Chapter 1

INTRODUCTION TO SVA

Understanding the Syntax

1.1 What is an Assertion?

An assertion is a description of a property of the design.

- If a property that is being checked for in a simulation does not behave the way we expect it to, the assertion fails.
- If a property that is forbidden from happening in a design happens during simulation, the assertion fails.

A list of the properties can be inferred from the functional specification of a design and can be converted into assertions. These assertions can be continuously monitored during functional simulations. The same assertions can also be re-used for verifying the design using formal techniques. Assertions, also known as monitors or checkers, have been used as a form of debugging technique for a very long time in the design verification process. Traditionally, they are written in a procedural language like Verilog. They can also be written in PLI and C/C++ programs. The following code shows a simple mutually asserted condition check written in Verilog, wherein signal "a" and signal "b" cannot be high at the same time. If they are, an error message is displayed.

```
`ifdef ma
if(a & b)
$display
```

```
("Error:Mutually asserted check failed\n");
`endif
```

This kind of a monitor is included only as part of the simulation and hence is included in the design environment only on a need basis. This can be accomplished with the **`ifdef** construct which enables conditional compilation of Verilog code.

1.2 Why use SystemVerilog Assertions (SVA)?

While Verilog language can be used to write certain checks easily, it has a few disadvantages.

1. Verilog is a procedural language and hence, does not have good control over time.
2. Verilog is a verbose language. As the number of assertions increase, it becomes very difficult to maintain the code.
3. The procedural nature of the language makes it difficult to test for parallel events in the same time period. In some cases, it is also possible that a Verilog checker might not capture all the triggered events.
4. Verilog language has no built-in mechanism to provide functional coverage data. The user has to produce this code.

SVA is a declarative language and is perfectly suited for describing temporal conditions. The declarative nature of the language gives excellent control over time. The language itself is very concise and is very easy to maintain. SVA also provides several built-in functions to test for certain design conditions and also provides constructs to collect functional coverage data automatically.

Example 1.1 shows a checker written both in Verilog and SVA. The checker verifies that if signal "a" is high in the current clock cycle, then signal "b" should be high within 1 to 3 clock cycles. Figure 1-1 shows the waveform of a sample simulation of the signals "a" and "b."

Example 1.1 Sample assertion written in Verilog and SVA

```
// Sample Verilog checker

always @(posedge a)
begin
```

```
repeat (1) @(posedge clk);
  fork: a_to_b

      begin
      @(posedge b)
      $display
      ("SUCCESS: b arrived in time\n", $time);
      disable a_to_b;
      end

      begin
      repeat (3) @(posedge clk);
      $display
      ("ERROR:b did not arrive in time\n", $time);
      disable a_to_b;
      end

   join
end

// SVA Checker

a_to_b_chk:
assert property
@(posedge clk) $rose(a) |-> ##[1:3] $rose(b));
```

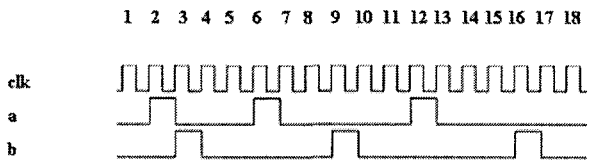

Figure 1-1. Waveform for sample assertion

Example 1.1 shows the advantages of SVA very clearly. SVA syntax is discussed in detail in this chapter. The checker represents a very simple protocol. It can be written in one line in SVA, although the same protocol description takes several lines in Verilog. Also, the error and success conditions need to be defined in Verilog explicitly, whereas the failure will automatically display an error message in SVA. Results of a sample simulation are shown below.

```
SUCCESS: b arrived in time 127
vtosva.a_to_b_chk:
started at 125s succeeded at 175s

SUCCESS: b arrived in time 427
vtosva.a_to_b_chk:
started at 325s succeeded at 475s

ERROR: b did not arrive in time 775
vtosva.a_to_b_chk:
started at 625s failed at 775s
        Offending '$rose(b)'
```

1.3 SystemVerilog Scheduling

The SystemVerilog language is defined to be an event based execution model. In each time slot, many events are scheduled to happen. This list of events follows the algorithm specified by the standard. By following this algorithm, the simulators can avoid any inconsistencies in the interactions between the design and testbench. There are three regions that are involved in the evaluation and execution of the assertions.

Preponed – Values are sampled for the assertion variables in this region. In this region, a net or variable cannot change its state. This allows the sampling of the most stable value at the beginning of the time slot.

Observed – All the property expressions are evaluated in this region.

Reactive – The pass/fail code from the evaluation of the properties are scheduled in this region.

Figure 1-2 shows a simplified SystemVerilog event schedule flow chart. To understand the SystemVerilog scheduling algorithm thoroughly, please refer to the SystemVerilog 3.1a LRM [1].

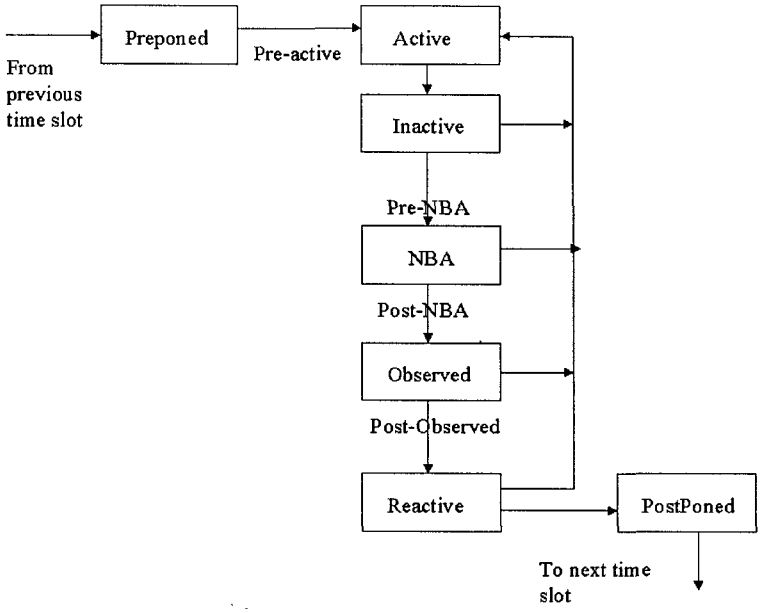

Figure 1-2. Simplified SV event schedule flow chart

1.4 SVA Terminology

There are two types of assertions defined in the SystemVerilog language: Concurrent assertions and Immediate assertions.

1.4.1 Concurrent assertions

- Based on clock cycles.
- Test expression is evaluated at clock edges based on the sampled values of the variables involved.
- Sampling of variables is done in the "preponed" region and the evaluation of the expression is done in the "observed" region of the scheduler.
- Can be placed in a procedural block, a module, an interface or a program definition.
- Can be used with both static (formal) and dynamic verification (simulation) tools.

A sample concurrent assertion is shown below.

```
a_cc: assert property(@(posedge clk)
                                not (a && b));
```

Figure 1-3 shows the results of the concurrent assertion a_cc. All successes are shown with an up arrow and all failures are shown with a down arrow. The key concept in this example is that the property is being verified on every positive edge of the clock irrespective of whether or not signal "a" and signal "b" changes.

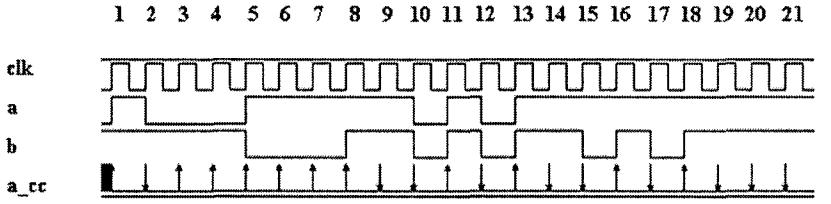

Figure 1-3. Waveform for a sample concurrent assertion

1.4.2 Immediate assertions

- Based on simulation event semantics.
- Test expression is evaluated just like any other Verilog expression within a procedural block. These are not temporal in nature and are evaluated immediately.
- Have to be placed in a procedural block definition.
- Used only with dynamic simulation.

A sample immediate assertion is shown below.

```
always_comb
begin
   a_ia: assert (a && b);
end
```

The immediate assertion a_ia is written as part of a procedural block and it follows the same event schedule of signal "a" and "b." The always block executes if either signal "a" or signal "b" changes. The keyword that

differentiates the immediate assertion from the concurrent assertion is "**property.**" Figure 1-4 shows the results of the immediate assertion a_ia.

