

Section 1 Assertions

1.1 Introduction (informative)

System Verilog adds features to specify assertions (or properties) of a system. An assertion specifies a specific behavior of the system. There are two kinds of assertions: concurrent or immediate.

Immediate assertions follow simulation event semantics for their execution and get executed like a statement in a procedural block. Immediate assertions are primarily 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.

This section describes both types of assertions.

1.2 Syntax

The following statements are part of `module_or_generate_item` and `interface_or_generate_item`.

```
concurrent_assertion_item ::=
    concurrent_assert_statement
    | concurrent_cover_statement
    | concurrent_assertion_item_declaration
concurrent_assertion_item_declaration ::=
    property_declaration
    | sequence_declaration
```

The following are part of `statement_item`

```
procedural_assertion_items ::=
    concurrent_assert_statement
    | concurrent_cover_statement
    | immediate_assert_statement
```

1.3 Immediate assertions

The immediate assertion 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. The syntax of the immediate assertion statement is as follows.

```

immediate_assert_statement ::=
    assert ( expression ) action_block
action_block ::=
    statement_or_null
    | [statement] else statement_or_null

```

Syntax 1-1—Immediate assertion syntax

The statement associated with the success of the `assert` statement is called `pass` statement, and is executed if 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 user specified action is taken when the `assert` expression is true. The statement associated with `else` is called a 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 named block around the assertion statement (or any other SystemVerilog statement) and can be displayed using the `%m` format code.

```

assert_foo : assert(foo) $display("%m passed"); else $display("%m failed");

```

Note: The `pass` and `fail` statements are executed in the reactive region. This is explained in the scheduling semantics section of System Verilog.

System tasks are provided to control assertions, and limit them to part of the design or part of the simulation time as described in Section 22.6.

EDITOR's NOTE : Cross-ref for section 22.6

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.

Need software cross reference above

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)
    assert(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.

```

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

```

1.4 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 values. 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 values of variables used in assertions are sampled in the preponed region of a time slot and the assertions are evaluated during the observe region. This is explained in the scheduling semantics section of System Verilog.

The timing model employed in a 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.

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, and the sampled values for that simulation time are used for evaluation. In an assertion, the sampled value is the only valid value of a variable at a clock tick. Figure 1-1 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.

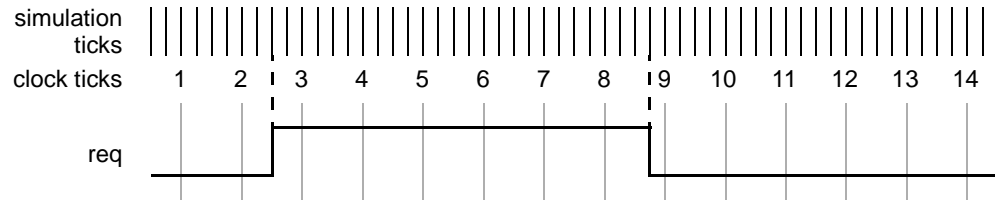


Figure 1-1—Sampling a Variable on Simulation Ticks

An expression is always tied to a clock definition. The sampled values are used to evaluate value change expressions or boolean sub-expressions that are required to determine a match with respect to a sequence expression.

Note:

- It is important to ensure that the defined clock behavior is glitch free. Otherwise, wrong values may get sampled.
- If a variable that appears in the expression for clock also appear in an expression for the assertion, the values of the two usages of the variable may be different. The value of the variable used in the clock expression is the current value, while for the assertion the sampled value of the variable is used.

The clock expression that controls evaluation of a sequence may be more complex than just a single signal name. An expression such as `(clk && gating_signal)` or `(clk iff gating_signal)` may 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.

1.5 Sequences

```

sequence_expr ::=
    [ cycle_delay_range ] sequence_expr { cycle_delay_range sequence_expression }
    | expression { , function_blocking_assign } [ boolean_abbrev ]
    | sequence_instance [ sequence_abbrev ]
    | ( sequence_expr ) [ sequence_abbrev ]
    | sequence_expr and sequence_expr
    | sequence_expr intersect sequence_expr
    | sequence_expr or sequence_expr
    | first_match ( sequence_expr )
    | expression throughout sequence_expr
    | sequence_expr within sequence_expr

boolean_abbrev ::=
    consecutive_repetition
    | non_consecutive_repetition
    | goto_repetition

sequence_abbrev ::=
    consecutive_repetition

consecutive_repetition ::=
    [* const_range_expression ]

non_consecutive_repetition ::=
    [*= const_range_expression ]

goto_repetition ::=
    [*-> const_range_expression ]

const_or_range_expression ::=
    constant_expression
    | constant_range_expression

cycle_delay_range ::=
    ## constant_expression
    | ## [ constant_range_expression ]

constant_range_expression ::=
    | constant_expression : constant_expression
    | constant_expression : $

```

Syntax 1-2—Sequence syntax

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 when an expression needs to hold true. These sequential regular expressions can actually describe a set of one or more sequences that satisfy the sequential expression.

The basic composition of a sequence consist of a boolean expression concatenated by another boolean expression. The concatenation specifies a delay, using `##`, between the two boolean expressions. The following is the syntax for sequence concatenation.

```
sequence_expr ::=
    [ cycle_delay_range ] sequence_expr { cycle_delay_range sequence_expression }
cycle_delay_range ::=
    ## constant_expression
    | ## [ constant_range_expression ]
constant_range_expression ::=
    | constant_expression : constant_expression
    | constant_expression : $
```

Syntax 1-3—Sequence concatenation syntax

In this syntax:

- `constant_expression` is computed at compile time and must result in an integer value
- `constant_expression` can only be 0 or greater
- The keyword `$` is used to indicate the end of simulation. For formal verification tools, `$` is interpreted as infinity.
- When a range is specified with two expressions, the second expression must be greater or equal to the first expression.

The context in which a sequence occurs determines when the sequence is evaluated. The first element in a sequence is checked at the first occurrence of the clock at or after the element that triggered evaluation of the sequence. Each successive element (if any) in the sequence is checked at the next subsequent occurrence of the clock.

A delay specification must be followed by an expression. A delay of `##1` indicates that the next element should occur a single cycle later than the ‘current’ cycle. A delay of `##0` specifies that the next element should occur in parallel with the ‘current’ cycle.

The following are examples of unary delay expressions. ``true` is used to indicate that the expression is true.

```
##0 a means a
##1 a means `true ##1 a
##2 a means `true ##1 `true ##1 a
##[0:3]a means(a) or (`true ##1 a) or (`true ##1`true ##1 a) or
      (`true ##1 `true ##1 `true ##1 a)
```

Note:

- ``true` is not provided by the language, but is used as an expression macro that always returns true. It can be defined as


```
`define true 1
```

An example of a delay expression is as follows:

```
a ##2 b means a ##1 `true ##1 b
```

A sequence:

```
req ##1 gnt ##1 !req
```

This sequence specifies that **req** be true on the current clock tick, **gnt** will be true on the first subsequent tick and **req** will be false on the next clock-tick after that. The '##1' operator specifies one clock tick separation. 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 subsequent sample, as shown in figure Figure 1-2.

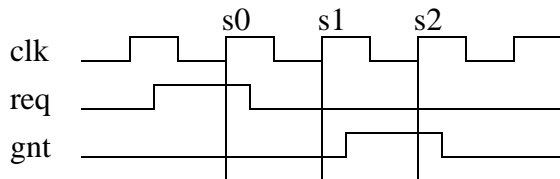


Figure 1-2—Concatenation

The following specifies that 'b' will be true on the Nth sample after 'a':

```
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 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 a concatenated sequence 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 follows:

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

In the above case, signal gnt must be true at some clock tick between 4 to 32 after the current sample.

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

```
req ##[4:$] gnt
```

A sequence can be unconditionally extended by using 'true.

```
a ##1 b ##1 c ##3 `true
```

After signal c, the sequence length is extended by 3 clock ticks. Such adjustments in the length of sequences

are required when complex sequences are constructed by combining simpler sequences.

1.6 Declaring sequences

Sequences are declared using the following syntax.

```

sequence_declaration ::=
    sequence sequence_identifier [ sequence_formal_list ] ;
    { assertion_variable_declaration }
    sequence_spec ;
    endsequence [ : sequence_identifier ]
sequence_formal_list ::=
    ( formal_list_item { , formal_list_item } )
sequence_spec ::=
    multi_clock_sequence
    | sequence_expr
multi_clock_sequence ::=
    clocked_sequence { ## clocked_sequence }
clocked_sequence ::=
    clocking_event sequence_expr
sequence_instance ::=
    sequence_identifier [ ( actual_arg_list ) ]
formal_list_item ::=
    formal_identifier [ = actual_arg_expr ]
actual_arg_list ::=
    ( actual_arg_expr { , actual_arg_expr } )
    | ( .formal_identifier ( actual_arg_expr ) { , .formal_identifier ( actual_arg_expr ) } )
actual_arg_expr ::=
    event_expression
assertion_variable_declaration ::=
    data_type list_of_variable_identifiers;

```

Syntax 1-4—Declaring sequence syntax

The `clocking_event` specifies the clock for the sequence.

Formal arguments may be optionally specified. A formal argument is untyped, and is used for syntactic replacement of a name or an expression in the sequence.

An actual argument may replace

— a name that is not a definition name

- expression
- event control expression

Note that variables used in a sequence that are not formal arguments to the sequence are resolved hierarchically from the scope in which the sequence is declared.

