an example with a variable delay specification as shown below.

```
seq t1 = (te1 ;[2:5]te2);
seq ts1 = (first_match(te1 ;[2:5]te2));
```

Each attempt of sequence t1 can result in matches for up to four following sequences:

```
te1 ;[2] te2
te1 ;[3] te2
te1 ;[4] te2
te1 ;[5] te2
```

However, sequence ts1 can result in a match for only one of the above four sequences. Whichever of the above four sequences matches first becomes the result of sequence ts1.

11.7.6 Conditional sequences

These constructs allow a user to monitor sequences based on satisfying some criteria. Most common uses are to attach a precondition to a sequence, and to select a sequence between two alternatives, where the selection is made based on the success of a condition.

Two kinds of clauses are provided:

if_sequence::=

if (expression**)** sequence_expr

This clause is used to precondition monitoring of a sequence expression. The condition `boolean_cond` must be satisfied in order to monitor `sequence_expr`. If the condition `boolean_cond` fails then `sequence_expr` is skipped for monitoring. `expression` is a logical expression that results in true or false, and `sequence_expr` is a sequence expression that can result in one or match sequence matches. *If the expression evaluates to true, then the first element of the sequence_expr is evaluated on the same clock tick.*

If the condition is evaluated to true, then the evaluation of `sequence_expr` is conducted. The sequence matches of `sequence_expr` become the matches of the clause **if**.

if_else_sequence::=

if (expression**)** sequence_expr1 **else** sequence_expr2

This clause is used to select a sequence expression between two alternatives. If `boolean_cond` evaluates true, then `sequence_expr1` is monitored. If `boolean_cond` is false, then `sequence_expr2` is selected for monitoring. The expression `expression` is logical and must result in true or false. `sequence_expr1` and `sequence_expr2` can be sequence expressions. The match of clause ***if-else*** depends on the match of the sequence expression, `sequence_expr1` or `sequence_expr2`, whichever gets selected for monitoring.

Clauses **if** and **if-else** can be nested to contain another conditional sequence within it, such as:

```
if (!reset)
   if (data_phase) ([0:7] data_end);
```

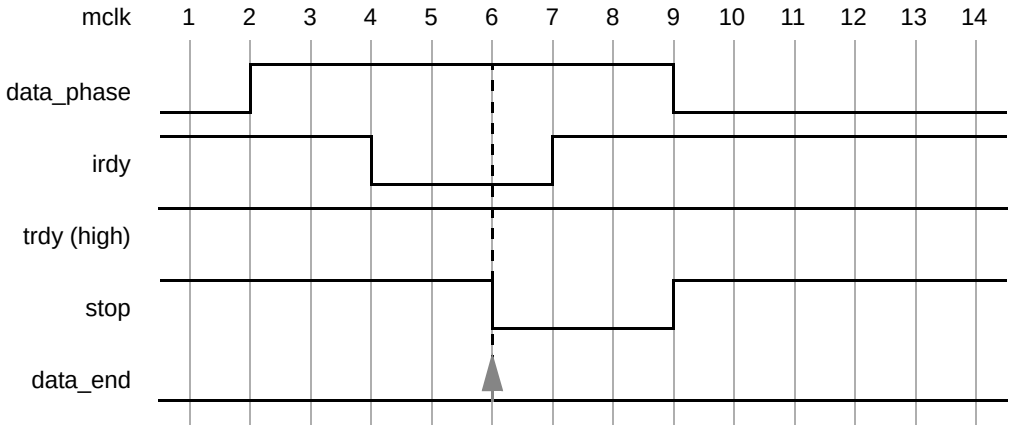
When `(!reset)` is true, then the second **if** condition `data_phase` is tested. If `data_phase` evaluates to true, then the evaluation continues for the expression (`[0:7] data_end`).

The semantics of **if-else** and **if** specifications is next illustrated by examples. Consider a bus operation for data transfer from a master to a target device. When the bus enters a data transfer phase, multiple data phases can occur to transfer a block of data. During the data transfer phase, a data phase completes on any rising clock edge on which `irdy` is asserted and either `trdy` or `stop` is asserted. Note that an asserted signal here implies a value of low. The end of a data phase can be expressed as:

```
seq @(posedge mclk) data_end =
                    ( if (data_phase) ((irdy==0)&&(fell trdy || fell stop)));
```

Each time a data phase completes, a match for `data_end` is recognized. The attempt at clock tick 6 is illustrated in Figure 11-10. The values shown for the signals are the sampled values with respect to the clock. At clock tick 6 `data_end` is matched because `stop` gets asserted while `irdy` is asserted.

Figure 11-10—Conditional Sequence Matching



`data_end` can be used to ensure that `frame` is de-asserted within 2 clock ticks after `data_end` occurs. Further, it is also required that `irdy` gets de-asserted one clock tick after `frame` gets de-asserted.

A sequence expression is written to express this condition as shown below.

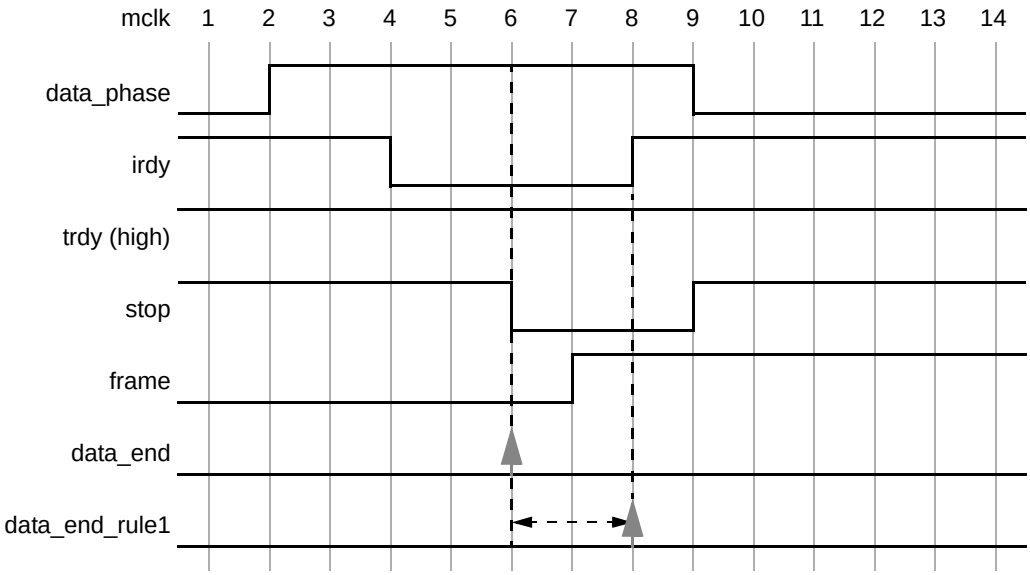
```
seq @(posedge mclk)
   data_end = ( data_phase &&((irdy==0) && (fell trdy || fell stop)) ),
   data_end_rule1 =( if (ended data_end1) ([1:2] rose frame ; rose irdy) );
```

`seq data_end_rule1` first evaluates `data_end` at every clock tick to test if its value is true. If the value is false, then that particular attempt to check the assertion is considered a success. Otherwise, the sequence expression following the **if** clause is monitored. The sequence expression

```
[1:2] rose frame ; rose irdy
```

specifies looking for the rising edge of `frame` within two clock ticks in the future. After `frame` toggles high, `irdy` must also toggle high after one clock tick. This is illustrated in Figure 11-11. Sequence `data_end` is acknowledged at clock tick 6. Next, `frame` toggles high at clock tick 7. Since this falls within the timing constraint imposed by `[1:2]`, it satisfies the sequence and continues to monitor further. At clock tick 8, `irdy` is evaluated. Signal `irdy` transitions to high at clock tick 8, satisfying the sequence specification completely for the attempt that began at clock tick 6.