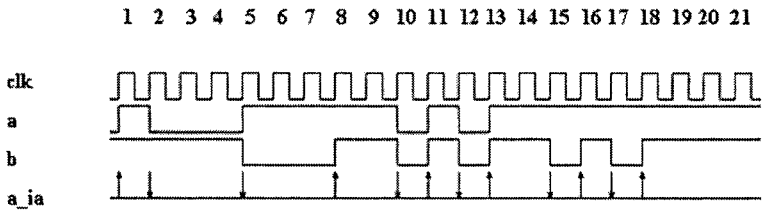

Figure 1-4. Waveform for a sample immediate assertion

1.5 Building blocks of SVA

In any design model, the functionality is represented by the combination of multiple logical events. These events could be simple boolean expressions that get evaluated on the same clock edge or could be events that evaluate over a period of time involving multiple clock cycles. SVA provides a key word to represent these events called "**sequence.**" The basic syntax of a **sequence** is as follows.

```
sequence name_of_sequence;
   < test expression>;
endsequence
```

A number of sequences can be combined logically or sequentially to create more complex sequences. SVA provides a key word to represent these complex sequential behaviors called "**property.**" The basic syntax of a **property** is as follows.

```
property name_of_property;
   < test expression >; or
   < complex sequence expressions >;
endproperty
```

The property is the one that is verified during a simulation. It has to be asserted to take effect during a simulation. SVA provides a key word called "**assert**" to check the property. The basic syntax of an **assert** is as follows.

```
assertion_name: assert property(property_name);
```

The steps involved in the creation of a SVA checker are shown in Figure 1-5.

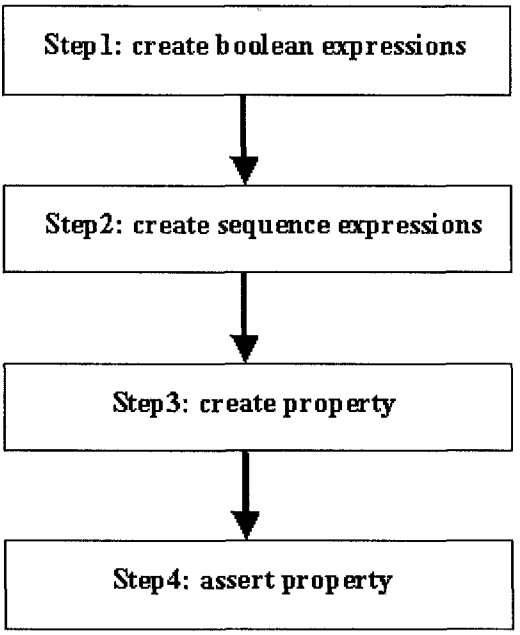

Figure 1-5. SVA Building blocks

1.6 A simple sequence

Sequence s1 checks that the signal "a" is high on every positive edge of the clock. If signal "a" is not high on any positive clock edge, the assertion will fail. Note that "a" is the same as "a==1'b1."

```
sequence s1;
  @(posedge clk) a;
endsequence
```

Figure 1-6 shows a sample waveform for signal "a" and how sequence s1 responds to this signal during simulation. Signal "a" goes to zero on the

positive edge of clock cycle 7. This change in value is sampled in clock cycle 8. Since concurrent assertions use the values sampled in the "preponed" region of the scheduler, in clock cycle 7, the most stable value of signal "a" sampled by the sequence s1 is 1. Hence, the sequence succeeds. In clock cycle 8, the sampled value of signal "a" is a 0 and hence the sequence fails. A success is denoted with an arrow pointing up and a failure is denoted with an arrow pointing down. Table 1-1 summarizes the sampled values of signal "a" on each clock cycle up to clock cycle 15.

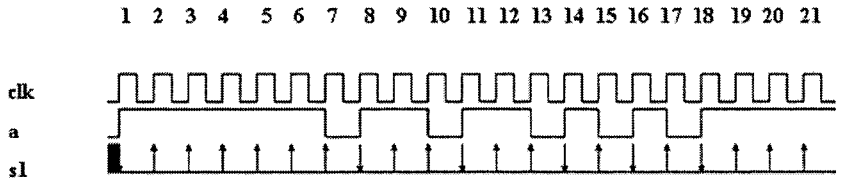

Figure 1-6. Waveform for simple sequence s1

Table 1-1. Evaluation table for sequence s1

Clock tick	Sampled value of signal "a"
1	0
2	1
3	1
4	1
5	1
6	1
7	1
8	0
9	1
10	1
11	0
12	1
13	1
14	0
15	1

1.7 Sequence with edge definitions

In sequence s1, the logical value of the signal was used. SVA also has built-in edge expressions that let the user monitor the transition of signal value from one clock cycle to the next. This allows one to check for the edge sensitivity of signals. Three of these useful built-in functions are shown below.

$rose (*boolean expression or signal_name*)

- This returns true if LSB of signal/expression changed to 1

$fell (*boolean expression or signal_name*)

- This returns true if LSB of signal/expression changed to 0

$stable (*boolean expression or signal_name*)

- This returns true if the value of the expression did not change

Sequence s2 checks that the signal "a" transitions to a value of 1 on every positive edge of the clock. If the transition does not occur, the assertion will fail.

```
sequence s2;
  @(posedge clk) $rose(a);
endsequence
```

Figure 1-7 shows how sequence s2 responds to the transition of signal "a." Marker 1 shows the first success of sequence s2. At clock cycle 1, the value of signal "a" goes from 0 to 1. At this clock, the sampled value of signal "a" within the sequence is 0. Before clock cycle 1, there is no history for signal "a" and hence the value is assumed to be "x." A transition of value from x to 0 is not a rising edge and hence the sequence fails. At clock cycle 2, the sampled value of signal "a" within the sequence is 1. A transition of value from 0 to 1 is a rising edge and hence, the sequence s2 succeeds in clock cycle 2. Another success is shown with marker 2 at clock cycle 9. Table 1-2 summarizes the transition of signal "a" over time until clock cycle 9 and how the sequence samples and updates the values.

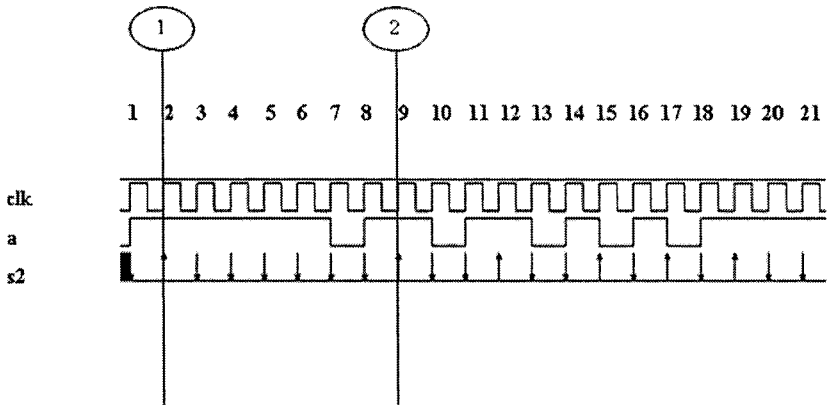

Figure 1-7. Waveform for simple sequence with edge definition

Table 1-2. Evaluation table for sequence s2

Clock Tick	Sampled value of "a" from the previous cycle	Sampled value of "a" in the current cycle	Sequence s2 - status
1	X	0	Fail
2	0	1	Success
3	1	1	Fail
4	1	1	Fail
5	1	1	Fail
6	1	1	Fail
7	1	1	Fail
8	1	0	Fail
9	0	1	Success

1.8 Sequence with logical relationship

Sequence s3 checks that on every positive edge of the clock, either signal "a" or signal "b" is high. If both the signals are low, the assertion will fail.

```
sequence s3;
  @(posedge clk)  a || b;
endsequence
```

Figure 1-8 shows how the sequence s3 responds to signal "a" and "b." Marker 1 shows that at clock cycle 12, the sampled values of both signals "a" and "b" are 0 and hence the sequence fails. The same is true for clock cycle 17 shown by marker 2. In all other clock cycles, either signal "a" or signal "b" has a value of 1 and hence the sequence succeeds in those clock cycles.