```
sequence s1;
  @(posedge clk) a ##1 b ##1 c;
endsequence
sequence s2;
  @(posedge clk) d ##1 e ##1 f;
endsequence
sequence s3;
  @(negedge clk) g ##1 h ##1 i;
endsequence
```

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

Another example of sequence declaration with arguments is shown below:

```
sequence s20_1(data,en);
  (!frame && (data==data_bus)) ##1 (c_be[0:3] == en);
endsequence
```

Sequence s20_1 does not specify a clock. In this case, a clock would be inherited from some external source, such as a property or an assert statement. A sequence can be referred by its name. A hierarchical name can be used consistent with the System Verilog naming conventions. A sequence can be referenced in a property, an **assert** statement or a **cover** statement.

To use **sequence** 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 **sequence** expression is performed the same way as if the sequence expression contained in the **sequence** was a lexical part of the expression, with the formal arguments substituted by the actual ones and the remaining variables which were not arguments substituted from the scope of declaration. An example is shown below:

```
sequence s;
  a ##1 b ##1 c;
endsequence
sequence rule;
  @(posedge sysclk)
  trans ##1 start_trans ##1 s ##1 end_trans;
endsequence
```

Sequence rule is equivalent to:

```
sequence rule;
  @(posedge sysclk)
  trans ##1 start_trans ##1 a ##1 b ##1 c ##1 end_trans ;
endsequence
```

Any form of syntactic cyclic dependency of the sequence names is disallowed. The example below illustrates dependency of s1 on s2, and s2 on s1, which creates a cyclic dependency.

```
sequence s1;
  @(posedge sysclk) (x ##1 s2);
endsequence
sequence s2;
  @(posedge sysclk) (y ##1 s1);
```

```
endsequence
```

1.7 Sequence operations

1.7.1 Repetition in sequences

Following is the syntax for sequence concatenation (sequence_phrase from concatenation has been extended with repetition clauses).

```
sequence_expr ::=
    expression { , function_blocking_assign } [ boolean_abbrev ]
    | sequence_instance [ sequence_abbrev ]
    | ( sequence_expr ) [ sequence_abbrev ]
boolean_abbrev ::=
    consecutive_repetition
    | non_consecutive_repetition
    | goto_repetition
sequence_abbrev ::=
    consecutive_repetition
consecutive_repetition ::=
    [* const_range_expression ]
non_consecutive_repetition ::=
    [*= const_range_expression ]
goto_repetition ::=
    [*-> const_range_expression ]
```

Syntax 1-5—Sequence repetition syntax

The repetition counts are specified with range and must be literals or constant expressions.

Three kinds of repetition are provided:

- consecutive repetition ([*], where a sequence is consecutively repeated with one cycle delay between the repetitions
- goto repetition ([->*], where a boolean expression is repeated with one or more cycle delays between the repetitions and the resulting sequence terminates at the last boolean expression
- non-consecutive repetition (*=, where a boolean expression is repeated with one or more cycle delays between the repetitions and the resulting sequence may proceed beyond the last boolean expression, but before the occurrence of the boolean expression

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

```
a ##1 b ##1 b ##1 b ##1 c
```

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

```
a ##1 b [*3] ##1 c
```

A repetition specifies that the item or expression should occur a specified number of times. Each repeated item is concatenated (with a delay of 1 clock tick) to the next repeated item. A repetition of N specifies that the sequence should occur N times in succession - e.g.,

```
a [*3] means a ##1 a ##1 a
```

The syntax allows combination of a delay and repetition in the same sequence. The following are both allowed:

```
`true ##3 (a [*3])means    `true ##1 `true ##1`true ##1 a ##1 a ##1 a
(`true ##2 a) [*3]means    (`true ##2 a) ##1 (`true ## 2 a) ##1 (`true ##2 a)
which means `true ##1 `true ##1 a ##1`true ##1`true ##1 a ##1`true ##1`true ##1 a
```

A sequence can be repeated as follows:

```
(a ##1 b) [*5]
```

is same as:

```
(a ##1 b ##1 a ##1 b ##1 a ##1 b ##1 a ##1 b ##1 a ##1 b ##1 a ##1 b)
```

A repetition with a range of maximum and minimum number of times can be expressed with [* min:max]. As an example, the following two expression are equivalent.

```
(a ##1 b)[*1:5]
(a ##1 b)
or (a ##1 b ##1 a ##1 b)
or (a ##1 b ##1 a ##1 b ##1 a ##1 b)
or (a ##1 b ##1 a ##1 b ##1 a ##1 b ##1 a ##1 b)
or (a ##1 b ##1 a ##1 b ##1 a ##1 b ##1 a ##1 b ##1 a ##1 b)
```

The following two expression are also equivalent.

```
(a[*0:3] ##1 b ##1 c)
(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 ##1c)
```

To specify potentially infinite number of repetitions, \$ is used. So,

```
a ##1 b [*1:$] ##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 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 match should not be empty

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

```
a ##1 b [*->min:max] ##1 c
```

This is equivalent to:

```
a ##1 ((!b [*0:$] ##1 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[*->1:N] ##1 c //a followed by at most N occurrences of b, followed by c
```

A non-consecutive count repetition extends the non-consecutive repetition by extra samples where the boolean expression is not true.

```
a ##1 b [*=min:max] ##1 c
```

This is equivalent to:

```
a ##1 ((!b [*0:$] ##1 b)) [*min:max]) ##1 !b[*0:$] ##1 c
```

1.7.2 Value change functions

Three functions are provided to detect changes in values between two adjacent clock ticks: **\$rose**, **\$fell** and **\$stable**.

```
value_change_functions ::=
    $rose ( expression )
    | $fell ( expression )
    | $stable (expression )
```

Syntax 1-6—Value change function syntax

A value change expression at a clock tick detects the change in value of an expression from the value of that expression at the previous clock tick. The result of a value change expression is true or false, and can be used as a boolean expression. At the first clock tick after the assertion is started, the result of these functions are computed by comparing the current value to 'x'.

\$rose returns true if the least significant bit of the expression changed to 1. Otherwise, it returns false.

\$fell returns true if the least significant bit of the expression changed to 0. Otherwise, it returns false.

\$stable returns true if the value of the expression did not change. Otherwise, it returns false.

Figure 1-3 illustrates two examples of value changes:

— value change expression `e1` is defined as `$rose (req)`

— value change expression `e2` is defined as `$fell (ack)`

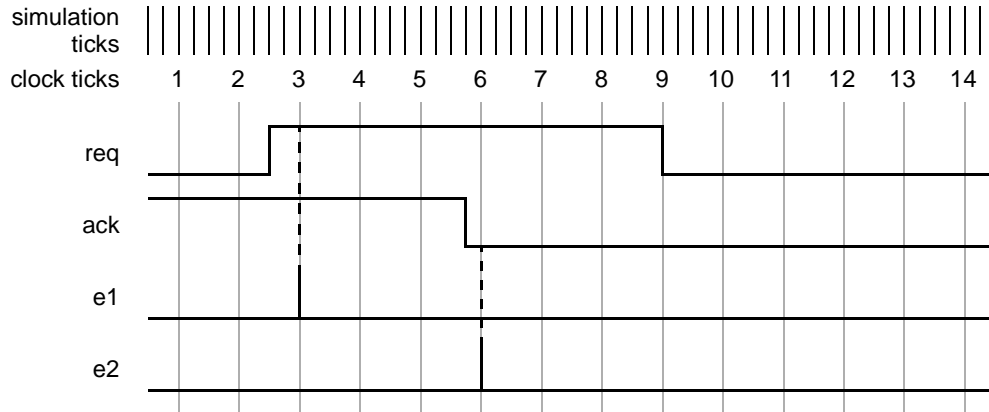


Figure 1-3—Value Change Expressions

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

1.7.3 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_expr ::=
    sequence_expr and sequence_expr
```

Syntax 1-7—and operator syntax

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. The operand expressions start at the same time. 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.

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