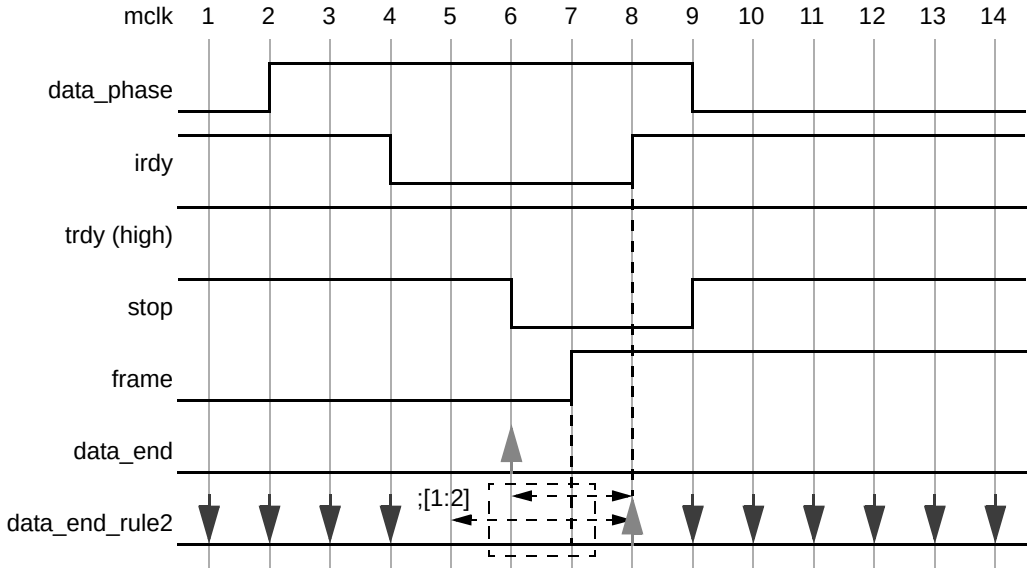
Figure 11-11—Nested Conditional Sequences



Generally, assertions are associated with preconditions so that the checking is performed only under certain specified conditions. As seen from the previous example, the **if** clause provides this capability to specify pre-conditions with sequences that must be satisfied before continuing to match those sequences. Let us modify the above example to see the effect on the results of the assertion by removing the precondition for the sequence. This is shown below and illustrated in Figure 11-12.

```
seq @(posedge mclk) data_end_rule2 = ( [1:2] rose frame ; [1] rose irdy );
```

Figure 11-12—Results without the Condition



The sequence is evaluated at every clock tick. For the evaluation at clock tick 1, the rising edge of signal `frame` does not occur at clock tick 1 or 2, so the evaluation fails and the result for the sequence is a failed match at clock tick 1. Similarly, there is a failure at clock ticks 2, 3, and 4. For attempts starting at clock ticks 5 and 6, the rising edge of signal `frame` at clock tick 7 allows checking further. At clock tick 8, the sequences

complete according to the specification, resulting in a match for attempts starting at 5 and 6. All later attempts to match the sequence fail because `rose frame` does not occur again. That also means that there is no match at 5, 6 and 7.

As one can see from Figure 11-12, removing the precondition of checking event `data_end` from the assertion causes failures that are not relevant to the verification objective. It becomes important from the validation standpoint to determine these preconditions and use them in the assertion to filter out inappropriate or extraneous situations.

11.7.7 Conditions over sequences

Sequences of events often occur under the assumptions of some conditions for correct behavior. A logical condition must hold true, for instance, while processing a transaction. Also frequently, occurrence of certain events is prohibited while processing a transaction. Such situations can be expressed directly using the following construct:

istrue_within_sequence::=

istrue expression **within** sequence_expr

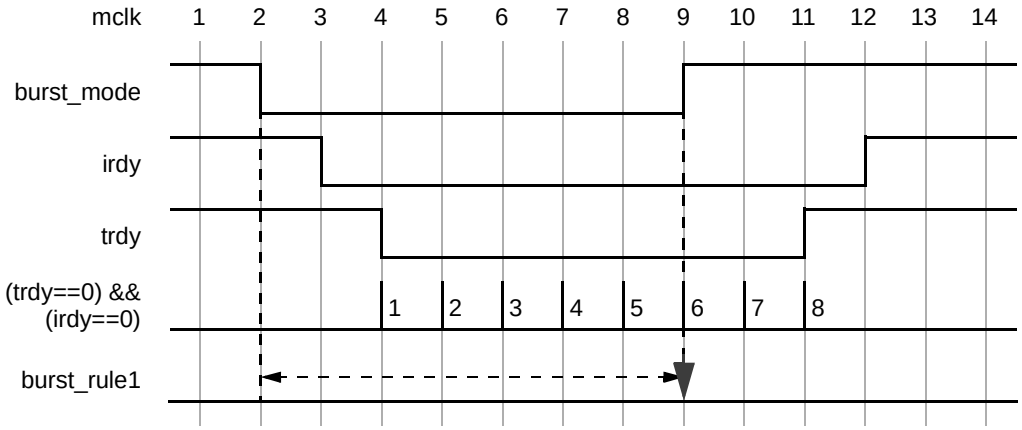
`expression` is an expression which must evaluate true at every clock tick while monitoring `sequence_expr`. If a sequence for `sequence_expr` starts at time t1 and ends at time t2, then `expression` must hold true from time t1 to t2. If either the sequence expression does not match or the boolean expression becomes false while the sequence is being evaluated, the composite sequence does not match and a property stated over this composite sequence would declare a failure.

The **`istrue`** construct is an abbreviation for writing
`(boolean_expr) *[0:inf]` intersect *sequence_expr*

Consider the example illustrated in Figure 11-13. If an additional constraint were placed on the expression as shown below, then the checker `burst_rule` would fail at clock tick 9.

```
seq @(posedge mclk) burst1 = ( if (fell burst_mode)
            istrue (!burst_mode) within  ([2] ((trdy==0)&&(irdy==0)) * [7]) );
burst_rule1: property(burst1);
```

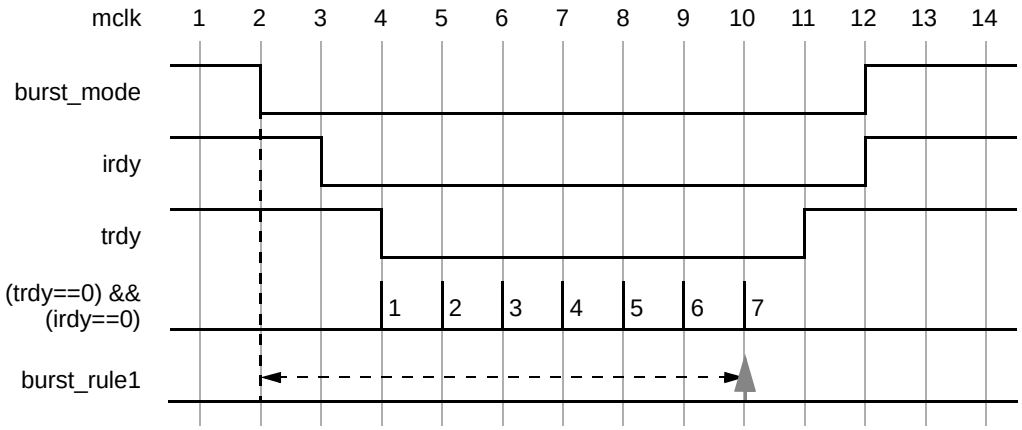
Figure 11-13—Match with istrue-within Restriction Fails



In the above expression, the value of signal `burst_mode` is required to be low during the sequence (from clock tick 2 to 11), and is checked at every clock tick during that period. At clock ticks from 2 to 8, signal `burst_mode` remains low and matches the expression at those clock ticks. At clock tick 9, signal `burst_mode` becomes high, thereby failing to match the expression for `burst_rule1.`

If signal `burst_mode` were to be maintained low until clock tick 11, the expression would result in a match as shown in Figure 11-14.