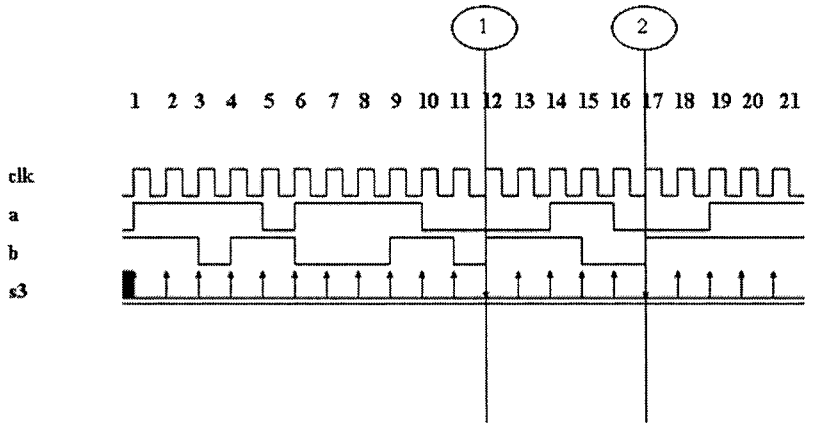

Figure 1-8. Waveform for sequence s3

1.9 Sequence Expressions

By defining formal arguments in a sequence definition, the same sequence can be re-used on other signals of a design that have similar behavior. For example, we can define a sequence as follows.

```
sequence s3_lib (a, b);
   a || b;
endsequence
```

The generic sequence s3_lib can be re-used on any two signals. For example, say we have two signals "req1" and "req2" and one of them should be asserted on the positive edge of a clock. We can write a sequence as follows.

```
sequence s3_lib_inst1;
   S3_lib(req1, req2);
```

```
endsequence
```

Some of the common properties that are normally present in designs can be developed as a library and re-used. For example, one-hot state machine checks, parity checks, etc. are good candidates for a checker library.

1.10 Sequences with timing relationship

Simple boolean expressions are checked on every clock edge. In other words, they are simple combinational checks. A lot of times, we are interested in checking events that take several clock cycles to complete. These are called "**sequential checks.**" ***In SVA, clock cycle delays are represented by a "##" sign. For example, ##3 means 3 clock cycles.***

Sequence s4 checks for the signal "a" being high on a given positive edge of the clock. If signal "a" is not high, then the sequence fails. If signal "a" is high on any given positive edge of clock, then signal "b" should be high 2 clock cycles after that. If signal "b" is not asserted after 2 clock cycles, the assertion fails. Note that the sequence begins when signal "a" is high on a positive edge of the clock.

```
sequence s4;
 @(posedge clk)  a ##2 b;
endsequence
```

Figure 1-9 shows how sequence s4 responds in a simulation. Table 1-3 summarizes the sampled values of signal "a" and signal "b" on every clock cycle.

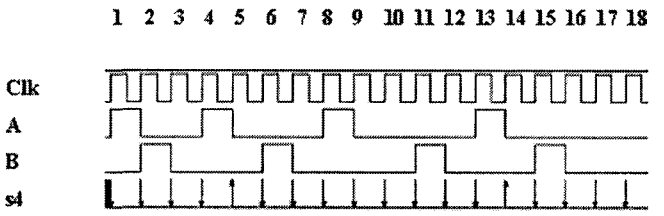

Figure 1-9. Waveform for sequence s4

Unlike the examples from the previous section, note that the start and end time of sequence s4 are not the same. If signal "a" is not high on any given clock cycle, then the sequence starts and fails on the same clock cycle. If signal "a" is high, then the sequence starts. The sequence succeeds after 2 clock cycles if signal "b" is high (clock 5 and clock 14). On the other hand, if signal "b" is not high after 2 clock cycles, then the sequence fails. Note that the success of a sequence is always represented in the figure at the starting point of the sequence.

Table 1-3. Evaluation table for sequence s4

Clock tick	**Sampled value of "a"**	**Sampled value of "b"**	**Valid start of s4**	**S4 status**
1	0	0	No	Fail
2	1	0	Yes	Fail (start at 2, end at 4)
3	0	1	No	Fail
4	0	0	No	Fail
5	1	0	Yes	Success (start at 5, end at 7)
6	0	0	No	Fail
7	0	1	No	Fail
8	0	0	No	Fail
9	1	0	Yes	Fail (start at 9, end at 11)
10	0	0	No	Fail
11	0	0	No	Fail
12	0	1	No	Fail
13	0	0	No	Fail
14	1	0	Yes	Success (start at 14, end at 16)
15	0	0	No	Fail
16	0	1	No	Fail
17	0	0	No	Fail

1.11 Clock definitions in SVA

A sequence or a property does not do anything by itself in a simulation. They have to be asserted to take effect as shown below.

```
sequence s5;
  @(posedge clk)  a ##2 b;
endsequence

property p5;
  s5;
endproperty

a5 : assert property(p5);
```

Note that the clock is specified in the sequence s5. While this is one way of relating a check to a clock there are also other ways of doing it. A clock can be specified in a sequence, in a property or even in an assert statement. The following code shows the clock defined in the property definition p5a.

```
sequence s5a;
  a ##2 b;
endsequence

property p5a;
  @(posedge clk)  s5a;
endproperty

a5a : assert property(p5a);
```

In general, it is a good idea to define the clocks in property definitions and keep the sequences independent of the clocks. This will help increase the re-use of the basic sequence definitions.

A separate property definition is not needed to assert a sequence. Since the assert statement calls a property, the expression to be checked can be called from the assert statement directly as shown below in assertion a5b.

```
sequence s5b;
  a ##2 b;
endsequence
```

```
a5b : assert property(@(posedge clk) s5b);
```

While we can call a sequence with a clock definition from within the assert statement, calling a property with a clock definition from within the assert statement is not allowed. This coding style is shown below in assertion a5c.

```
a5c : assert property(@(posedge clk) p5a); // Not
allowed
```

1.12 Forbidding a property

In all the examples shown so far, the property is checking for a true condition. A property can also be forbidden from happening. In other words, we expect the property to be false always. If the property is true, the assertion fails.

Sequence s6 checks that if signal "a" is high on a given positive edge of the clock, then after 2 clock cycles, signal "b" shall not be high. The keyword "**not**" is used to specify that the property should never be true.

```
sequence s6;
 @(posedge clk)  a ##2 b;
endsequence

property p6;
 not s6;
endproperty

a6 : assert property(p6);
```

Figure 1-10 shows how the checker a6 responds in a simulation. Note that the checker fails on two occasions (clock 5 and clock 14) as shown by markers 1 and 2. In both these clock cycles, the sequence that was forbidden happened and hence asserted a failure.

On the other hand, the checker passes on two occasions when there is a valid signal "a" (clock 2 and clock 9). For the checks that began in these clock cycles, signal "b" does not go high after two clock cycles and hence the checker succeeded. All other clock cycles wherein signal "a" was not high succeeded automatically. Table 1-4 summarizes the sampled values of signal "a" and signal "b" on each clock cycle.