```
te1 and te2
```

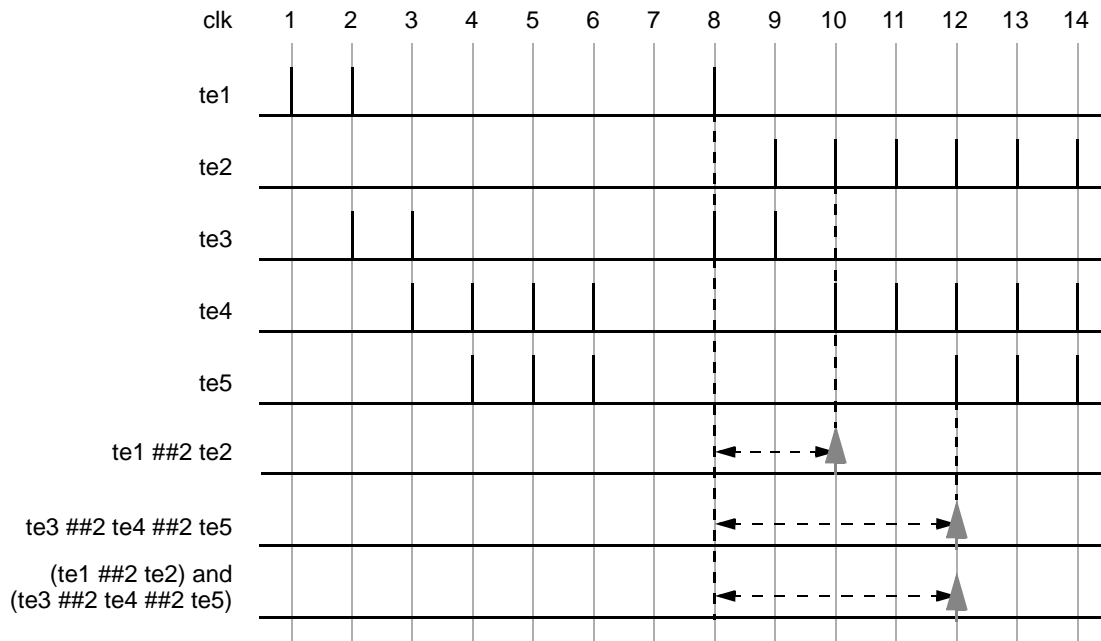
- Succeeds if `te1` and `te2` succeed.
- The end time is the end time of either `te1` or `te2`, whichever terminates last.

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

An example is illustrated in Figure 1-4. Consider the following expression with operator **and** where the two operands are sequences.

```
(te1 ##2 te2) and (te3 ##2 te4 ##2 te5)
```

Figure 1-4—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 1-4 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:

- 1) The first operand sequence starts five sequences of evaluation.
- 2) The second operand sequence has only one possibility of match, so only one sequence is started.
- 3) Figure 1-5 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.
- 4) 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 1-5 shows the two unique successes at clock ticks 12 and 13.

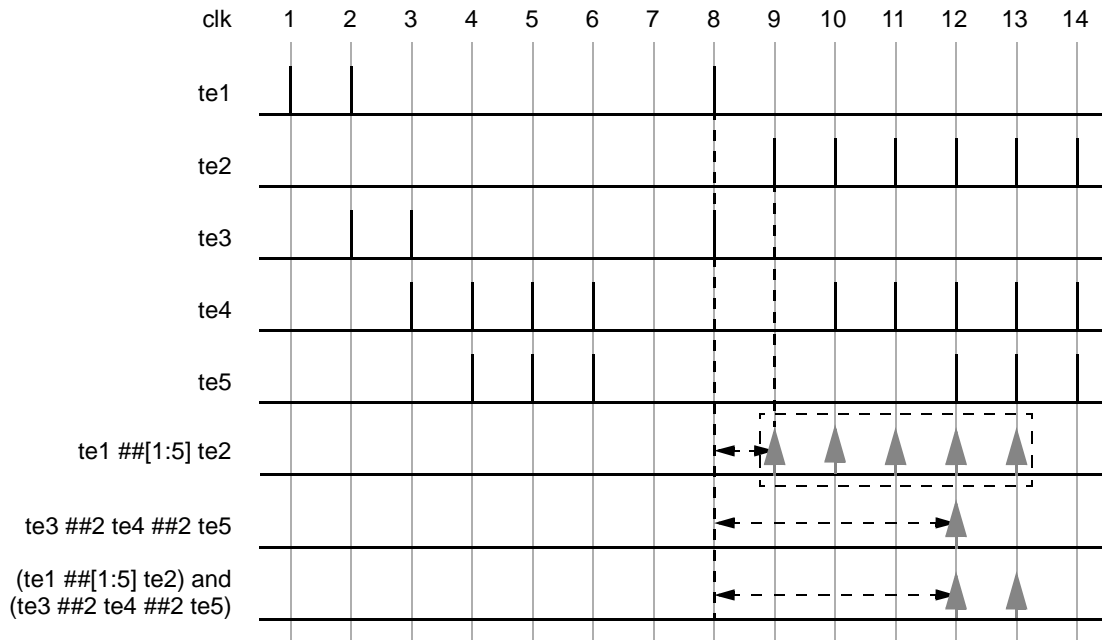


Figure 1-5—ANDing (and) Two Sequences Including a Time Range

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

An example is illustrated in Figure 1-6 to show the results for attempt at every clock tick. The expression matches at clock tick 1, 3, 8 and 14 because both `te1` and `te2` are simultaneously true. At all other clock ticks, the **and** operation fails because either `te1` or `te2` is false.

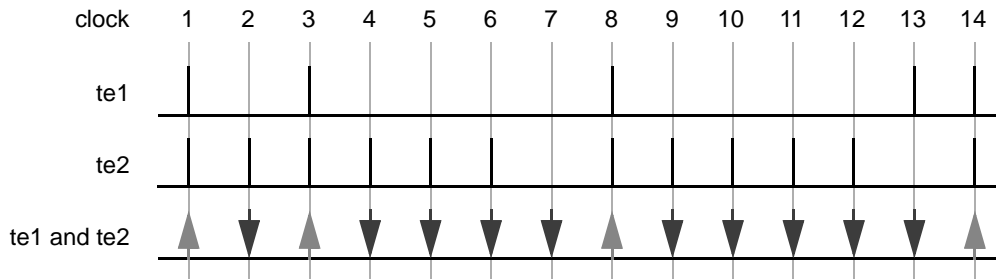


Figure 1-6—ANDing (and) Two Boolean Expressions

1.7.4 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_expr ::=
    sequence_expr intersect sequence_expr
```

Syntax 1-8—intersect operator syntax

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**.

For each attempted evaluation of `sequence_expr`, there could be multiple matches. 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.
- 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 length of the pair.

Consider the case in Figure 1-7. It is similar to Figure 1-5, except that **and** is replaced by **intersect**. Compared with Figure 1-5, there is only a single match in this case.

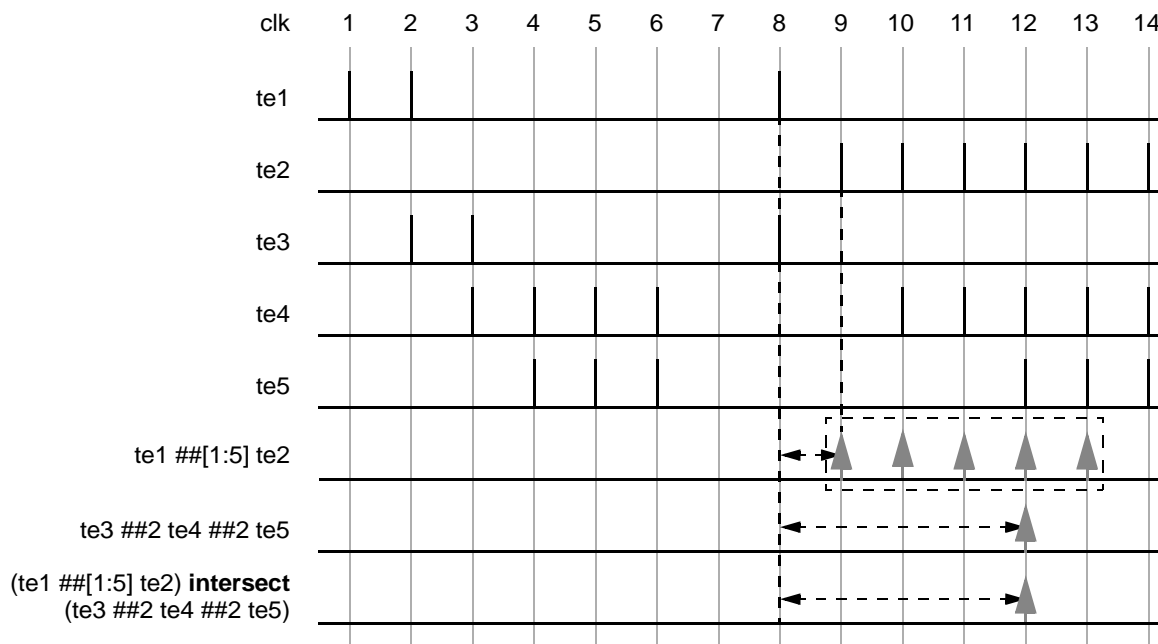


Figure 1-7—Intersecting two sequences

1.7.5 OR operation

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

```
sequence_expr ::=
    sequence_expr or sequence_expr
```

Syntax 1-9—or operator syntax

The two operands of `or` are sequence expressions.

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

```
te1 or te2
```

when operands `te1` and `te2` are expressions, the sequence matches whenever at least one of two operands `te1` and `te2` is evaluated to true.

Figure 1-8 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.