Figure 11-14—Match with istrue-within Restriction Succeeds

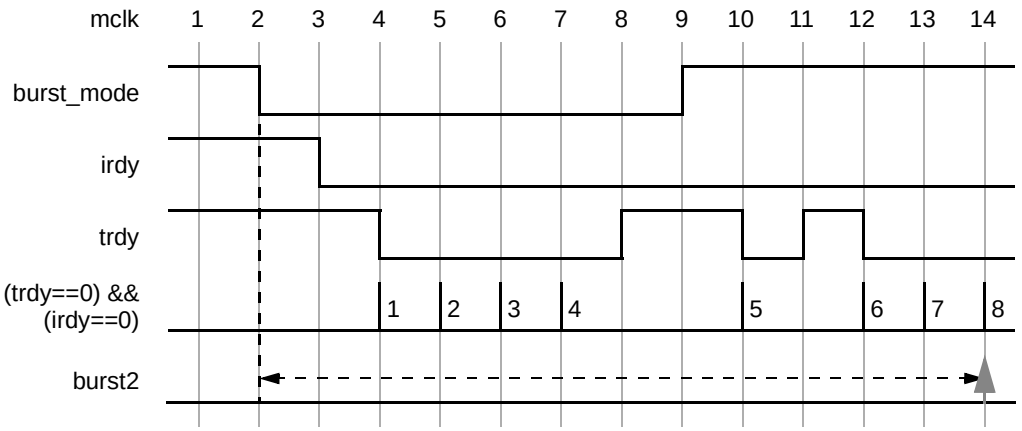


Let us consider a modified version of the example in Figure 11-14 as shown below.

```
seq @(posedge mclk) burst2 = ( if (fell burst_mode)
                                 ([0:4] (trdy==0) && (irdy==0)) * [8] );
```

The sequence `burst2` has been relaxed to require each repetition of the sequence to occur between 1 and 4 clock ticks after the preceding occurrence of the sequence. This is illustrated in <hotlink>Figure 11-15 on page 11-59.

Figure 11-15—Match without Restriction Succeeds



Two additional clock ticks delay the fifth repetition and one additional clock tick delays the sixth repetition as signal trdy becomes high to suspend the next data phase for two clock ticks and one clock tick respectively. The expression matches at clock tick 14.

11.7.8 Sequence with length specification

Another type of restriction over sequences is the total length of the sequence in terms of the clock ticks. For example, a transaction must complete within a given period of time, no matter what variation of commands are issued in the transaction to be processed.

l

ength_within_sequence::=

length repeat_range **within** sequence_expr

`repeat_range` specifies the length of the *sequence_expression*. The length is measured as the total number of clock ticks during the sequence. All variations of the `repeat_range` specification are allowed. If a single number is specified for `repeat_range`, then it represents fixed length. In other words, the sequence expression must end with a match at a specific clock tick that is determined by the `repeat_range` number. If `repeat_range` specifies a range of numbers, then the sequence_expression must end with a match anytime within a time period determined by the minimum and maximum number of clock ticks.

The **length** construct is an abbreviation for writing

```
1*[int] intersect sequence_expr
```

or

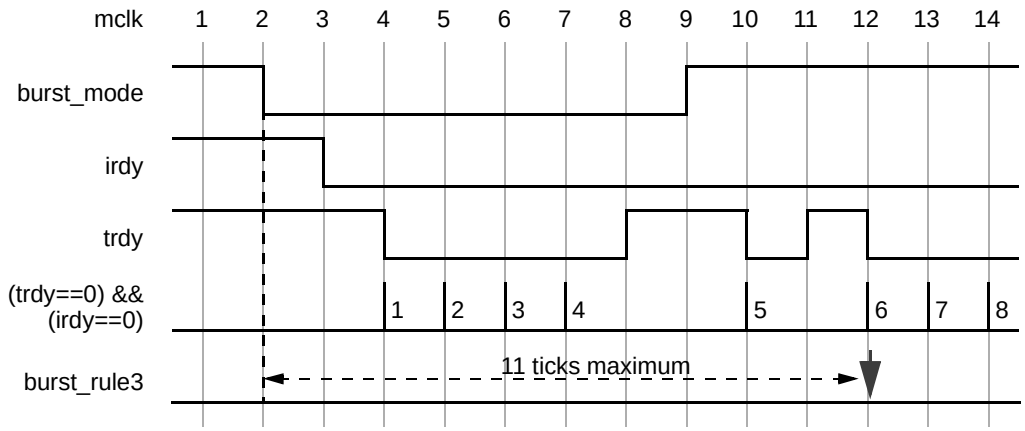
```
1*[int_interval] intersect sequence_expr
```

If an additional constraint were placed on the expression as shown below, then the expression for checker `burst_rule3` would not match at clock tick 12.

```
seq @(posedge mclk) burst3 = ( if (fell burst_mode)
                length [9:11] within (;[1:4] ((trdy==0)&&(irdy==0))*[8]) );
burst_rule3: property(burst3);
```

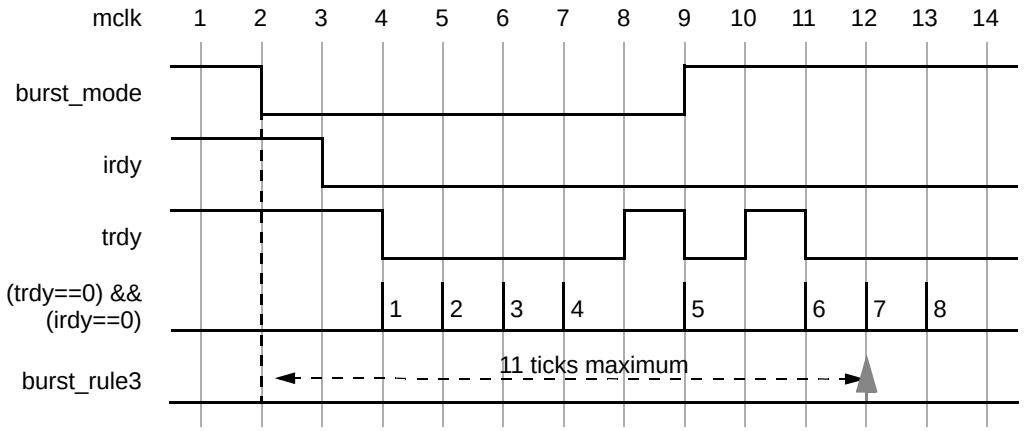
In the above expression, the total length of the entire repeated sequence must not be less than 9 clock ticks and not greater than 11 clock ticks. This restriction is expressed by `length [9:11]` in the expression. From <hotlink>Figure 11-16 on page 11-60, the corresponding time to complete all repetitions is 13 clock ticks which exceeds the maximum allowed length, so the expression fails to match at clock tick 12.