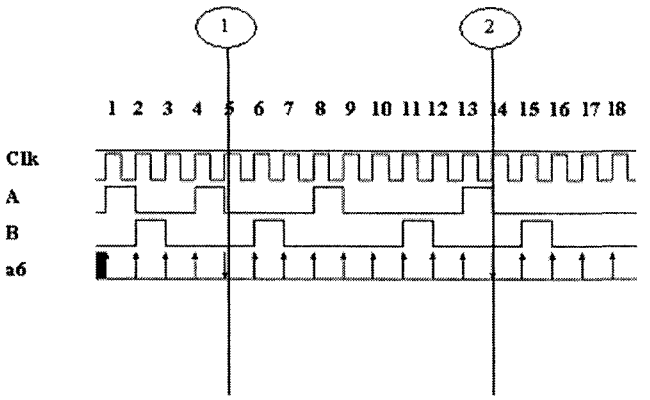

Figure 1-10. Waveform of SVA checker forbidding a property

Table 1-4. Evaluation table for property p6

Clock tick	Sampled value of "a"	Sampled value of "b"	Valid start of s6	a6 status
1	0	0	Yes	Success (same clock)
2	1	0	Yes	Success (start at 2, end at 4)
3	0	1	Yes	Success (same clock)
4	0	0	Yes	Success (same clock)
5	1	0	Yes	Fail (start at 5, end at 7)
6	0	0	Yes	Success (same clock)
7	0	1	Yes	Success (same clock)
8	0	0	Yes	Success (same clock)
9	1	0	Yes	Success (start at 9, end at 11)
10	0	0	Yes	Success (same clock)
11	0	0	Yes	Success (same clock)
12	0	1	Yes	Success (same clock)
13	0	0	Yes	Success (same clock)
14	1	0	Yes	Fail (start at 14, end at 16)
15	0	0	Yes	Success (same clock)
16	0	1	Yes	Success (same clock)
17	0	0	Yes	Success (same clock)

1.13 A simple action block

The SystemVerilog language is defined such that, every time an assertion check fails, the simulator is expected to print out an error message by default. The simulator need not print anything upon a success of an assertion. A user can also print a custom error or success message using the "**action block**" in the assert statement. The basic syntax of an **action block** is shown below.

```
assertion_name :
     assert property(property_name)
                <success message> ;
          else
                <fail message>;
```

The checker a7 shown below uses simple display statements in the action block to print successes and failures.

```
property p7;
  @(posedge clk)  a ##2 b;
endproperty

a7 : assert property(p7)
    $display("Property p7 succeeded\n");
    else
    $display("Property p7 failed\n");
```

The action block is not just limited to displaying success and failure. It can be used for other applications such as controlling the simulation environment and gathering functional coverage data. These topics will be discussed in detail in Chapter 2.

1.14 Implication operator

In the property p7, the following can be noticed.

1. The property looks for a valid start of the sequence on every positive edge of the clock. In this case, it looks for signal "a" to be high on every positive clock edge.
2. If signal "a" is not high on any given positive clock edge, an error is issued by the checker. This is not a valid error message since we are not interested in just checking for a specific level on signal "a." This

error just means that we did not get a valid starting point for the checker at this clock. While these errors are benign, they can log a lot of error messages over time, since the check is performed on every clock edge. To avoid these errors, some kind of gating technique needs to be defined, which will ignore the check if a valid starting point is not present.

SVA provides a technique to achieve this goal. This technique is called "**Implication**." Implication is equivalent to an if-then structure. The left hand side of the implication is called the "**antecedent**" and the right hand side is called the "**consequent**." The antecedent is the gating condition. If the antecedent succeeds, then the consequent is evaluated. If the antecedent does not succeed, then the property is assumed to succeed by default. This is called a "vacuous success." While implication avoids unnecessary error messages, it can produce vacuous successes. ***The implication construct can be used only with property definitions. It cannot be used in sequences.***

There are 2 types of implication: Overlapped implication and Non-overlapped implication.

1.14.1 Overlapped implication

Overlapped implication is denoted by the symbol |->. If there is a match on the antecedent, then the consequent expression is evaluated in the same clock cycle. A simple example is shown below in property p8. This property checks that, if signal "a" is high on a given positive clock edge, then signal "b" should also be high on the same clock edge.

```
property p8;
  @(posedge clk)  a |-> b;
endproperty

a8 : assert property(p8);
```

Figure 1-11 shows how the assertion a8 responds in a simulation. Table 1-5 summarizes the sampled values of signal "a" and signal "b" and the status of the assertion. There are 3 types of results shown in the table. A real success is one where a valid high on signal "a" was detected, and at the same clock edge a valid high on signal "b" was detected. A vacuous success is one where signal "a" was not high and the assertion succeeded by default. A failure is one where a valid high on signal "a" was detected and at the same clock edge a valid high on signal "b" was not detected high.

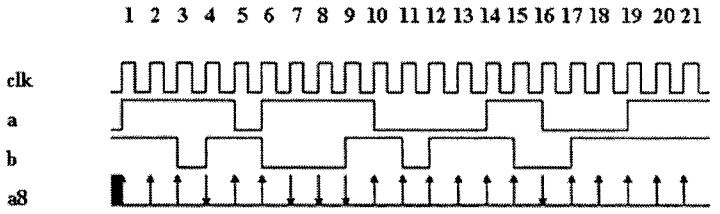

Figure 1-11. Waveform for property p8

Table 1-5. Evaluation table for property p8

Clock Tick	Sampled value of "a"	Sampled value of "b	A8 status
1	0	1	Vacuous success
2	1	1	Real Success
3	1	1	Real Success
4	1	0	Fail
5	1	1	Real Success
6	0	1	Vacuous success
7	1	0	Fail
8	1	0	Fail
9	1	0	Fail

1.14.2 Non-overlapped implication

Non-overlapped implication is denoted by the symbol |=>. If there is a match on the antecedent, then the consequent expression is evaluated in the next clock cycle. A delay of one clock cycle is assumed for the evaluation of the consequent expression. A simple example is shown below in property p9. This property checks that, if signal "a" is high on a given positive clock edge, then signal "b" should be high on the next clock edge.

```
property p9;
 @(posedge clk)  a |=> b;
endproperty

a9 : assert property(p9);
```

Figure 1-12 shows how the assertion a9 responds in a simulation. Table 1-6 summarizes the sampled values of signal "a" and signal "b" and the status of the assertion. Note that this assertion starts in the current clock cycle and succeeds in the next clock cycle if it is a real success. Similarly, if there is a valid start for the property (high on signal "a"), the property fails in the next clock cycle if signal "b" is not high in that clock cycle.

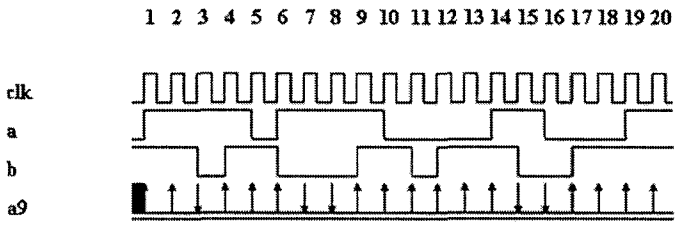

Figure 1-12. Waveform for property p9

Table 1-6. Evaluation table for property p9

Clock Tick	Sampled value of "a"	Sampled value of "b	a9 status
1	0	1	Vacuous success
2	1	1	Real success (start at 2, end at 3)
3	1	1	Fail (start at 3, end at 4)
4	1	0	Real success (start at 4, end at 5)
5	1	1	Real success (start at 5, end at 6)
6	0	1	Vacuous success
7	1	0	Fail (start at 7, end at 8)
8	1	0	Fail (start at 8, end at 9)
9	1	0	Real success (start at 9, end at 10)

1.14.3 Implication with a fixed delay on the consequent

Property p10 checks that if signal "a" is high in a given positive clock edge, then signal "b" should be high after 2 clock cycles. A similar check was shown before without the use of the implication operator. By using the implication, all the false errors are removed. A check for the consequent (signal "b") is performed only if there is a valid start for the property (high

on signal "a"). Figure 1-13 shows a sample simulation of the property p10. Table 1-7 summarizes the sampled values of the signals involved in property p10.

```
property p10;
  @(posedge clk)  a |-> ##2 b;
endproperty

a10 : assert property(p10);
```

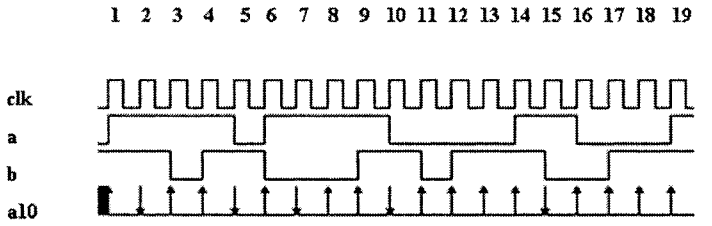

Figure 1-13. Waveform for property p10

Table 1-7. Evaluation table for property p10

Clock Tick	Sampled value of "a"	Sampled value of "b	a10 status
1	0	1	Vacuous success
2	1	1	Fail (start at 2, end at 4)
3	1	1	Success (start at 3, end at 5)
4	1	0	Success (start at 4, end at 6)
5	1	1	Fail (start at 5, end at 7)
6	0	1	Vacuous success
7	1	0	Fail (start at 7, end at 9)
8	1	0	Success (start at 8, end at 10)
9	1	0	Success (start at 9, end at 11)