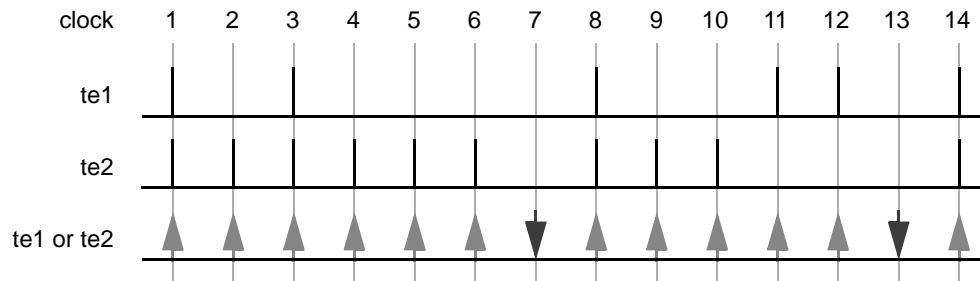


Figure 1-8—ORing (or) Two Sequences

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 1-9. Consider an expression with `or` operator where the two operands are sequences.

```
(te1 ##2 te2) or (te3 ##2 te4 ##2 te5)
```

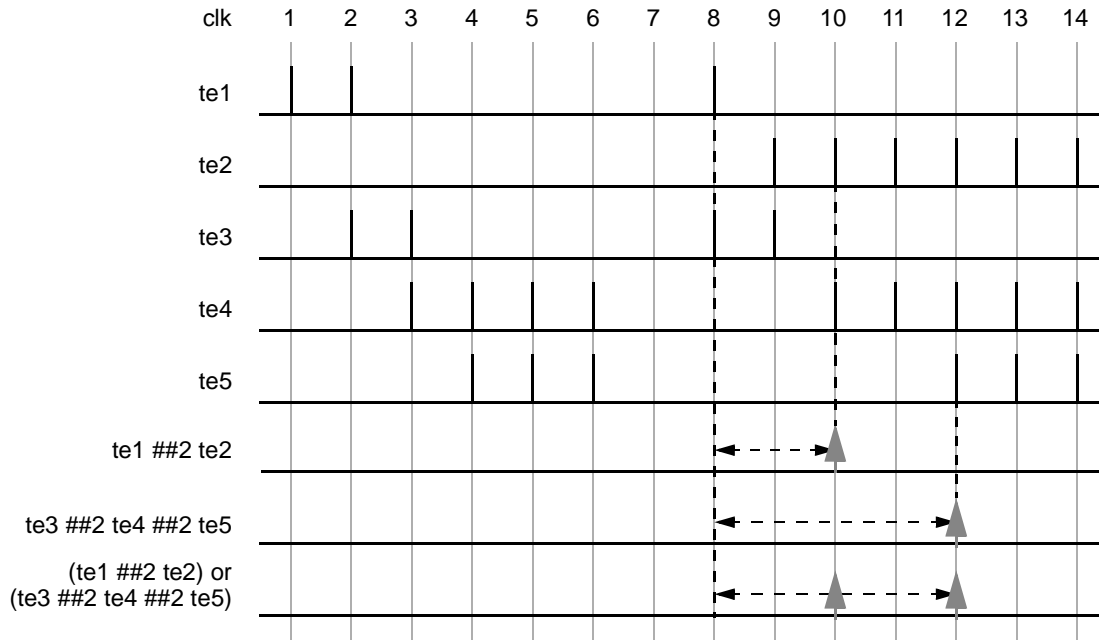


Figure 1-9—ORing (or) Two Sequences

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 1-9, 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 a 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 1-10, 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.

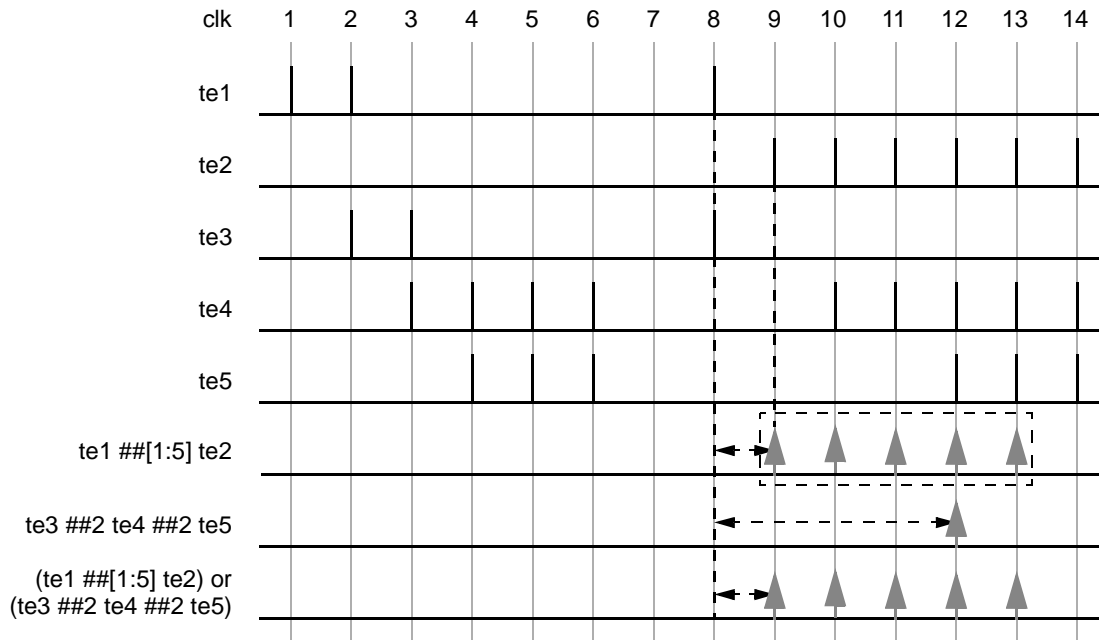


Figure 1-10—ORing (or) Two Sequences Including a Time Range

1.7.6 first_match operation

The **first_match** operator matches only the first match of possibly multiple matches for an evaluation attempt of a sequence expression. 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_expr ::=
    first_match ( sequence_expr )
```

Syntax 1-10—first_match operator syntax

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_phrase` 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 range of delay specification as shown below.

```
sequence t1;
    te1 ##[2:5]te2;
```

```

endsequence
sequence ts1;
    first_match(te1 ##[2:5]te2);
endsequence

```

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.

Another example,

```

sequence t2;
    (a ##[2:3] b) or (c ##[1:2] d);
endsequence
sequence ts2;
    first_match(ts2);
endsequence

```

Each attempt of sequence t2 can result in matches for up to four following sequences:

```

a ##2 b
a ##3 b
c ##1 d
c ##2 d

```

Sequence ts2 results in the earliest match. In this case, it is possible to have two matches ending at the same time.

```

a ##2 b
c ##2 d

```

1.7.7 Conditions over sequences

Sequences 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 values is prohibited while processing a transaction. Such situations can be expressed directly using the following construct:

<pre> sequence_expr ::= expression throughout sequence_expr </pre>

Syntax 1-11—throughout construct syntax

`expression` is an expression which must evaluate true at every clock tick during `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 **throughout** construct is an abbreviation for writing:

*(expression) [*0:\$] intersect sequence_expr*

Consider the example illustrated in Figure 1-11. If a constraint were placed on the expression as shown below, then the checker `burst_rule1` would fail at clock tick 9.

```
sequence burst_rule1;
  @(posedge mclk)
    $fell(burst_mode) ##0
    (!burst_mode) throughout (##2 ((trdy==0)&&(irdy==0)) [*7]);
endsequence
```

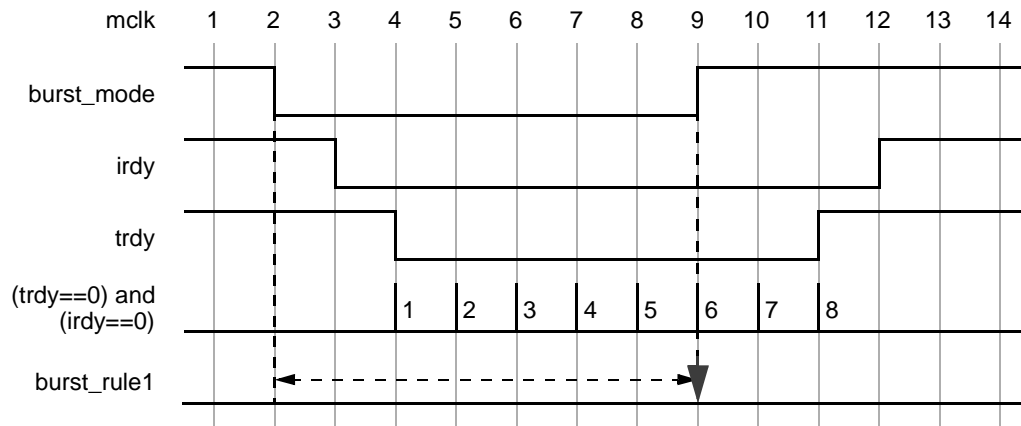


Figure 1-11—Match with throughout Restriction Fails

In the above expression, the value of signal `burst_mode` is required to be low during the sequence (from clock tick 2 to 10), 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 10, the expression would result in a match as shown in Figure 1-12.

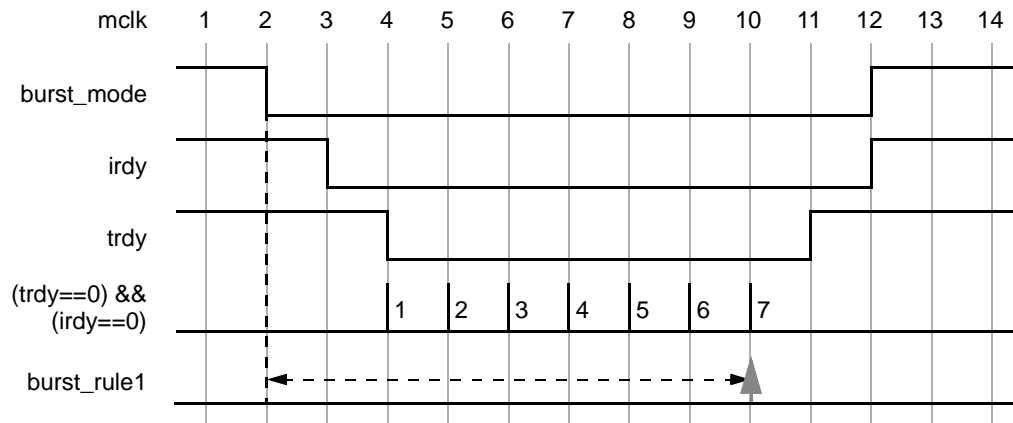


Figure 1-12—Match with throughout Restriction Succeeds

1.7.8 Sequence occurrence within another sequence

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