Figure 11-16—Match with length-within Restriction Fails



The failure is corrected by reducing the delay for the fifth repetition from 2 clock ticks to 1 clock tick. This is shown in Figure 11-17.

Figure 11-17—Match with length-within Restriction Succeeds



To express the constraints of a condition and a time period on the same sequence, the two constraint clauses are specified separated with a comma as shown below.

```
seq @(posedge mclk) burst4 = ( if (fell burst_mode)
      (istrue(!burst_mode), length [9:11]) within
      (;[1:4] ((trdy==0)&&(irdy==0))*[8]) );
burst_rule4: property(burst4);
```

Now the two constraints are:

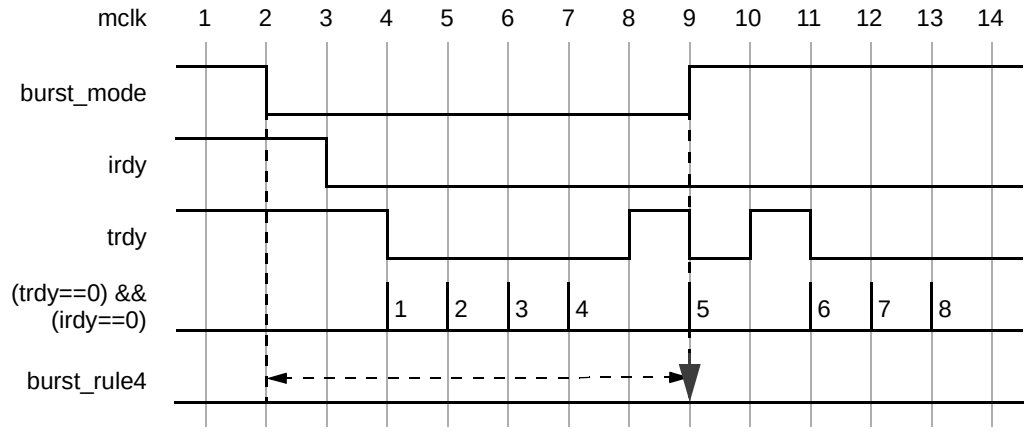
`istrue(!burst_mode)` to ensure that signal `burst_mode` remains low

`length [9:11]` to ensure that the sequence takes at least 9 clock ticks and completes in at most 11 clock ticks

Both constrains must hold for the assertion to succeed, i.e, signal `burst_mode` must remain low throughout the allowed period for the sequence.

The expression for `burst_rule4` fails to match in <hotlink>Figure 11-18 on page 11-61 because signal `burst_mode` becomes high at clock tick 9.

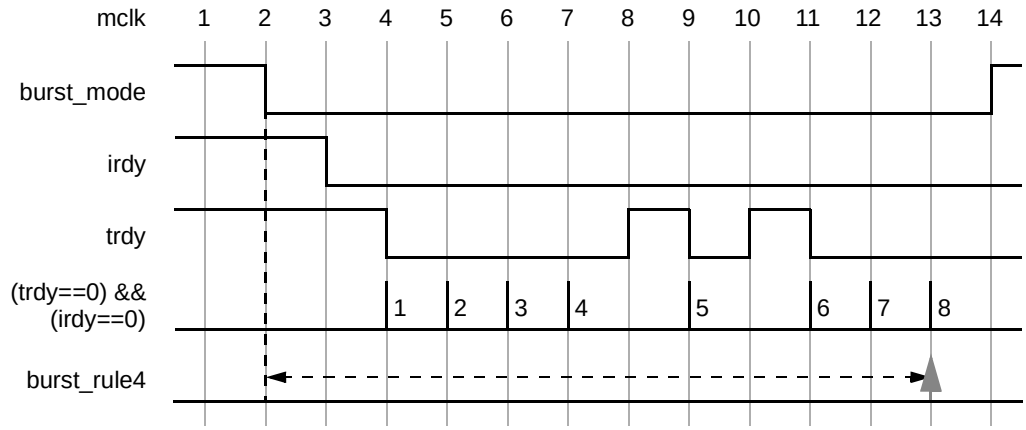
Figure 11-18—Match with Two Restrictions Fails



The failure is corrected by maintaining signal `burst_mode` to low value throughout the sequence and by

allowing the length to be in the range of 9 to 12 as shown in <hotlink>Figure 11-19 on page 11-62. The expression matches at clock tick 13 as it satisfies both constraints on the sequence: the total time period for the sequence is 12 clock ticks that is within the time period requirement, and signal `burst_mode` is held low throughout these 12 clock ticks.

Figure 11-19—Match with Two Restrictions Succeeds



11.7.9 Sequence occurrence within another sequence

The containment of a sequence expression within another sequence is expressed as

sequence_within ::=

occurs sequence_expr1 **within** sequence_expr2

The sequence sequence_*expr1* must occur entirely within the sequence *sequence_expr2*.

That is sequence_*expr1* must satisfy the following:

- The start point of sequence_*expr1* must be between the start point and the end point of *sequence_expr2*
- The end point of sequence_*expr1* must be between the start point and the end point of *sequence_expr2*

11.7.10 Detecting and using endpoint of a sequence

There are two ways in which a complex **seq** can be decomposed into simpler sub-expressions.

To use **seq** as a sub-expression, or a part of the expression is by simply referencing its name. The evaluation of a sequence expression that references a **seq** expression is performed the same way as if the **seq** expression was a lexical part of the expression. In other words, the sequence expression is "invoked" from the expression where it is referenced. An example is shown below:

```
seq @(rose sysclk) s = (a;[1] b;[1] c),
        rule = (if (trans) (start_trans;[1] s;[1] end_trans));
This is equivalent to:
    seq @(rose sysclk)
            s = (a;[1] b;[1] c),
             rule = (if (trans) (start_trans;[1] a;[1] b;[1] c;[1] end_trans)) ;
```

Another way to use the **seq** expression is to detect its end point in another sequence. The end point of a sequence is reached whenever there is a match on its expression. The occurrence of the end point can be tested in any sequence expression by using the clause **ended**. An example is shown below:

```
seq @(posedge sysclk) e1 = (rose ready;[1] proc1;[1] proc2),
                      rule = (if (reset) (inst;[1] ended e1;[1] branch_back));
```

In this example sequence expression e1 must end successfully one clock tick after inst. If the keyword **ended** wasn't there, sequence expression e1 starts one clock tick after inst.

11.8 Declaring Booleans

Because sequences are composed of boolean expressions, it is useful to allow boolean expressions to be declared as objects of type **bool**.

bool_declaration ::= **bool** named_bool { **,** named_bool } **;**

named_bool ::= identifier [**(** identifier { **,** identifier } **)**] **=** expression

The boolean object can then be declared as:

```
bool b1(a,b) = a && b && c;
```

and used in a sequence as:

```
(b1(foo,bar);[1]c;[1]d)
(b1(.a(f1),.b(b1));[1]c;[1]d)
```

Note that, in the boolean expression b1, the formal arguments 'a' and 'b' are replaced by the corresponding actual arguments when the bool is instantiated. Any variables referenced within the bool that are not formal arguments get resolved via standard rules from the scope in which the bool is instantiated.

11.9 Manipulating Data in a Sequence

NOTE: This next bit should be in the "Manipulating data" section.