1.14.4 Implication with a sequence as an antecedent

Property p10 has a signal in the antecedent position. It is also possible to have a sequence definition in the antecedent. In this case, a check for the consequent sequence or Boolean expression is performed only if the sequence in the antecedent succeeds. Sequence s11a checks that in any given positive clock edge, if signal "a" and signal "b" are detected to be high, then one clock cycle later, signal "c" should be high. Sequence s11b checks that, after 2 clock cycles from the current positive edge of the clock, signal "d" should be low. The final property checks that, if sequence s11a succeeds, then a check for sequence s11b is performed. If a valid sequence s11a is not detected, then the sequence s11b is not checked for and the property succeeds vacuously.

```
sequence s11a;
 @(posedge clk)   (a && b) ##1 c;
endsequence

sequence s11b;
 @(posedge clk)   ##2  !d;
endsequence

property p11;
 s11a |-> s11b;
endpeoperty

a11 : assert property(p11);
```

Figure 1-14 shows how the assertion a11 behaves in a simulation. The markers 1s and 1e show the start and end of a successful property evaluation. The markers 2s and 2e show the start and end of a failure. At clock cycle 11, both signal "a" and signal "b" are detected high. In clock cycle 12, signal "c" is high and hence the antecedent of the implication succeeds. This means that, 2 clock cycles from now, which is clock cycle 14, signal "d" should be low. But in the sample waveform signal "d" is a high and hence the property fails.

All the vacuous successes are shown with a simple straight line. The markers 3s and 3e show the start and end of a successful property evaluation. The expression "a && b" is evaluated to be true in clock cycle 17 and one clock cycle later, the signal "c" is high, as expected. Hence, at clock cycle 18, the sequence s11a succeeds. The signal "d" is expected to be low 2 clock

cycles from here and it is low as expected. Hence, the property succeeds at clock cycle 20.

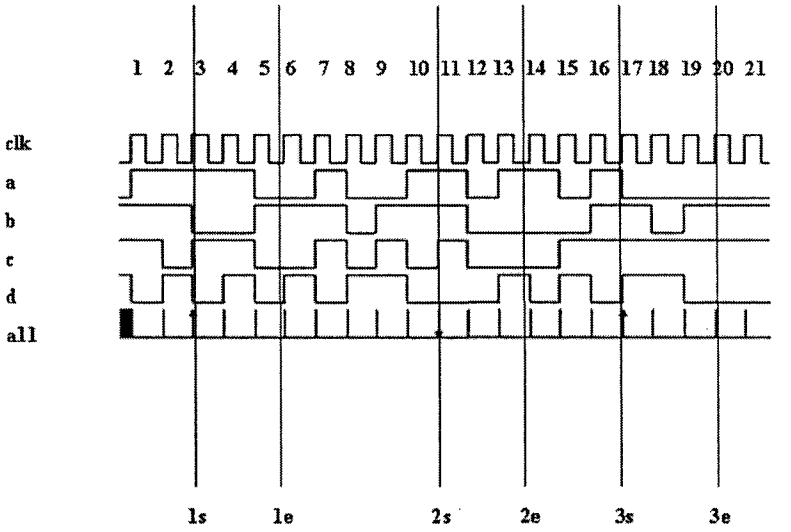

Figure 1-14. Waveform for property p11

1.15 Timing windows in SVA Checkers

So far, the examples shown with delays have a fixed delay greater than 0. In the next few examples, different ways of specifying delays will be discussed.

Property p12 checks whether the boolean expression "a **&&** b" is true on any given positive edge of the clock. If it is true, then within 1 to 3 clock cycles, the signal "c" should be high. SVA allows specifying a timing window for the consequent to match. The value specified in the left hand side of the timing window should be less than the value specified in the right hand side of the timing window. The left hand side can also have a value of 0. If 0 is specified in the left hand side, it means that the consequent must be checked starting from the same clock edge at which the antecedent succeeded.

```
property p12;
  @(posedge clk)  (a && b) |->  ##[1:3] c;
endproperty

a12 : assert property(p12);
```

Figure 1-15 shows how the property p12 responds in a simulation. Whenever a timing window is specified, multiple threads get kicked off for all possible matches in every clock edge. The property gets executed as three separate threads as follows.

```
(a && b) |->  ##[1] c or
(a && b) |->  ##[2] c or
(a && b) |->  ##[3] c
```

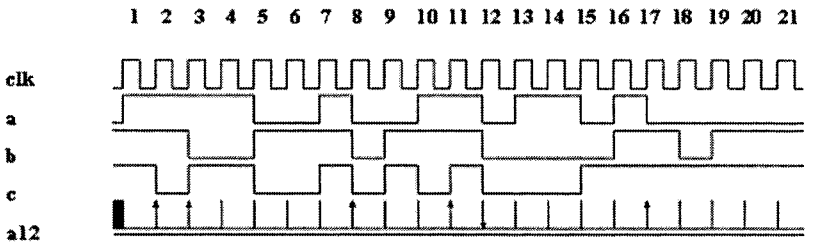

Figure 1-15. Waveform for p12

The property has 3 chances to succeed. All the three threads have the same starting point but the first thread that succeeds will make the property succeed. ***Also note that, there can be only one valid start on any give positive edge of the clock, but there can be multiple valid endings.*** This happens due to the fact that each valid start has 3 possible chances to succeed.

Table 1-8 summarizes the sampled values of all signals involved in the evaluation of the property. On a given positive clock edge, if signal "a" and signal "b" are both not high, then the property succeeds vacuously. On the other hand, on a given positive clock edge, if signal "a" and signal "b" are both high, then there is a valid start for the property. If signal "c" is not detected high in the next 1 to 3 clock cycles, the property fails.

Note that there is a valid start of the property detected on both clock cycle 2 and 3. Both of these valid starts succeed in clock cycle 4. The check that started at clock cycle 2 detected a high on signal "c" after 2 clock cycles. The check that started at clock cycle 3 detected a high on signal "c" after 1 clock cycle. Both of these are valid conditions and hence they succeed. There is also a valid start on clock cycle 12. The property checks for a high on signal "c" on clock cycles 13, 14 and 15. Since signal "c" remained low in all three possible clock cycles, the check failed.

Table 1-8. Evaluation table for property p12

Clock tick	**Sampled value of "a"**	**Sampled value of "b"**	**Sampled value of "c"**	**Valid start of p12**	**a12 status**
1	0	1	1	No	Vacuous success
2	1	1	1	Yes	Real Success (start at 2, end at 4)
3	1	1	0	Yes	Real Success (start at 3, end at 4)
4	1	0	1	No	Vacuous success
5	1	0	1	No	Vacuous success
6	0	1	0	No	Vacuous success
7	0	1	0	No	Vacuous success
8	1	1	1	Yes	Real Success (start at 8, end at 10)
9	0	0	0	No	Vacuous success
10	0	1	1	No	Vacuous success
11	1	1	0	Yes	Real Success (start at 11, end at 12)
12	1	1	1	Yes	Fail (start at 12, end at 15)
13	0	0	0	No	Vacuous success
14	1	0	0	No	Vacuous success
15	1	0	0	No	Vacuous success
16	0	0	1	No	Vacuous success
17	1	1	1	Yes	Real Success (start at 17, end at 18)

1.15.1 Overlapping timing window

Property p13 is similar to property p12. The main difference between the two is that the consequent of property p13 will be checked in the same clock edge in which the antecedent has a valid match.

```
Property p13;
 @(posedge clk)  (a && b) |->  ##[0:2] c;
endproperty

a13 : assert property(p13);
```

Figure 1-16 shows how property p13 responds in a simulation. The main difference in the response when compared to property p12 is that, the valid start that happens in clock cycle 12 succeeds. This succeeds because of the overlap in checking. The value of signal "c" is detected high in the same clock edge as the valid match on the antecedent.

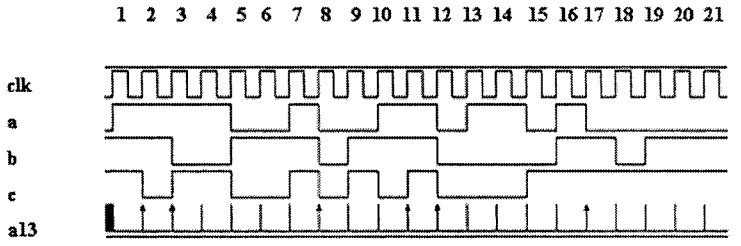

Figure 1-16. Waveform for property p13

1.15.2 Indefinite timing window

The upper limit of the timing window specified in the right hand side can be defined with a "$" sign which implies that there is no upper bound for timing. This is called the "**eventuality**" operator. The checker will keep checking for a match until the end of simulation. This is not a very efficient way of writing SVA since this has a huge impact on the simulation performance. It is best to always have a defined upper value in the timing window.