```
sequence_expr ::=
    sequence_expr within sequence_expr
```

Syntax 1-12—Sequence within another sequence syntax

The **within** construct

```
sequence_expr1 within sequence_expr2
```

is an abbreviation for writing:

```
(1[*0:$] ##1 sequence_expr1 ##1 1[*0:$]) intersect sequence_expr2
```

The sequence *sequence_expr1* must occur at least once 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 (start and end point being inclusive) of *sequence_expr2*.
- The end point of *sequence_expr1* must be between the start point and the end point (start and end point being inclusive) of *sequence_expr2*.

For example, the sequence expression

```
trdy[*7] within (($fell irdy) ##1 irdy[*8])
```

matches on the trace shown in Figure 1-12.

1.7.9 Detecting and using endpoint of a sequence

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

One is to reference the name of a sequence, and thereby causing it to be started at the point where it is referenced, such as shown below:

```
sequence s;
    a ##1 b ##1 c;
endsequence
sequence rule;
    @(posedge sysclk)
    trans ##1 start_trans ##1 s ##1 end_trans);
endsequence
```

Sequence *s* is evaluated one cycle after the occurrence of *start_trans* in the sequence rule.

Another way to use the **sequence** 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 method **ended**.

```
boolean_expr_op ::=
    seq_name.ended
```

Syntax 1-13—ended operator syntax

ended is a method on a sequence. The result of its operation is true or false. When method **ended** is applied in an expression, it tests whether sequence `seq_name` has reached the end point at that particular point in time. The result of **ended** does not depend upon the starting point of `seq_name`. An example is shown below:

```
sequence e1;
    @(posedge sysclk) $rose(ready) ##1 procl ##1 proc2 ;
endsequence
sequence rule;
    @(posedge sysclk) reset ##1 inst ##1 e1.ended ##1 branch_back;
endsequence
```

In this example sequence expression `e1` must end successfully one clock tick after `inst`. If the method **ended** wasn't there, sequence expression `e1` must start one clock tick after `inst`. Notice that method **ended** only tests for the end point of `e1`, and has no bearing on the starting point of `e1`.

ended can be used directly on sequences that do not have formal arguments. To use **ended** on a sequence with arguments, first define a sequence without formal arguments that instantiates the sequence with actual arguments. For example,

```
sequence e2(a,b,c);
    @(posedge sysclk) $rose(a) ##1 b ##1 c;
endsequence
sequence e2_instantiated;
    e2(ready,procl,proc2);
endsequence
sequence rule2;
    @(posedge sysclk) reset ##1 inst ##1 e2_instantiated.ended ##1 branch_back;
endsequence
```

1.7.10 Implication

This construct allows a user to monitor sequences based on satisfying some criteria. Most common uses are to attach a precondition to a sequence, where the evaluation of the sequence is based on the success of a condition.

Syntax 1-14—implication syntax

```

property_implication ::=
    sequence_expr |-> [not] sequence_expr
  | clocked_sequence |-> [not] sequence_expr
  | sequence_expr |=> [not] sequence_expr
  | clocked_sequence |=> [not] sequence_expr
  | multi_clock_sequence |=> [not] multi_clock_sequence
clocked_sequence ::=
    clocking_event sequence_expr
multi_clock_property_expr ::=
    multi_clock_sequence
  | multi_clock_sequence |=> [ not ] multi_clock_sequence

```

This clause is used to precondition monitoring of a sequence expression and is allowed at the property level. The result of the implication is either true or false.

The following points should be noted for |-> implication.

- antecedent sequence_expr can result in multiple successful sequences.
- If there is no match of the antecedent sequence, implication succeeds vacuously by returning true.
- For each successful match of antecedent sequence_expr, consequent sequence_expr is separately evaluated, beginning at the end point of the match. That is, the end point of matching sequence from antecedent sequence_expr overlaps with start point of the consequent sequence_expr
- All matches of antecedent sequence_expr must satisfy consequent.
- Nesting of implication is not allowed.

Two forms of implication are provided: overlapped using operator |->, and non-overlapped using operator |=>. For overlapped implication, if the expression evaluates to true, then the first element of the sequence_expr is evaluated on the same clock tick. For non-overlapped implication, the first element of the sequence_expr is evaluated on the next clock tick So,

```
boolean_expr |=> [not] sequence_spec
```

is equivalent to:

```
boolean_expr ##1 'true |-> [not] sequence_spec
```

If **not** is used on the consequent, the result of consequent sequence_expr is reversed.

The use of implication when multi-clock sequences are involved is explained in Section 1.11.

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:

```

property data_end;
    @(posedge mclk)
    data_phase |-> ((irdy==0) && ($fell(trdy) || $fell(stop))) ;
endproperty

```

Each time a data phase completes, a match for `data_end` is recognized. The attempt at clock tick 6 is illustrated in Figure 1-13. 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.

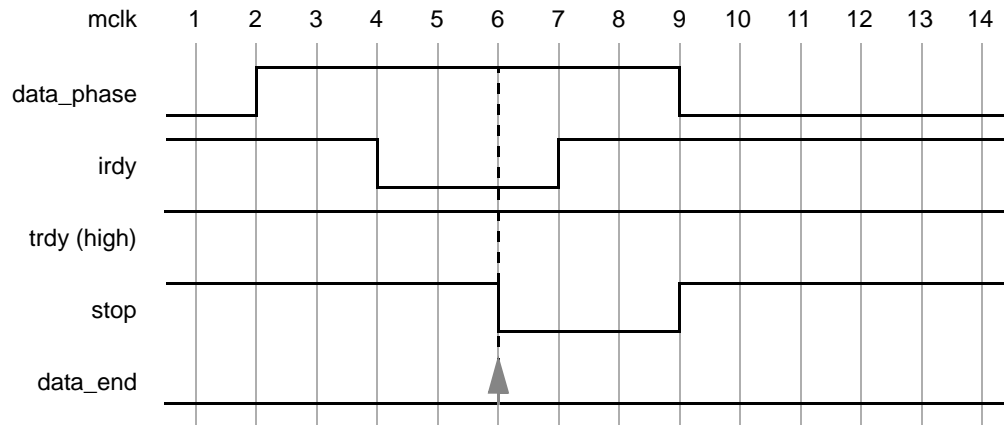


Figure 1-13—Conditional Sequence Matching

`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 property is written to express this condition as shown below.

```

`define data_end (data_phase && ((irdy==0) && ($fell(trdy) || $fell(stop))))
property data_end_rule1;
  @(posedge mclk)
  `data_end |-> ##[1:2] $rose(frame) ##1 $rose(irdy);
endproperty

```

property `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 evaluate `data_end_rule1` is considered true. Otherwise, the following sequence expression is evaluated. The sequence expression:

```
##[1:2] $rose(frame) ##1 $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 1-14. 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.

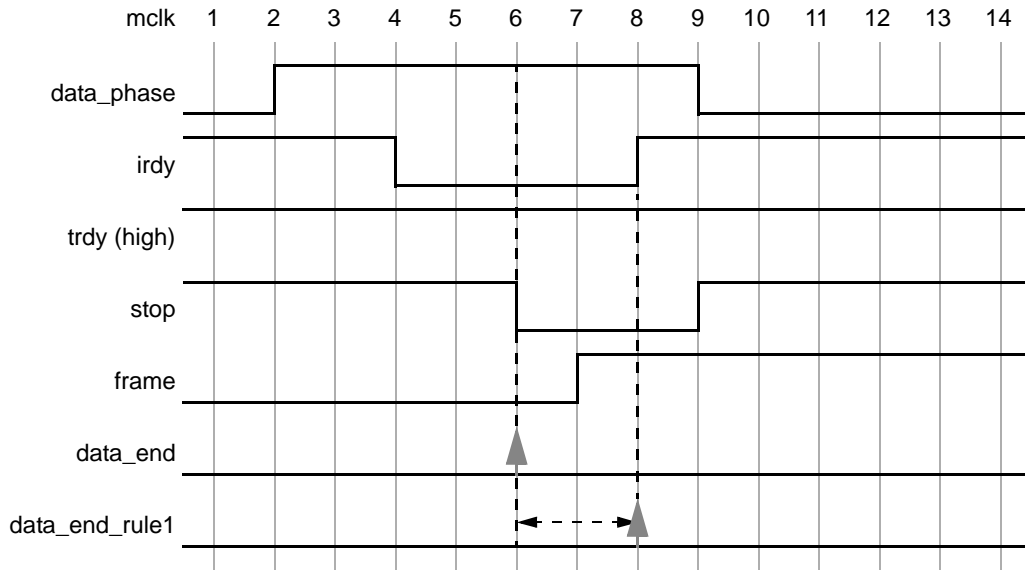


Figure 1-14—Conditional Sequences

Generally, assertions are associated with preconditions so that the checking is performed only under certain specified conditions. As seen from the previous example, the `|>` operator provides this capability to specify preconditions 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 1-15.

```
property data_end_rule2;
  @(posedge mclk) ##[1:2] $rose(frame) ##1 $rose(irdy);
endproperty
```