The use of System Verilog variables implies that only one copy exists. Therefore, if data values need to be checked in pipelined designs, then for each data entering the pipeline we may need a separate variable to store the predicted output of the pipeline for later comparison when the result actually exits the pipe. We can build such a storage by using an array of variables arranged in a shift register to mimic the data propagating through a pipeline. However, in more complex situations where the latency of the pipe is variable and out of order, this construction could become very complex and error prone. In other words, we need variables that are local to and are used within a particular transaction check which can span an arbitrary interval of time and may overlap with other transaction checks. Such a variable must thus be dynamically created when needed within an instance of a sequence and removed when the end of the sequence is reached.

The dynamic variable creation and destruction can be achieved using the following clause:

let_sequence::=

let (type identifier = expression) **within** sequence_expr;

The type of name is explicitly specified. The value of the expression is sampled at the time of the beginning of sequence_expression and stored in the dynamically created variable var_name. Inside sequence_expression, the value of the variable remains unchanged for the entire duration of the sequence. Variable var_name can be used in sequence_expression as any other variable. For every attempt, a new instance of variable var_name is created for the sequence_expression.

For example, assume a pipeline that has a fixed latency of 5 clock cycles. The data enters the pipe on pipe_in when valid_in is true and the value computed by the pipeline appears 5 clock cycles later on the signal pipe_out1. The data as transformed by the pipe is predicted by a function that increments the data. The following sequence expression verifies this behavior.

```
seq e = ( if (valid_in)
            let (int x = pipe_in) within ;[5] (pipe_out1 == (x+1)) );
```

Suppose now that the output of this pipe is chained to another pipe of latency 3 that computes the value as predicted by data and pipe_val. The transfer to the second pipe happens only if the result of the first pipe satisfies some Boolean variable pipe_cont. We can modify and extend the above example as follows:

```
seq e_two_pipes =
  ( if (valid_in) let (int x = pipe_in) within
                     (;[5] ( pipe_out1 == (x+1));
                     if (pipe_out1==pipe_cont)
                        let (int y == x+1) within
                             ;[3](pipe_out2==(y+pipe_val))) );
```

The nested **let-within** construct uses the variable name y which is assigned a value from the enclosing declared variable x.

Note that, for practical debugging and verification performance reasons, it may be preferable to verify each of the two pipelines by a separate sequence rather than in one single sequence as in the above example.

If the pipeline supported out-of-order execution in which the outputs can exit with variable latency and in a different order than the data entered, it is a simple matter to add an id to each input data and then check that data is correct when the id appears on the output. The first example modified to include the id on input and output is as follows:

```
seq e_with_id = ( if (valid_in) let (int x = pipe_in, int id = id_in) within
                                    ;[1:inf] (if (id_out == id && valid_out)
                                                 pipe_out1 == x+1) );
```

In this example, notice the use of two dynamic variables, x and id, assigned in the same **let** statement by separating them by a comma.

In addition to accessing values of signals at the time of evaluation of a boolean expression, the past values can be accessed with the $past function.

```
$past(bit_vector_expr [, number_of_ticks])
```

The argument number_of_ticks specifies the number of clock ticks in the past. If number_of_ticks is not specified, then it defaults to 1. $past returns the value of the expression bit_vector_expr that was present number_of_ticks prior to the time of evaluation of $past.

Another useful function provided for the boolean expression is $**countones**, to count the number of 1s in a bit vector expression.

```
$countones(bit_vector_expr)
```

11.10 The Property Definition

A property defines a behavior of the design. A property can be used for verification as an assumption, a checker or a coverage specification. In order to use the behavior for verification, a verification directive must be used. A property declaration by itself does not produce any result.

To declare a property, the **property** construct is used as shown below.

prop_declaration ::= **property** named_prop { **,** named_prop } **;**

named_prop ::= identifier [**(** identifier { **,** identifier } **)**] = prop_expr

prop_expr ::=
[**initial**] [reset_mode **(** expression **)**] [**not**] clocked_sequence
| identifier [**(** expression_list **)**] // identifier must be a property
reset_mode ::=
accept
| **reject**
clocked_sequence ::= [event_control] sequence_expr

A **property** declaration is parameterized, like a **seq** and **bool** declaration. When a property is instantiated, actual arguments can be passed to the property. The property gets expanded with the actual arguments by replacing the formal arguments with the actual arguments. The semantic checks are performed to ensure that the expanded property with the actual arguments is legal.

The **accept** and **reject** clauses allow you to specify asynchronous resets. For a particular attempt, if the accept boolean expression becomes true at any time during the evaluation of the attempt, then the attempt for the property is considered to be a success. Conversely, for the reject clause, if the boolean expression becomes false during an attempt, the property for that attempt results in a failure. Please note that the boolean expressions for both clauses are evaluated asynchronously with respect to the clock for the property.

The **not** clause states that the expression associated with the property must never evaluate to true. Effectively, it negates the property expression. For each attempt, clocked_sequence results in either true of false, based on whether there is a match for the sequence. The **not** clause reverses the result of clocked_sequence. It should be noted that there is no complementation or any form of negation for the sequence itself.

The **initial** clause states that the property should only be evaluated on the first clock tick. Thereafter, there should be no evaluation of the property. Without the **initial** clause the property is evaluated for every clock tick.

This allows for the following examples:

```
property rule1 = @(posedge clk) (if (a)   (b;[1]c;[1]d));
property rule2 = (accept = foo) not @(clkev) (if(a)   (b;[1]c;[1]d));
```

A property by default is not evaluated for checking the expression. A verification directive states the verification function to be performed on the property. The directive can be one of the following:

— **assert** to specify the property as a checker to ensure that the property holds for the design

— **assume** to specify the property as an assumption for the design

— **restrict** to specify the property as an assumption without any obligation to prove that it holds for the environment.

— **cover** to monitor the property evaluation for coverage

property_directive ::=
[identifier **:**] directive **(*) ;**
| [identifier **:**] directive prop_expr [{ **,** prop_expr }] **;**
| [identifier **:**] directive prop_expr [{ **,** prop_expr }] action_block **;**
directive ::=
assert
| **assume**
| **restrict**
| **cover**

When the property for a directive is evaluated to be true, the action statements associated with the **do** clause are executed. Otherwise, the statements associated with **else** clause are executed.

The **assert** directive is used to enforce a property as a checker. The tool may not consider the property as an assumption.

The **assume** directive is used when the assertion expression is applied as a constraint for the environment to the design under test. When used with model checking, the assumption must be verified with a proof that it holds for the environment.

The **restrict** directive is same as the **assume** directive, except that no proof is required by a model checker. This directive is used primarily to deal with the capacity limitations of the formal verification methods

To monitor sequences and other behavioral aspects of the design for coverage, the same syntax is used with the **cover** directive. The tools can gather information about the evaluation and report the results at the end of simulation. The pass and fail action statements can specify a coverage function, such as monitoring all paths for a sequences. For example,

```
property  abc(a,b,c) = accept(a==2) not @clk (b;[1]c);
env_prop: assume abc(rst,in1,in2) do pass_stat else fail_stat;
```

A directive can directly specify an expression, without first declaring it as a property. For example,

```
input_prop: assert accept(in1=2) not @clk (f;[1]g),
            reject(in2=1) @clk2 (m;[1]n)
            do pass_stat else fail_stat;
```

In the above example, two properties are specified with the **assert** clause. The **do** and **else** clauses are commonly specified for both the properties and will apply to the results of both properties.