Property p14 checks that on a given positive edge of clock, signal "a" is high. If so, then signal "b" will be high eventually starting from the next clock cycle and after that, signal "c" will be high eventually starting at the same clock cycle in which signal "b" was high.

```
property p14;
  @(posedge clk)  a |->  ##[1:$] b ##[0:$] c;
endproperty

a14 : assert property(p14);
```

Figure 1-17 shows how property p14 reacts in a simulation. Table 1-9 summarizes the sampled values of the signals and the status of the assertion a14. Note that the real successes can take any number of clock cycles to finish. If there is a valid start and if either signal "b" or signal "c" does not match before the end of the simulation, these checks are reported as "**incomplete checks.**" Since overlap is allowed in the matching of signal "b" and signal "c," the whole check can finish in one clock cycle. Clock cycle 17 shows such a condition, wherein signal "a" was detected high on clock cycle 17 and both signal "b" and signal "c" were detected high on clock cycle 18.

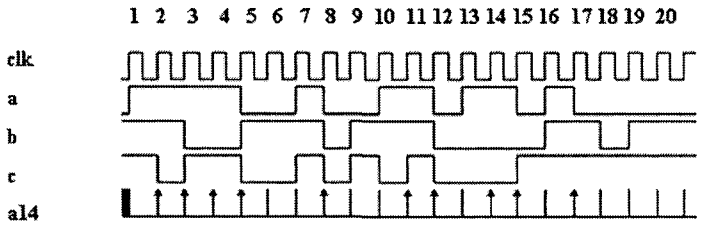

Figure 1-17. Waveform for property p14

Table 1-9. Evaluation table for p14

Clock tick	Sampled value of "a"	Sampled value of "b"	Sampled value of "c"	Valid start of p14	a14 status
1	0	1	1	No	Vacuous success
2	1	1	1	Yes	Real success (start at 2, end at 4)
3	1	1	0	Yes	Real success (start at 3, end at 8)

Clock tick	Sampled value of "a"	Sampled value of "b"	Sampled value of "c"	Valid start of p14	a14 status
4	1	0	1	Yes	Real success (start at 4, end at 8)
5	1	0	1	Yes	Real success (start at 5, end at 8)
6	0	1	0	No	Vacuous success
7	0	1	0	No	Vacuous success
8	1	1	1	Yes	Real success (start at 8, end at 10)
9	0	0	0	No	Vacuous success
10	0	1	1	No	Vacuous success
11	1	1	0	Yes	Real success (start at 11, end at 12)
12	1	1	1	Yes	Real success (start at 12, end at 17)
13	0	0	0	No	Vacuous success
14	1	0	0	Yes	Real success (start at 14, end at 17)
15	1	0	0	Yes	Real success (start at 15, end at 17)
16	0	0	1	No	Vacuous success
17	1	1	1	Yes	Real success (start at 17, end at 18)

1.16 The "ended" construct

The sequences defined so far use simple concatenation mechanism. In other words, multiple sequences were combined together over time by using the starting point of the sequence as the synchronization point. SVA provides another mechanism to concatenate sequences wherein the ending point of the sequence is used as a synchronization point. This is expressed by attaching the keyword "**ended**" to a sequence name. For example s.**ended** means the ending point of the sequence. The keyword **ended** stores a boolean value true or false depending on whether the sequence matched on that particular clock edge. ***This boolean value of the s.ended is available only in the same clock cycle.***

Sequence s15a and s15b are two 2 simple sequences that take more than 1 clock cycle to match. Property p15a checks that sequence s15a and sequence s15b match with a delay of one clock cycle in between them. Property p15b checks the same protocol but by using the keyword **ended**. In

this case, the end point of the sequences does the synchronization. Since the endpoints are used, a delay of 2 clock cycles is defined between the 2 sequences.

```
sequence s15a;
  @(posedge clk) a ##1 b;
endsequence

sequence s15b;
  @(posedge clk) c ##1 d;
endsequence

property p15a;
  s15a |=> s15b;
endproperty

property p15b;
  s15a.ended |-> ##2 s15b.ended;
endproperty

a15a: assert property(p15a);
a15b: assert property(p15b);
```

Figure 1-18 shows how properties p15a and p15b react in a simulation. Table 1-10 summarizes the status of the assertions a15a and a15b. The first real success for assertion a15a happens at clock cycle 2. The check becomes active at clock cycle 2 when signal "a" is detected high. The check completes at clock cycle 5 when signal "d" is detected high. The first real success for assertion a15b occurs at clock cycle 3. The check becomes active at clock cycle 3 when the sequence s15a matches or in other words, signal "b" is detected high. The check completes at clock cycle 5 when the sequence s15b matches or in other words, when signal "d" is detected high.

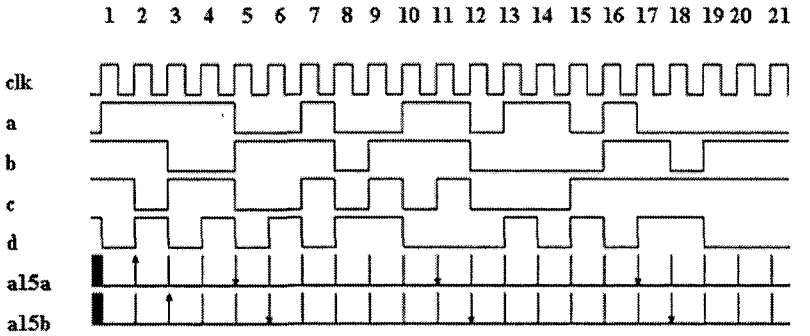

Figure 1-18. Waveform for SVA checker using "ended"

The first failure for assertion a15a happens at clock cycle 5. A valid starting point is detected when signal "a" is detected high on a given positive clock edge and is followed by a high on signal "b" one clock cycle later (clock cycle 6). This leads to checking the consequent and since signal "c" is not high after one clock cycle, the check fails at clock cycle 7.

The first failure for assertion a15b occurs at clock cycle 6. A valid starting point is detected when sequence s15a ends successfully at clock cycle 6. This leads to checking the consequent wherein a valid end point for sequence s15b is expected at clock cycle 8. Since signal "c" does not go high as expected at clock cycle 7, the end point value of the sequence is false and hence the check fails at clock cycle 8.

There are 2 different ways of writing the same check. The first method synchronizes the sequences based on the starting points of the sequences. The second method synchronizes the sequences based on the end points of the sequences.

Table 1-10. Evaluation table for SVA checker using "ended"

Clock tick	Sampled value of "a"	Sampled value of "b"	Sampled value of "c"	Sampled value of "d"	A15a status	A15b status
1	0	1	1	1	Vacuous success	Vacuous success
2	1	1	1	0	Real Success (start at 2, end at 5)	Vacuous success
3	1	1	0	1	Vacuous success	Real Success (start at 3, end at 5)
4	1	0	1	0	Vacuous success	Vacuous success
5	1	0	1	1	Fail (start at 5, end at 7)	Vacuous success
6	0	1	0	0	Vacuous success	Fail (start at 6, end at 8)
7	0	1	0	1	Vacuous success	Vacuous success
8	1	1	1	0	Vacuous success	Vacuous success
9	0	0	0	1	Vacuous success	Vacuous success
10	0	1	1	1	Vacuous success	Vacuous success
11	1	1	0	0	Fail (start at 11, end at 13)	Vacuous success
12	1	1	1	0	Vacuous success	Fail (start at 12, end at 14)
13	0	0	0	0	Vacuous success	Vacuous success
14	1	0	0	1	Vacuous success	Vacuous success
15	1	0	0	0	Vacuous success	Vacuous success
16	0	0	1	1	Vacuous success	Vacuous success
17	1	1	1	0	Fail (start at 17, end at 20)	Vacuous success

1.17 SVA Checker using parameters

SVA allows using parameters in the checkers just like Verilog. This gives great flexibility in creating re-usable properties. For example, the delay information between 2 signals can be parameterized within the checker and then the checker can be re-used in a similar situation elsewhere in the design with different timing relationships. Example 1.2 shows a checker defined with a default value for the parameter delay. If this checker is called within the design, it uses a delay of one clock cycle by default. The same checker can be re-used by over-writing the delay parameter value while instantiating the checker. In Example 1.2, module "top" has 2 instances of the "generic_chk" checker. Instance i1 overwrites the delay parameter as 2 clock cycles and instance i2 uses the default value of 1 clock cycle.