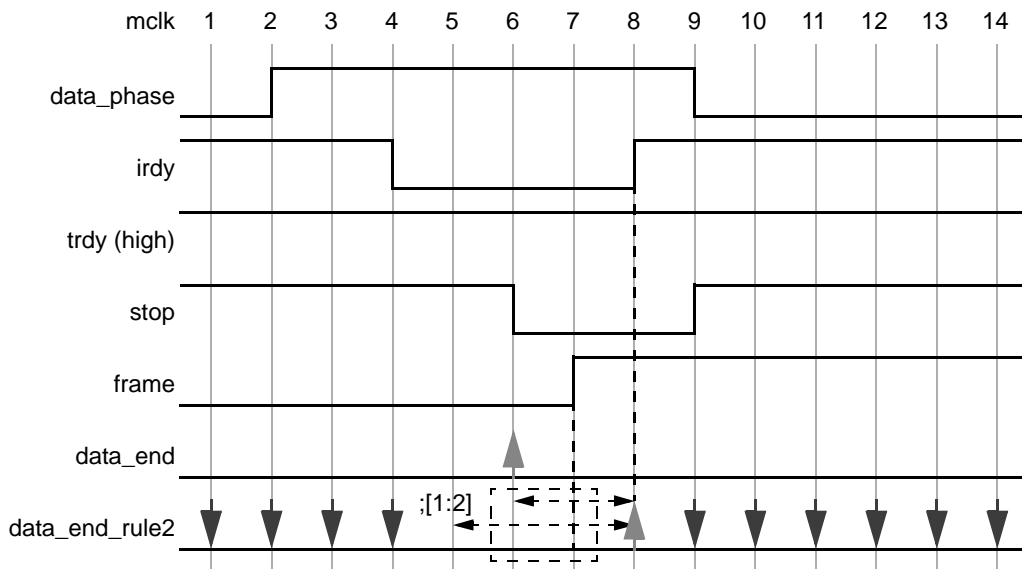


Figure 1-15—Results without the Condition

The property 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 property fails 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 1-15, removing the precondition of checking `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.

An example of sequential implication is:

```
(a ##1 b ##1 c) |-> (d ##1 e)
```

If the sequence `(a ##1 b ##1 c)` matches then the sequence `(d ##1 e)` must also match. On the other hand, if the sequence `(a ##1 b ##1 c)` does not match, then the result is true.

Consider now:

```
(a ##[1:3] b ##1 c) |-> (d ##1 e)
```

In the above example, all matches of `(a ##[1:3] b ##1 c)` must match `(d ##1 e)`. If there are no matches of `(a ##[1:3] b ##1 c)`, then there is a vacuous success for the property.

The following example illustrates chaining of sequential implications.

```
property p16;
    (write_en & data_valid) ##0
    (write_en && (retire_address[0:4]==addr)) [*1] |->
    ##[3:8] write_en && !data_valid && (write_address[0:4]==addr);
endproperty
```

Multi-clock sequence implication is explained in Section 1.11.

1.8 Manipulating data in a sequence

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 creation of a variable and its assignment is achieved by using the local variable declaration in sequence or property definition and making an assignment in the sequence.

```
sequence_expr ::=
    expression { , function_blocking_statement }
    | ( expression { , function_blocking_assign } )
```

Syntax 1-15—variable assignment syntax

The type of variable is explicitly specified. The variable may be assigned anywhere in the sequence, and reassigned later in the sequence. For every attempt, a new instance of the variable is created for the sequence. The variable value may be tested like any other System Verilog variable.

Hierarchical references to a local variable is not allowed.

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.

```
property e;
  int x;
  (valid_in, (x = pipe_in)) |-> ##5 (pipe_out1 == (x+1));
endproperty
```

Variables can be used in sequences or properties as examples show here.

```
sequence data_check;
  int x;
  a ##1 !a, x = data_in ##1 !b*[0:inf] ##1 b && (data_out == x);
endsequence
property data_check_p
  int x;
  a ##1 !a, x = data_in |> !b*[0:inf] ##1 b && (data_out == x);
endproperty
```

Furthermore, one can write local variables on repeated sequences and accomplish accumulation of values, as in the following example.

```
sequence rep_v;
  int x;
  `true, x = 0 ##0
  (!a [* 0:inf] ##1 a, x = x+data)[*4] ##1 b ##1 c && (data_out == x);
endsequence
```

There are special considerations on using local variables in parallel branches using operators **or**, **and**, and **intersect**.

- 1) Variables assigned on parallel threads cannot be accessed in sibling threads. For example,


```
sequence s4;
  int x;
  (a ##1 b, (x = data) ##1 c) || (d ##1 (e==x)); // illegal
endsequence
```
- 2) In the case of **or**, it is the intersection of the variables (names) that passes on past **or**. More precisely, a local variable passes on past the **or** if and only if it passes through both branches of **or**. All succeeding threads out of **or** branches continue as separate threads, carrying with them their own latest samplings of the local variables. These threads do not have to have consistent valuations for the local variables. For example,


```
sequence s5;
  int x,y;
  ((a ##1 b, x = data, y = data ##1 c) or (d ##1 `true, x = data ##0 (e==x))) ##1 (y==data2);
  // illegal since y is not in the intersection
endsequence
sequence s6;
```

```
int x,y;
((a ##1 b, x = data, y = data1 ##1 c) or (d ##1 'true, x = data ##0 (e==x))) ##1 (x==data2);
// legal since x is in the intersection
endsequence
```

- 3) In the case of **and** and **intersect**, the symmetric difference of the local variables that are sampled in the two joining threads passes on past the join. More precisely, a local variable that passes through at least one branch of the join will be passed on past the join unless it is blocked. A local variable is blocked from passing on past the join if either
- The local variable is sampled in and passes through each branch of the join. Or,
 - The local variable is blocked from passing through at least one of the branches of the join.
- The value passed on is the latest sampled value. The two joining threads are merged into one thread at the join.

```
sequence s7;
int x,y;
((a ##1 b, x = data, y = data1 ##1 c) and (d ##1 'true, x = data ##0 (e==x))) ##1 (x==data2);
// illegal since x common to both threads
endsequence
sequence s8;
int x,y;
(a ##1 b, x = data, y = data1 ##1 c) and (d ##1 'true, x = data ##0 (e==x)) ##1 (y==data2);
// legal since y is in the difference
endsequence
```

- 4) The intersection and difference of the sets of names should be computed statically at compile time.

1.9 System functions

ADD DAS FUNCTIONS

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 (expression [, number_of_ticks])

Syntax 1-16—\$past function syntax

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 sampled value of the expression that was present `number_of_ticks` prior to the time of evaluation of **\$past**.

If the specified clock tick in the past is before the start of simulation, the returned value from the **\$past** function is 'x'.

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

\$countones (expression)

Syntax 1-17—\$countones function syntax

The 'x' and 'z' value of a bit is not counted towards the number of ones.

1.10 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, an **assert** or **cover** statement 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.

```

property_declaration ::=
    property property_identifier [ property_formal_list ] ;
    { assertion_variable_declaration }
    property_spec ;
    endproperty [: property_identifier ]
property_formal_list ::=
    ( formal_list_item { , formal_list_item } )
property_spec ::=
    [ clocking_event ] [ disable iff ( expression ) ] [ not ] property_expr
    | [ disable iff ( expression ) ] [ not ] multi_clock_property_expr
property_expr ::=
    sequence_expr
    | sequence_expr |-> [ not ] sequence_expr
    | sequence_expr |=> [ not ] sequence_expr
multi_clock_property_expr ::=
    multi_clock_sequence
    | multi_clock_sequence |=> [ not ] multi_clock_sequence
property_instance ::=
    property_identifier [ ( actual_arg_list ) ]
assertion_variable_declaration ::=
    data_type list_of_variable_identifiers

```

Syntax 1-18—property construct syntax

A **property** is declared with optional formal arguments, like a **sequence** 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 result of property evaluation is either true or false. If the property is just a sequence, the result of a sequence for every evaluation attempt is true or false. This is accomplished by implicitly transforming `sequence_expr` to **first_match**(`sequence_expr`). That is, as soon as a match of `sequence_expr` is determined, the result is considered to be true, and no other matches are required for that evaluation attempt. However, if the property is an implication, then the semantics of implication determine whether the property is true or false.

The **disable iff** clause allows you to specify asynchronous resets. For a particular attempt, if the 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.

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 or 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.