Please note that a property specification can be just a bool or a sequence.
The star notation allows to apply a directive to all declared properties. It provides a short hand for individually writing the directive for each property.

```
mod_cover: cover   (*)
               do $display("property passing coverage");
                else $display("property failing coverage”);
```

The star notation will apply the coverage directive to all un-parameterized properties. Parameterized properties must be explicitly instantiated for them to be used. The scope of the star notation is the current module and the hierarchy under it. For example, if it is placed in $root, then it will apply to all properties wherever specified.

11.10.1 Declaring Properties Outside Of Procedural Code

The property block can be used directly within a module as a *module_item* or within interface as an *interface_item*. For example,

```
module top(input bit clk);
reg a,b,c;
property rule3 = @(posedge clk) (if(a) (b;[1]c));
...
endmodule
```

rule3 is a property declared in module top.

11.10.2 Embedding Properties in Procedural Code

The property block can be declared or instantiated directly in a procedural block as in:

```
always @(posedge clk) begin
   property rule = (a;[1]b;[1]c);
   <statements>;
   <statements>;
   end
```

When a property is placed in a procedural block, it potentially inherits a clock, an enabling condition and a reset from its procedural context. A procedural property is equivalent to a declarative property when the clock, reset and enabling conditions from its context are applied to the property.

For example, when a property is instantiated in a procedural block, the sample event may be specified explicitly, or it may be inherited from the event expression that governs the procedural block in which it is placed. The above example is equivalent to:

```
property rule = @(posedge clk)(a;[1]b;[1]c);
always @(posedge clk) begin
   <statements>;
   <statements>;
   end
```

A property when instantiated in a procedural block uses a similar mechanism as synthesis to infer the context of the clock, reset and enabling condition. Also, the synthesis rules are followed to determine whether a property can be placed in a specific procedural block. When illegally placed in a procedural block, an error is reported.

Note: The details of these rules are being worked out presently and will be spelled out.

11.11 Grouping Assertions as a Library

The syntax for library groupings is as follows:

template_declaration ::=

template template_identifier [**(** template_formal_list **)**] **;**
{ template_item_declaration }
endtemplate [**:** template_identifier]

template_formal_list ::=

task_formal_arg { **,** task_formal_arg }

task_formal_arg ::=

[**input** | **output** | **inout**] [data_type] formal_identifier [= boolean_expr | sequence_expr]

template_item_declaration ::=
property_decl
| directive_decl
| seq_decl
| bool_decl

This section describes how to group statements to construct a library of properties and expressions. Such a group is called **template** which is given a name and can be instantiated with parameters. When instantiated with parameters, the parameters provide the binding to the actual design objects or other definitions specified elsewhere in the description.

A formal parameter is used to replace a name in the template body. The formal parameter can have an optional specification of type and direction.

The default values for a formal parameter can be specified by using an equal sign with the left-hand side of the equal sign as the formal parameter name and right-hand side as the default value. For example,

```
template hold(exp, min = 0, max = 15, clk);
    seq @(posedge clk) ova_e_hold = ( past(exp)==exp)*[min:max] );
endtemplate
```

The body of the template may contain:

- **property** declaration
- verification directives
- **seq** and **bool** declarations

A **template** is instantiated with the following syntax:

template_instantiation ::= template_identifier [instance_name] [(list_of_port_connections)];

The actual parameters can be given as an ordered list, as a named list. In an ordered list, the parameters are listed in the same order as in the template definition.

For example, the hold template defined above can be instantiated with:

```
hold ordered(counter, 2, 5, rose clk);
```

Or it can be instantiated with:

```
hold named(.exp(counter), .min(2), .max(5), .clk(rose clk));
```

The template instance name is optional. When the name is not specified, the name is the global sequence number of the instance in the form *seq_number*. For example, the first template instance compiled would be assigned the name ti1.

As template instances are expanded, the names of declarations in the template body are constructed by appending the definition name with the template instance name and a dot character. Such an expansion of a name uniquely identifies its definition. The following example illustrates the name expansion of definitions.

```
template range();
   bool c1 = ( enable );
```

```
      seq @(posedge clk2) crange_en = ( if (c1) (minval <= expr) );
      range_chk: assert (crange_en);
    endtemplate
    range t1();
    range t2();
    property term_check = (if (t1.c1) p_low ;[1] p_end);
```

The definitions `c1`, `crange_en`, and `range_chk` are expanded as shown below.

```
    bool t1.c1 = ( enable );
    seq @(rose clk2) t1.crange_en = ( if (t1.c1) (minval <= expr) );
    t1_range_chk: assert (t1.crange_en);
    bool t2.c1 = ( enable );
    @ (rose clk2) seq t2.crange_en = ( if (t2.c1) (minval <= expr) );
    t2.range_chk: assert (t2.crange_en);
    property term_chk = (if (t1.c1) p_low ;[1] p_end;)
```

Using this naming scheme, an expression defined within a template can be referenced outside the template via a standard hierarchical reference.

The actual parameters may not resolve all signals specified within the template. When the template is instantiated, the parameters and the unresolved signals get bound to the design objects in the instantiating scope.

If a formal parameter is specified with a default value in the template definition, then the corresponding actual parameter may be optionally omitted. In the example below, the formal parameter `max` is not supplied when the template is instantiated.

```
    template hold(exp, min = 0, max = 15, clk);
        seq @(rose clk) e_hold = ( ($past(exp) == exp) * [min:max] );
    endtemplate
    hold hold_instance(s, 5, , rose clk);
```

If the default parameter value is not declared in the template definition, omission of the corresponding actual parameter value in the template instantiation will result in an error.

11.12 Inferred Template Parameters from Context

In addition to specifying default clocks and reset conditions that simplify the use of assertions and templates within a design code, it is also possible to specify that certain template parameters should be automatically extracted from the context of the template instance unless specified explicitly.

There are two aspects of the source context feature:

- context inference, where certain assertion elements are derived from the source code, and
- context substitution, where the inferred elements are substituted assertions

Source context is inferred from the source code. The context can be determined at the design elaboration time. The context consists of:

- clock ($clk), declared using the clock construct, or inferred from an always block, or defaults to simulation time.
- accept ($acc), inferred reset signal from an always block or if none then the condition declared using a default accept construct, or if none then 0 (false).
- reject ($rej), inferred reset signal from an always block or if none then the condition declared using a default reject construct, or if none then 0 (false).
- enable ($enb), nested conditions in a procedural block leading to the assert block ($enb is true otherwise).
- expression ($lhs), left hand side expression of immediately preceding assignment, or immediately preceding variable declaration.
- expression ($rhs), right hand side expression of immediately preceding assignment.

$clk, $acc, $rej, $enb, $lhs or $rhs can be used in expressions or as default values in template declarations. When used, their values are substituted at compile time from the source context (of the expression or template instantiation context).

For example,

```
reg1 = (a & b);
property c1 = ($rhs > 0);
```

The use of context substitution feature is valuable for defining assertion templates, as shown in the following example:

```
template one_hot(en = $enb, clk = $clk, a = $lhs)
begin
   seq @(clk) one_hot_c = ( if (en) ($coutones(a) == 1) );
   property one_hot_prop = (one_hot_c) else ($fatal());
end
endtemplate
```

NOTE: The following example needs to be reworked.

```
always @(negedge sysclk)
begin
   if (start) begin
      state_reg = (a & b);
      c2: one_hot();
      // checks one hot for state_reg
      // substitutes "rose sysclk" for $clk,
      // "start" for $enb and "state_reg" for $lhs
   end
end
```

11.13 Binding Properties to Scopes or instances

To facilitate verification separate from the design, it is possible to specify properties and bind them to specific modules or instances. This can be done by adding a scope or instance qualifier to the label of the property:

bind_instance ::= instance_identifier::<property_name>

bind_module ::= module_identifier::<property_name>

bind_instance provides the scope of the specified instance to the property for name resolution. The named property is applied only to the specified instance.

bond_module provides the scope of the specified module to the property. The named property is applied to all instances of the specified module.