Example 1.2 Sample SVA checker using parameters

```
module generic_chk (input logic a, b, clk);

parameter delay = 1;

property p16;
  @(posedge clk) a |-> ##delay b;
endproperty

a16: assert property(p16);

endmodule

// call checker from the top level module

module top(.....);
logic clk, a, b, c, d;
.
.
generic_chk #(.delay(2)) i1 (a, b, clk);
generic_chk i2 (c, d, clk);
.
.
endmodule
```

Figure 1-19 shows how the 2 instances of checkers, i1 and i2, react to transitions in signals during a simulation.

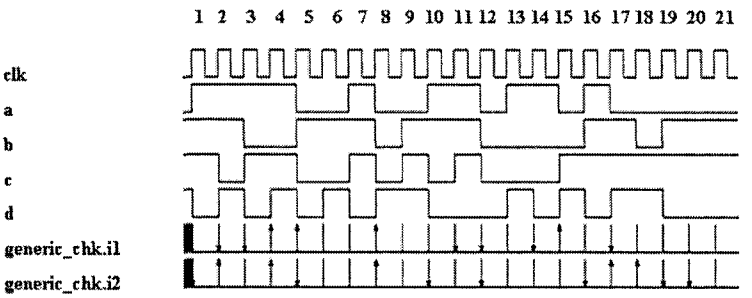

Figure 1-19. Waveform for SVA checker with parameters

1.18 SVA Checker using a select operator

SVA allows using logical operators within sequences and properties. Property p17 checks that, if signal "c" is high then the value of signal "d" is equal to the value of signal "a." If signal "c" is not detected high, then the value of signal "d" is equal to the value of signal "b." This is a combinational check and is performed on every positive edge of clock.

```
property p17;
  @(posedge clk) c ? d == a : d == b;
endproperty

a17: assert property(p17);
```

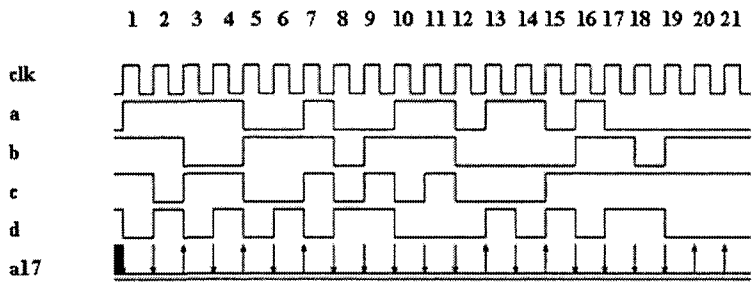

Figure 1-20. Waveform for SVA checker using select operator

Figure 1-20 shows how property p17 reacts in a simulation. Table 1-11 summarizes the sampled values of the respective signals and the status of the assertion a17. At clock cycle 1, signal "c" is detected high and hence, the

check expects that signal "d" and signal "a" have the same value. But signal "d" is detected as high and signal "a" as low and hence the check fails.

Table 1-11. Evaluation table for SVA checker using select operator

Clock tick	Sampled value of "a"	Sampled value of "b"	Sampled value of "c"	Sampled value of "d"	a17 status
1	0	1	1	1	Fail
2	1	1	1	0	Fail
3	1	1	0	1	Success
4	1	0	1	0	Fail
5	1	0	1	1	Success
6	0	1	0	0	Fail
7	0	1	0	1	Success
8	1	1	1	0	Fail
9	0	0	0	1	Fail
10	0	1	1	1	Fail
11	1	1	0	0	Fail
12	1	1	1	0	Fail
13	0	0	0	0	Success
14	1	0	0	1	Fail
15	1	0	0	0	Success
16	0	0	1	1	Fail
17	1	1	1	0	Fail

1.19 SVA Checker using true expression

SVA checkers can be extended in time by using a **`true** expression. This represents a "don't care" condition and it extends the sequence by a clock cycle. This can be used when writing complex protocols wherein multiple properties are monitored and matched simultaneously.

Sequence s18a checks for a simple condition. Sequence s18a_ext checks for the same condition, but moves the match on this sequence by one clock cycle. This has an impact on when this sequence is used in the antecedent of a property. The end points of the 2 sequences are different and hence the clock cycle at which the consequent will be checked will vary.

Property p18 checks for a match on s18a.ended in the antecedent and 2 clock cycles later, checks for a match on s18b.ended. Property p18_ext checks for a match on s18a_ext.ended in the antecedent. The match on this is the same as the match on s18a.ended, but moved 1 clock cycle ahead. Hence, the consequent of property p18_ext needs to match after one clock cycle and not 2 clock cycles as defined in property p18. Both properties p18 and p18_ext check for the same condition, but they both have different matching points for their antecedents.

```
`define true 1

sequence s18a;
 @(posedge clk) a ##1 b;
endsequence

sequence s18a_ext;
 @(posedge clk) a ##1 b ##1 `true;
endsequence

sequence s18b;
 @(posedge clk) c ##1 d;
endsequence

property p18
 @(posdge clk) s18a.ended |-> ##2 s18b.ended;
endproperty

property p18_ext
 @(posdge clk) s18a_ext.ended |=> s18b.ended;
endproperty

a18: assert property(p18);
a18_ext: assert property(p18_ext);
```

Figure 1-21 shows how property p18 and p18_ext react in a simulation. It is clearly seen that the starting point of assertion a18_ext is delayed by one cycle when compared to that of the assertion a18.

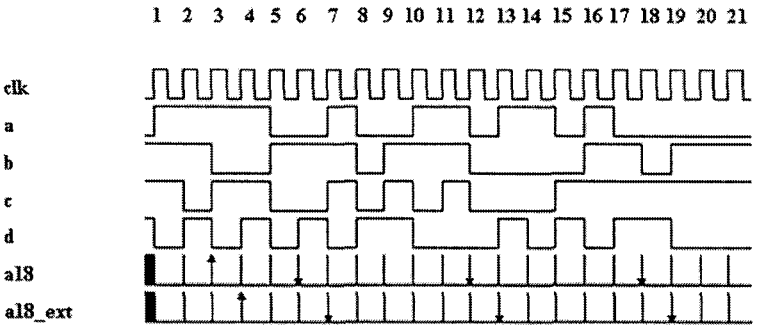

Figure 1-21. Waveform for SVA checker using `true expression

1.20 The "$past" construct

SVA provides a built in system task called **$past** that is capable of getting values of signals from previous clock cycles. By default, it provides the value of the signal from the previous clock cycle. The simple syntax of this construct is as follows.

$past (signal_name, number of clock cycles)

This task can be used effectively to verify that, the path taken by the design to get to the state in this current clock cycle is valid. Property p19 checks that in the given positive clock edge, if the expression (c **&&** d) is true, then 2 cycles before that, the expression (a **&&** b) was true.

```
Property p19;
  @(posedge clk) (c && d) |->
        ($past((a&&b), 2) == 1'b1);
endproperty

a19: assert property(p19);
```

Figure 1-22 shows how the property p19 reacts in a simulation. Table 1-12 summarizes the sampled values of the relevant signals and the status of the assertion a19. The assertion fails at clock cycle 1. At clock cycle 1, there is a valid start since both signal "c" and signal "d" are high. The consequent of the checker needs to compare the value of the expression (a **&&** b) 2 cycles before. This is not possible since there is no history for these signals

before clock cycle 1 and hence the values are assumed to be "x." Hence, the checker fails at clock cycle 1.

The check has a real success at clock cycle 5. At clock cycle 5, there is a valid start since both signal "c" and signal "d" are high. The consequent checks that at clock cycle 3, the expression (a **&&** b) is true. As expected, at clock cycle 3, the signals "a" and "b" are detected high and hence the check succeeds.