This allows for the following examples:

```
property rule1;
  @(posedge clk) a |-> b ##1 c ##1 d;
endproperty
property rule2;
  disable iff (foo) not
  @(clkdiv) a |-> b ##1 c ##1 d;
endproperty
```

A property can optionally specify the clock.

A property can be referenced by its name. A hierarchical name can be used consistent with the System Verilog naming conventions. Like sequence declarations, variables used within a property that are not formal arguments to the property are resolved hierarchically from the scope in which the property is declared.

1.11 Multiple clock support

Multiple clock sequences and properties can be specified using the following syntax.

```
sequence_spec ::=
    multi_clock_sequence
    | sequence_expr
multi_clock_sequence ::=
    clocked_sequence { ## clocked_sequence }
clocked_sequence ::=
    clocking_event sequence_expr
multi_clock_property_expr ::=
    multi_clock_sequence
    | multi_clock_sequence | => [ not ] multi_clock_sequence
```

Two cases are allowed:

- 1) Concatenation of two sequences, where each sequence may have a different clocks
- 2) The antecedent of an implication on one clock, while the consequent on another clock

The multi-clock concatenation operator **##** synchronizes between the two clocks.

```
@(posedge clk) a ## @(posedge clk1) b
```

When signal a matches at clock clk, **##** moves the time to the nearest clock tick of clk1 after the match. At the first clock tick of clk1, it matches b. If the two clocks, clk and clk1, are identical, then the above sequence is equivalent to:

```
@(posedge clk) a ##1 b
```

For two sequences, such as

```
@(posedge clk) s ## @(posedge clk1) s1
```

For every match of *s* at clock *clk*, *##* moves the time to the first clock tick of *clk1*. From that first tick of *clk1*, *s1* is matched.

Multi-clock implication is only allowed with the non-overlapping implication. The semantics are similar to the sequence concatenation with *##*. Whenever there is a match of the antecedent sequence, time is advanced to the nearest clock tick of the clock of the consequent sequence. The consequent is then evaluated for satisfaction.

Following are examples of multi-clock specifications.

```
sequence s1;
  @(posedge clk1) a ##1 b; // single clock sequence
endsequence
sequence s2;
  @(posedge clk2) c ##1 d; // single clock sequence
endsequence
```

1) multiple clock sequence

```
sequence mult_s;
  @(posedge clk) a ## @(posedge clk1) s1 ## @(posedge clk2) s2;
endsequence
```

2) property with a multiple clocked sequence

```
property mult_p1;
  @(posedge clk) a ## @(posedge clk1) s1 ## @(posedge clk2) s2;
endproperty
```

3) property with a named multi-clocked sequence

```
property mult_p2;
  mult_s;
endproperty
```

4) property with multiple clocked implication

```
property mult_p3;
  @(posedge clk) a ## @(posedge clk1) s1 | => @(posedge clk2) s2;
endproperty
```

5) property with named sequences at different clocks. In this case, if *s1* contains a clock, then it must be identical to (posedge *clk1*). Similarly, if *s2* contains a clock, it must be identical to (posedge *clk2*).

```
property mult_p5;
  @(posedge clk1) s1 | => @(posedge clk2) s2;
endproperty
```

6) property with implication, where antecedent and consequent are named multi-clocked sequences

```
property mult_p6;
  mult_s | => mult_s;
endproperty
```

1.11.1 Detecting and using endpoint of a sequence in multi-clock context

To detect the end point of a sequence when the clock of the source sequence is different than the desalination sequence, method **matched** on the source sequence is used. 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 method **ended**, when the clocks of the source and destination sequences are the same, while method **matched** is used when the clocks are different.

```
boolean_expr_op ::=
    seq_name.matched
```

Syntax 1-19—matched operator syntax

matched is a method on a sequence which return true or false. Unlike **ended**, **matched** uses synchronization between the two clock, by storing the result of the source sequence match until the arrival of the first destination clock tick after the match. When method **matched** is applied, it tests whether the source sequence has reached the end point at that particular point in time. The result of **matched** does not depend upon the starting point of the source sequence.

Like **ended**, **matched** can be used directly on sequences that do not have formal arguments.

An example is shown below:

```
sequence e1;
    @(posedge clk) $rose(ready) ##1 proc1 ##1 proc2 ;
endsequence
sequence e2;
    @(posedge sysclk) reset ##1 inst ##1 e1.matched [*->1] ##1 branch_back;
endsequence
```

In this example sequence expression source sequence e1 is evaluated at clock clk, while the desalination sequence e2 is evaluated at clock sysclk. In e2, the end point of e1 is tested to occur sometime after the occurrence of inst. Notice that method **matched** only tests for the end point of e1, and has no bearing on the starting point of e1.

1.12 Concurrent Assertions

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

- **assert** to specify the property as a checker to ensure that the property holds for the design
- **cover** to monitor the property evaluation for coverage

```
concurrent_assert_statement ::=
    assert property ( property_spec) action_block
    | assert property ( property_instance) action_block
concurrent_cover_statement ::=
    cover property ( property_spec) statement_or_null
    | cover property ( property_instance) statement_or_null
```

Syntax 1-20—Verification directive syntax

The **assert** statement is used to enforce a property as a checker. When the property for the **assert** statement is evaluated to be true, the pass statements of the action block are executed. Otherwise, the fail statements of the action block are executed. For example,

```
property abc(a,b,c);
    disable iff (a==2) not @clk (b ##1 c);
endproperty
env_prop: assert property (abc(rst,in1,in2)) pass_stat else fail_stat;
```

When no action is needed, a null statement (i.e.) is specified. If no statement is specified for **else**, then **\$error** is used as the statement when the assertion fails.

The action_block shall not include any concurrent **assert** or **cover** statement. The action_block, however, may contain immediate assertion statements.

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

The assert and cover statements can be referenced by its optional name. A hierarchical name can be used consistent with the System Verilog naming conventions. When a name is not provided, a tool shall assign a name to the statement for the purpose of reporting.

System tasks are provided to control assertions, and limit them to part of the design or part of the simulation time as described in Section 22.6.

EDITOR'S NOTE: Need cross ref to section 22.6

Coverage results are divided into two: coverage for properties, coverage for sequences.

For sequence coverage, the statement appears as:

cover property ([clocking_event] sequence_instance) statement_or_null, or

cover property ([clocking_event] sequence_expr) statement_or_null, or

For property coverage, the statement appears as

cover property (property_instance) statement_or_null, or

cover property ([clocking_event] property_modifier property_expr) statement_or_null, or

cover property ([clocking_event] property_implication) statement_or_nul

Results of coverage statement for a property shall contains:

- Number of times attempted
- Number of times succeeded
- Number of times failed
- Number of times succeeded because of vacuity (limited)
- Each attempt with attemptID and time

— Each success/failure with attemptId and time

statement_or_null gets executed every time a property succeeds.

Vacuity rules are applied only when implication operator is used.

A property succeeds non-vacuously only if the consequent of the implication contributes to the success.

Results of coverage for a sequence shall include:

— Number of times attempted

— Number of times matched (each attempt can generate multiple matches)

— 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.

— Each attempt with attemptId and time

— Each match with clock step, attemptId and time

1.12.1 Using concurrent assertion statements outside of procedural code

A concurrent assertion statements can be used directly within a module as a *module_item* or within *interface* as an *interface_item*. A concurrent assertion statement is either an **assert** or a **cover** statement. Such a concurrent assertion statement uses the “always” semantics.

The following two forms are equivalent:

```
assert property ( property_spec) action_block
```

```
always assert property ( property_spec) action_block;
```

Similarly, the following two forms are equivalent:

```
cover property ( property_spec) statement_or_null
```

```
| always cover property ( property_spec) statement_or_null
```

For example:

```
module top(input bit clk);  
  reg a,b,c;  
  property rule3;  
    @(posedge clk) a |-> b ##1 c;  
  endproperty  
  a1: g (rule3);  
  ...  
endmodule
```

rule3 is a property declared in module top. The assert statement a1 starts the checking the property from beginning to the end of simulation. The property is always checked. Similarly,

```
module top(input bit clk);  
  reg a,b,c;  
  sequence seq3;  
    @(posedge clk) b ##1 c;  
  endsequence  
  c1: cover property (seq3);  
  ...  
endmodule
```

```
endmodule
```

The cover statement `c1` start coverage of the sequence `seq3` from beginning to the end of simulation. The sequence is always monitored for coverage.

1.12.2 Embedding concurrent assertions in procedural code

A concurrent assertion statement can also be embedded in a procedural block as a `statement_item`. For example,

```
property rule;
  a ##1 b ##1 c;
endproperty

always @(posedge clk) begin
  <statements>;
  assert property (rule);
end
```

If the statement appears in an `always` block, the property is always monitored. If the statement appears in an initial block, then the monitoring is performed only on the first clock tick.

Two inferences are made from the procedural context: clock from the event control of an `always` block, and the enabling conditions.