Appendix A

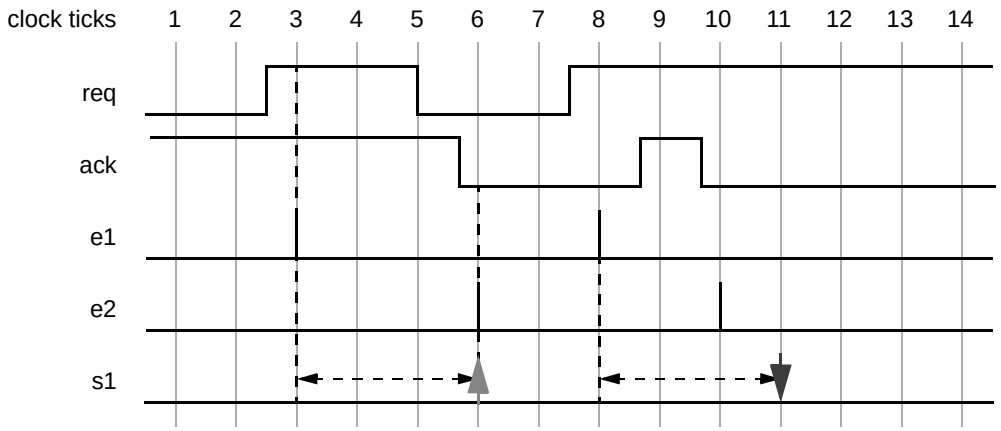
Matching a Sequence

Generally, at any given time, there may be several simultaneous evaluations of an assertion taking place. Although these evaluations are overlapped in time, each evaluation is independent of one another. To test the assertion at a clock tick, a new evaluation attempt for the expression is carried out, independent of any attempt at a previous clock tick. The results of each attempt determine the success/failure of an assertion for that clock tick. Because of the independence of each evaluation, we will be discussing one attempt when we describe the behavior of the language constructs.

Consider the sequence of boolean expressions, `s1,` in Figure 11-20. `s1` is defined as:

```
e1 ;[3] e2
```

Figure 11-20—Matching a Sequence



The above example says that e2 is expected to occur at the third clock tick after the occurrence of e1. Figure 11-20 illustrates this process for an attempt starting at clock tick 3 and shows how the time is advanced for the attempt. `e1` is evaluated to be true at clock tick 3. The outcome of this result is the continuation of checking the expression for the next checkpoint, which is `e2` at clock tick 6. No evaluation or checking is performed at clock ticks 4 and 5 for this attempt. Thus, variables can take on any values during these clock ticks. Expression `e2` occurs at clock tick 6, so the sequence is said to match for the attempt starting at clock tick 3. However, the evaluation attempt at clock tick 8 does not result in a match, as `e2` is false at clock tick 11.

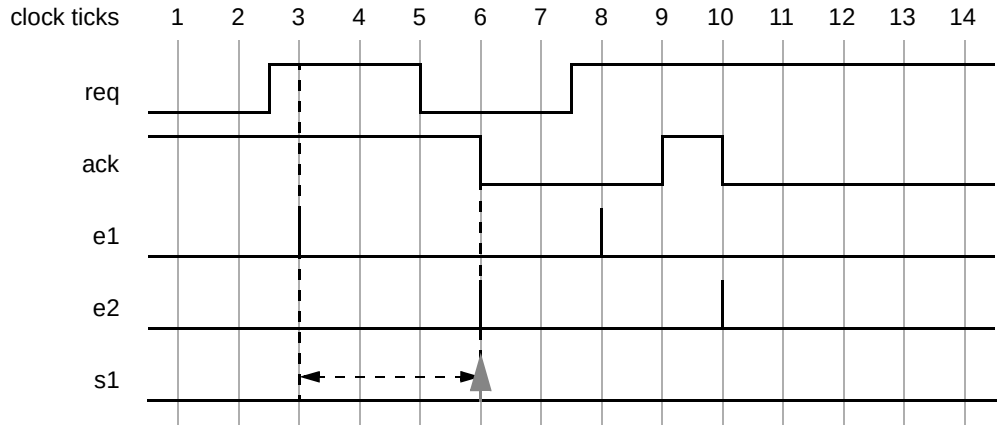
NOTE: A sequence match is indicated as an upward arrow and a no match is indicated as a downward arrow. At all other points in time where there is no upward or downward arrow, the expression is in the process of evaluating a match. A time line is shown with a dashed horizontal line ◄- - - - ► with a left and a right arrow to indicate that an evaluation is in progress during that time period.

Start and End Time of A Sequence

Each sequence has a start time and an end time. As seen from the example in Figure 11-20, while monitoring sequences the reference time (current time) is advanced according to the clock ticks between the checkpoints.

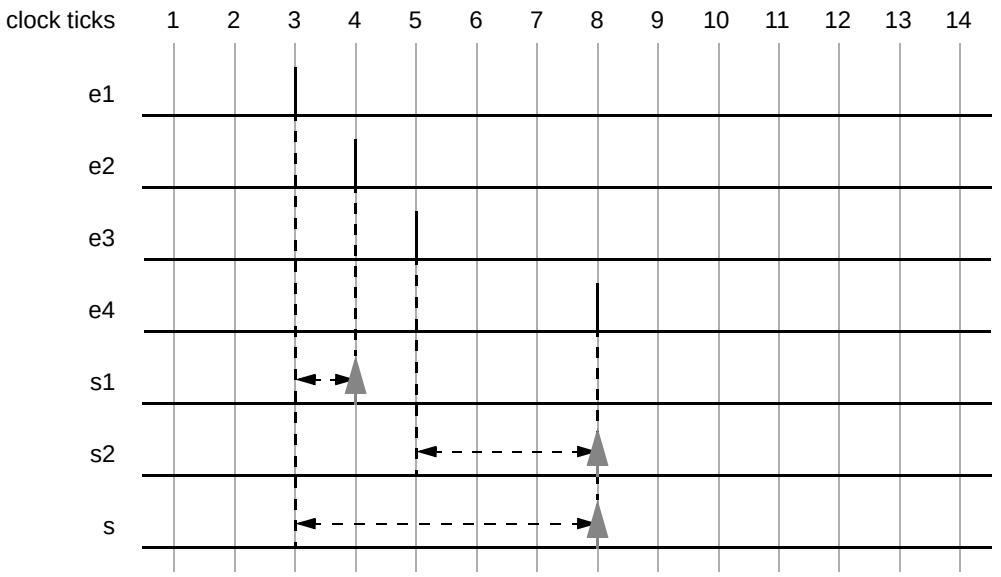
The start time for a sequence match is the time from which a new evaluation attempt of the sequence expression begins. The end time for that evaluation attempt is the time at which a match or a no-match for the sequence is detected. Let us examine the start and end times of the evaluation attempt at clock tick 3 for the example illustrated in Figure 11-21. The attempt starting at clock tick 3 matches at clock tick 6, so the start and end times are clock ticks 3 and 6 respectively.

Figure 11-21—Start and End Times of a Sequence



A sequence can consist of sub-sequences, again dispersed in time. The same rules apply to sub-sequences regarding the start time and end time. Now, assume a series of expressions (`e1`, `e2`, `e3` and `e4`) at the corresponding clock ticks (3, 4, 5 and 8). Consider a sequence s consisting of two sub-sequences s1 and s2, where s1 is `(e1 ;[1] e2)` and s2 is `(e3 ;[3] e4)`, and s is defined as `(s1 ;[1] s2)`, and shown in Figure 11-22. The time clause `;[1]` specifies the expectation of the occurrence of the second operand expression in the next clock tick after the occurrence of the first operand expression. The time clause `;[3]` specifies the expectation of the occurrence of the second operand expression at the third clock tick after the occurrence of the first operand expression.

Figure 11-22—Start and End Times of Sub-sequences



The sequence expression is:

```
(e1 ;[1] e2) ;[1] (e3 ;[3] e4)
```

Let us examine the evaluation attempt at clock tick 3 in Figure 11-22.

- The attempt starting at clock tick 3 succeeds for sub-sequence s1 at clock tick 4.
- Next, the evaluation of s2 begins at the next clock tick after sub-sequence s1, and the start time of sub-sequence s2 becomes 5.

- Sub-sequence s2 terminates when expression `e4` occurs, resulting in the end time for sub-sequence s2 as clock tick 8.

Single vs. Multiple Sequences of Evaluation

A more complex scenario arises when the expression evaluation branches out to compute all alternative sequences implied by a construct. In such cases, a sequence match is determined for every sequence independent of each other. The expression can result in multiple successful or failed matches. If such a sequence expression is a sub-expression of a larger expression, then the resulting matches are used to determine sequence matches of the enclosing expression.

To specify a range of possible delays between subsequent samples, the delay specifier is modified to use the standard range syntax, as in

```
a;[1:3]b;[1]c
```

This specifies that a will be true on the current sample, followed by b on the first, second, or third subsequent sample, and c will be true on the sample following b. Thus, any of the following sequences will match this sequential expression:

```
a;[1]b;[1]c
a;[2]b;[1]c
a;[3]b;[1]c
```

In the range syntax,

```
a;[min:max]b
c;[min:max];d
```

both min and max must be a constant expression or a literal, min must be greater than or equal to zero, and max must be greater than or equal to min.

In such cases, a sequence match is determined for every sequence independent of each other. The expression can result in multiple successful or failed matches. If such a sequence expression is a sub-expression of a larger expression, then the resulting matches are used to determine sequence matches of the enclosing expression. An example of evaluating multiple sequences follows:

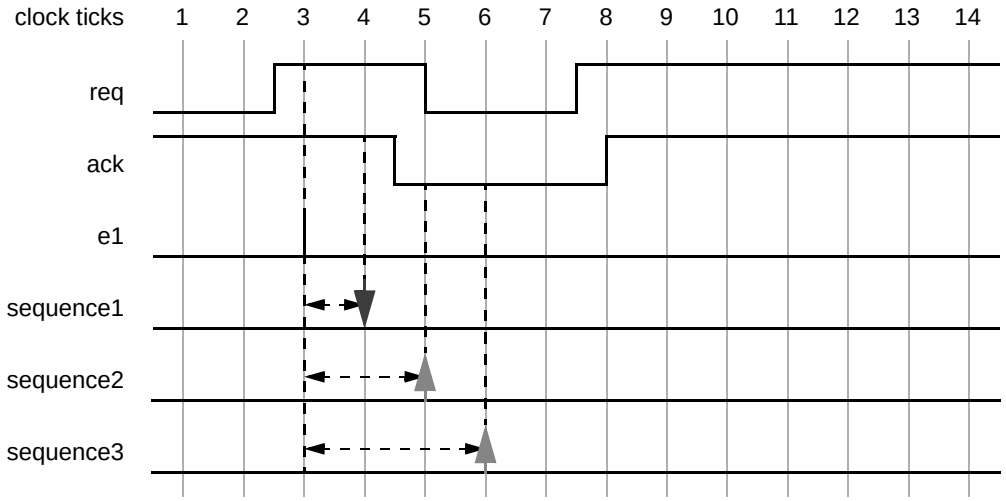
```
e1 ;[1:3] (ack==0)
```

`e1` is defined as (`rose req`).

This statement says that signal ack must be low at the first, second, or third clock ticks after the occurrence of e1. To determine a match for each of these three cases, three separate evaluations are started. An example is illustrated in Figure 11-23. The three sequences are:

```
e1 ;[1] (ack==0)
e1 ;[2] (ack==0)
e1 ;[3] (ack==0)
```

Figure 11-23—Evaluating Multiple Sequences



Let us consider an evaluation attempt at clock tick 3:

- At clock tick 3, `e1` occurs, so three sequences are started.
- Sequence1 fails to match at clock tick 4 as signal `ack` is 1.
- Sequence2 and sequence3 match at clock ticks 5 and 6 respectively, as signal `ack` is 0 at those clock ticks.

As shown, the sequence results in two matches for the example.

Note: When a sequence expression is used in a *property* directive that is to be verified on the design, the first match establishes the success of the property. In the example in Figure 11-23 a property verifying

```
e1 ;[1:3] (ack==0)
```

would declare success at clock tick 5.

Appendix B

BNF of Assertions

sequence_expr ::=
expression // expression is defined by System verilog
| identifier [**(** expression_list **)**] // identifier must be a sequence or a bool name
| [repeat_range] sequence_expr { **;** [repeat_range] sequence_expr }
| sequence_expr ***** [repeat_range]
| sequence_expr **=*** repeat_range
| sequence_expr **and** sequence_expr
| sequence_expr **or** sequence_expr
| sequence_expr **intersect** sequence_expr
| **first_match** sequence_expr
| **if** (expression) sequence_expr
| **if** (expression) sequence_expr **else** sequence_expr
| restriction **within** sequence_expr
restriction ::= **istrue** expression

| **length** repeat_range
| **occurs** sequence_expr
| **let (** variable_declaration **)**

repeat_range ::=
[*constant*_expression [**:** *constant*_expression] **]** // *constant*_expression must be 0 or greater
[*constant*_expression **: inf]**

variable_declaration ::= type identifier **=** expression

sequence_declaration ::= **seq** [event_control] named_seq { **,** named_seq} **;**

named_seq ::=identifier [**(** identifier { **,** identifier} **)**] **=** sequence_expr

bool_declaration ::= **bool** named_bool { **,** named_bool } **;**

named_bool ::= identifier [**(** identifier { **,** identifier } **)**] **=** expression

prop_declaration ::= **property** named_prop { **,** named_prop } **;**

named_prop ::= identifier [**(** identifier { **,** identifier } **)**] **=** prop_expr

prop_expr ::=
[**initial**] [reset_mode **(** expression **)**] [**not**] clocked_sequence
| identifier [**(** expression_list **)**] // identifier must be a property

reset_mode ::=
accept
| **reject**

clocked_sequence ::= [event_control] sequence_expr

property_directive ::=
[identifier **:**] directive **(*) ;**
| [identifier **:**] directive prop_expr [{ **,** prop_expr }] **;**
| [identifier **:**] directive prop_expr [{ **,** prop_expr }] action_block **;**

directive ::=
assert
| **assume**
| **restrict**
| **cover**

immediate_assertion ::=
[identifier **:**] **check (** expression **)**;
| [identifier **:**] **check (** expression **)** action_block

action_block ::=
do statement
| **else** statement
| **do** statement **else** statement

TEMPLATES