A clock is inferred if the statement is placed in an `always` or `initial` block with an event control abiding by the following rules:

- clock to be inferred must be placed as the first term of the event control as an edge specifier (`posedge` expression or `negedge` expression)
- `clock_name` must not be used anywhere in the `always` or `initial` block

For example:

```
property r1;
  q != d;
endproperty
always @(posedge mclk)begin
  q <= d1;
  r1_p: assert property (r1);
end
```

The above property can be checked by writing statement `r1_p` outside the `always` block, and declaring the property with the clock as:

```
property r1;
  @ (posedge mclk)q != d;
endproperty
always @(posedge mclk)begin
  q <= d1;
end
r1p: assert property (r1);
```

If the clock is explicitly specified with a property, then it must be identical to the inferred clock, as shown below:

```
property r2;
  @(posedge mclk)(q != d);
endproperty
```

```
always @(posedge mclk) begin
    q <= d1;
    r2_p: assert property (r2);
end
```

Another inference made from the context is the enabling condition for a property. Such derivation takes place when a property is placed in an if-else block or a case block. The enabling condition assumed from the context is used as the antecedent of the property.

```
property r3;
    @(posedge sclk)(q != d);
endproperty
always @(posedge mclk)begin
    if (a) begin
        q <= d1;
        r3_p: assert property (r2);
    end
end
```

The above example is equivalent to:

```
property r3;
    @(posedge sclk)a |-> (q != d);
endproperty
r3_p: assert property (r3);
always @(posedge mclk)begin
    if (a) begin
        q <= d1;
    end
end
```

Similarly, enabling condition is also inferred from case statements.

```
property r4;
    @(posedge sclk)(q != d);
endproperty
always @(posedge mclk)begin
    case (a)begin
        1:begin q <= d1;
            r4p: assert property (r4);
        end
        default: q1 <= d1;
    endcase
end
```

The above example is equivalent to:

```
property r4;
    @(posedge sclk)(a==1) |-> (q != d);
endproperty
r4_p: assert property (r4);
always @(posedge mclk)begin
    case (a)begin
        1:begin q <= d1;
            end
        default: q1 <= d1;
    endcase
end
```

The enabling condition is inferred from procedural code inside an always or initial block with the following restrictions:

- 1) There must not be a preceding statement with timing control.
- 2) A preceding statement may not invoke a task call which contains a timing control on any statement.
- 3) The concurrent assertion statement may not be placed in a looping statement, immediately or in any nested scope of the looping statement.

1.13 Clock resolution

There are a number of ways to specify a clock for a property:

- `clocked_sequence`, for example
`property p1; not @(posedge clk) a ##2 b; endproperty`
`assert property (p1);`
- `sequence_instance`, for example
`sequence s2; @(posedge clk) a ##2 b; endsequence`
`property p2; not s2; endproperty`
`assert property (p2);`
- `property`, for example
`property p3; @(posedge clk) not a ##2 b; endproperty`
`assert property (p3);`
- contextually inferred clock from a procedural block, for example
`always @(posedge clk) assert property (not a ##2 b);`
- `clocking domain`, for example
`clocking master_clk @(posedge clk);`
`property p3; not a ##2 b; endproperty`
`endclocking`
`assert property (master_clk.p3);`
- `default clock`, for example
`default clocking master_clk @(posedge clk);`

For a multi-clocked assertion, the clocks are explicitly specified. No default clock or inferred clock is used. In addition, multi-clocked properties are not allowed to be defined within a clocking domain.

A multi-clocked property assert statement must not be embedded in procedural code where a clock is inferred. For example, following forms are not allowed.

```
always @(clk) assert property (mult_clock_prop);
initial @(clk) assert property (mult_clock_prop);
```

Now, we will discuss the rules for an assertion with one clock.

The clock for an assertion statement is determined in the decreasing order of priority

- 1) explicitly specified clock for the assertion
- 2) inferred clock from the context of the code when embedded
- 3) default clock, if specified

Sequences and properties specified in clocking domains resolve the clock by the following rules:

- 1) Event control of the clocking domain specifies the clock.

- 2) No explicit event control is allowed in any definition.
- 3) If a named sequence is used which is defined outside the clocking domain, its clock, if specified, must be identical to the clocking domain's clock.
- 4) Multi-clock properties are not allowed.

Resolution of clock for a sequence definition assumes that only one explicit event control can be specified. Also, the named sequences used in the sequence definition may or may not contain event control in their definitions.

```
sequence s;
  //sequence composed of two named sub-sequences
  @(posedge s_clk) e ##1 s1 ##1 s2 ##1 f;
endsequence
sequence s1;
  @(posedge clk1) a ##1 b; // single clock sequence
endsequence
sequence s2;
  @(posedge clk2) c ##1 d; // single clock sequence
endsequence
```

These example sequences are used in the following table to explain the rules for a sequence definition. The clock of any sequence when explicitly specified is indicated by X. The absence of a clock is indicated by a dash.

Table 1-1:

s_clk	clk1	clk2	Resolved clock	Semantic restriction
-	-	-	unclocked	-
X	-	-	s_clk	-
X	X	-	s_clk	s_clk and clk1 must be identical
X	X	X	s_clk	s_clk, clk1 and clk2 must be identical
X	-	X	s_clk	s_clk and clk2 must be identical
-	X	-	clk1	-
-	X	X	clk1	clk1 and clk2 must be identical
-	-	X	clk2	-

Once the clock for a sequence definition is determined, the clock of a property definition is resolved similar to the resolution for a sequence definition. A single clocked property assumes that only one explicit event control can be specified. Also, the named sequences used in the property definition may or may not contain event control in their definitions. The following table specifies the rules for property definition clock resolution. The property has the form:

```
property p;
  @(posedge p_clk) not s1 |> s2;
endproperty
```

p_clk is the property for the clock, clk1 is the clock for sequence s1 and clk2 is the clock for sequence s2. Same rules apply for operator |>.

p_clk	clk1	clk2	Resolved clock	Semantic restriction
-	-	-	unclocked	-
X	-	-	p_clk	-
X	X	-	p_clk	p_clk and clk1 must be identical
X	X	X	p_clk	p_clk, clk1 and clk2 must be identical
X	-	X	p_clk	p_clk and clk2 must be identical
-	X	-	clk1	-
-	X	X	clk1	clk1 and clk2 must be identical. If clk1 and clk2 are different for the case of operator =>, then it is considered a multi-clock implication
-	-	X	clk2	-

Resolution of clock for an assert statement is based on the following assumptions:

- **assert** can appear in always block, initial block or outside procedural context
- clock is inferred from an always or initial block
- default clock may be specified using default clocking domain

The following table specifies the rules for clock resolution when **assert** appears in an always or initial block, where i_clk is the inferred clock from an always or initial block, d_clk is the default clock, and p_clk is the property clock.

i_clk	d_clk	p_clk	Resolved clock	Semantic restriction
-	-	-	unclocked	Error. An assertion must have a clock
X	-	-	i_clk	-
-	X	-	d_clk	
-	-	X	p_clk	
X	-	X	i_clk	i_clk and p_clk must be identical
X	X	-	i_clk	-
-	X	X	p_clk	
-	-	X	p_clk	-

When the assert statement is outside any procedural block, there is no inferred clock. The rules for clock resolution are specified in the table below.

d_clk	p_clk	Resolved clock	Semantic restriction
-	-	unlocked	Error. An assertion must have a clock
X	-	d_clk	
-	X	p_clk	
X	X	p_clk	

1.14 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. Following are the goals of providing this feature.

- It allows verification engineers to verify with minimum changes to the design code/files.
- It allows a convenient mechanism to attach verification IP to a module or an instance.
- No semantic changes to the assertions are introduced due to this feature. It is equivalent to writing properties with XMRs.

With this feature, a user can bind a module, interface or program instance to a module or a module instance.

The syntax of the **bind** construct is:

```

bind_directive ::=
    bind module_instance_name bind_instantiation ;
module_instance_name ::=
    module_identifier
    | name_of_instance
bind_instantiation ::=
    program_instantiation
    | module_instantiation
    | interface_instantiation
    
```

Syntax 1-21—bind construct syntax

The bind directive can be specified in a module or \$root as a module_item.

A program contains non-design code (either testbench or properties) and executes in the reactive region, as explained in the scheduling semantics of System verilog.

Example of binding a program instance to a module:

```
bind cpu fpu_props fpu_rules_1(a,b,c) ;
```

- cpu is the name of module.
- fpu_props is the name of the program containing properties fpu_rules_1 is the program instance name.

- Ports (a, b,c) get bound to signals (a,b,c) of module cpu .
- Every instance of cpu gets the properties.

Example of binding a program instance to a specific instance of a module:

```
bind cpu1 fpu_props fpu_rules_1(a,b,c);
```

- cpu1 is the name of module instance (cpu1 is an instance of module of module cpu)
- fpu_props is the name of the program containing properties.
- fpu_rules_1 is the program instance name.
- Ports (a, b,c) get bound to signals (a,b,c) of module instance cpu1.
- Only cpu1 instance of cpu gets the properties.

By binding a program to a module or an instance, the program becomes part of the bound object. The names of assertion related declarations can be referenced using the System Verilog hierarchical naming conventions.

Binding of a module instance or an interface instance works the same way as described for programs above.

```
interface range(input clk,enable,input int minval,expr);
  property crange_en;
    @(posedge clk) enable |-> (minval <= expr);
  endproperty
  range_chk: assert property (crange_en);
endinterface

bind cr_unit range r1(c_clk,c_en,v_low,(in1&&in2));
```

In this example, interface range is instantiated in the module cr_unit. Effectively, every instance of module cr_unit will contain the interface instance r1.