The check fails at clock cycle 16. At clock cycle 16, there is a valid start, since both signal "c" and signal "d" are high. The consequent checks that at clock cycle 14, the expression (a **&&** b) is true. The signal "a" is detected high as expected and signal "b" is detected low. This makes the expression (a **&&** b) false and hence the check fails.

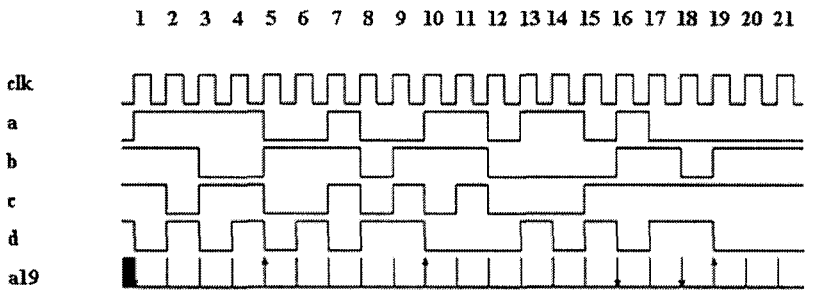

Figure 1-22. Waveform for SVA checker using **"$past"** construct

Table 1-12. Evaluation table for SVA checker using **$past** construct

Clock tick	Sampled value of "a"	Sampled value of "b"	Sampled value of "c"	Sampled value of "d"	a19 status
1	0	1	1	1	Fail
2	1	1	1	0	Vacuous success
3	1	1	0	1	Vacuous success
4	1	0	1	0	Vacuous success
5	1	0	1	1	Real Success
6	0	1	0	0	Vacuous success
7	0	1	0	1	Vacuous success

Clock tick	Sampled value of "a"	Sampled value of "b"	Sampled value of "c"	Sampled value of "d"	a19 status
8	1	1	1	0	Vacuous success
9	0	0	0	1	Vacuous success
10	0	1	1	1	Real Success
11	1	1	0	0	Vacuous success
12	1	1	1	0	Vacuous success
13	0	0	0	0	Vacuous success
14	1	0	0	1	Vacuous success
15	1	0	0	0	Vacuous success
16	0	0	1	1	Fail
17	1	1	1	0	Vacuous success

1.20.1 The $past construct with clock gating

The **$past** construct can be used with a gating signal. For example, on a given clock edge, the gating signal has to be true even before checking for the consequent condition. The simple syntax of a **$past** construct with a gating signal is as follows.

$past (signal_name, number of clock cycles, gating signal)

Property p20 is similar to the property p19. But the check is effective only if the gating signal "e' is valid on any given positive edge of the clock.

```
Property p20;
  @(posedge clk) (c && d) |->
              ($past((a&&b), 2, e) == 1'b1);
endproperty

a20: assert property(p20);
```

1.21 Repetition operators

If signal "start" is high on a given positive edge of the clock, then, starting from the next clock cycle, signal "a" stays high for 3 continuous clock cycles; one clock cycle after that, signal "stop" is high.

A sequence like this can be checked by the following SVA code.

```
@(posedge clk) $rose(start) |->
            ##1 a ##1 a ##1 a ##1 stop
```

Writing such a checker can get very verbose if signal "a" has to stay high for many cycles. Also, in this case, it is assumed that signal "a" stays high continuously. This protocol can get complex when we want to check if signal "a" stays high, not necessarily on three continuous clock cycles. In other words, signal "a" should repeat itself 3 times continuously or intermittently.

SVA language provides three different types of repetition operators: Consecutive repetition, go to repetition and non-consecutive repetition.

Consecutive repetition – This allows the user to specify that a signal or a sequence will match continuously for the number of clocks specified. A hidden delay of one clock cycle is assumed between each match of the signal. The simple syntax of consecutive repetition operator is shown below.

```
signal or sequence [*n]
```

"n" is the number of times the expression should match repeatedly.

For example a [*3] will expand to the following.

```
a ##1 a ##1 a
```

A sequence such as (a ##1 b) [*3] will expand as follows.

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

Go to repetition – This allows the user to specify that an expression will match the number of times specified not necessarily on continuous clock cycles. The matches can be intermittent. The main requirement of a "go to" repeat is that the last match on the expression checked for repetition should happen in the clock cycle before the end of the entire sequence matching. The simple syntax of "go to" repetition operator is shown below.

```
Signal [->n]
```

Consider the following sequence.

```
Start ##1 a[->3] ##1 stop
```

It is required that there is a match on signal "a" (the third and final repetition of signal "a") just before the success of "stop." In other words, signal "stop" succeeds on the last clock cycle of the sequence match, and in the previous clock cycle, there should be a match on signal "a."

Non-consecutive repetition – This is very similar to "go to" repetition except that it does not require that the last match on the signal repetition happen in the clock cycle before the end the entire sequence matching. The simple syntax of a non-consecutive repetition operator is shown below.

```
Signal [=n]
```

Only expressions are allowed to repeat in "go to" and "non-consecutive" repetitions. Sequences are not allowed.

1.21.1 Consecutive repetition operator [*]

Property p21 checks that two clock cycles after a valid start, signal "a" stays high for 3 continuous clock cycles and two clock cycles after that, signal "stop" is high. One clock cycle later signal "stop" is low.

```
Property p21;
 @(posedge clk) $rose(start) |->
                  ##2 (a[*3]) ##2 stop ##1 !stop;
endproperty

a21: assert property(p21);
```

Figure 1-23 shows how property p21 reacts in a simulation. The waveform shows 2 failures and 1 real success. All other successes are vacuous.

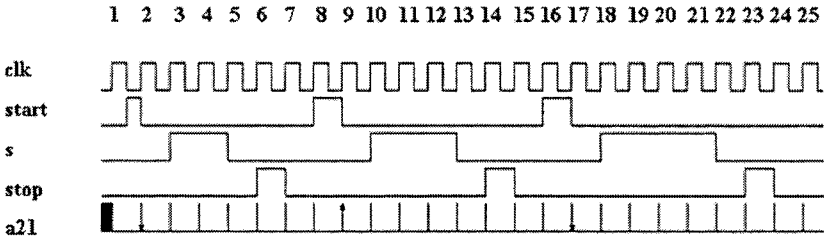

Figure 1-23. Waveform for SVA checker using consecutive repeat

Failure at clock cycle 2 – A valid start signal is detected at clock cycle 2. The checker then looks for signal "a" to be high on 3 continuous clock cycles starting from the positive clock edge of clock cycle 4. Signal "a" is detected high on clock cycle 4 and 5, but is detected low on clock cycle 6. Hence, the check fails. Note that the check started at clock cycle 2 and failed at clock cycle 6.

Success at clock cycle 9 - A valid start signal is detected at clock cycle 9. The checker then looks for signal "a" to be high for 3 continuous clock cycles starting from the positive clock edge of clock cycle 11. Signal "a" is detected high on clock cycle 11, 12 and 13 as expected. Two clock cycles later (at clock cycle 15) signal "stop" is high as expected. One clock cycle later the signal "stop" is detected low. Hence, the check succeeds. Note that the check started at clock cycle 9 and finished at clock cycle 16.

Failure at clock cycle 17 - A valid start signal is detected at clock cycle 17. The checker then looks for signal "a" to be high for 3 continuous clock cycles starting from the positive clock edge of clock cycle 19. Signal "a" is detected high on clock cycles 19, 20 and 21. The check now looks for a high on signal "stop" at clock cycle 23 but it is not there. Hence, the check fails. Note that signal "a" remained high for 4 clock cycles. The checker needs only 3 repeats and hence it moves on to look for the signal "stop." The check started at clock cycle 19 and failed at clock cycle 23.

1.21.2 Consecutive repetition operator [*] on a sequence

Property p22 checks that two clock cycles after a valid start, sequence (a ##2 b) repeats 3 times and two clock cycles after that, signal "stop" is high.

```
Property p22;
 @(posedge clk) $rose(start) |->
        ##2 ((a ##2 b)[*3]) ##2 stop;
endproperty

a22: assert property(p22);
```

Figure 1-24 shows how property p22 reacts in a simulation. It shows 2 failures and one real success.

Failure 1 – The first failure is shown by marker 1s. A valid start is detected at this point. After 2 clock cycles from this point, the checker expects that the sequence (a ##2 b) repeats three times. But in this case, the sequence is repeated only 2 times. Hence, the checker fails and the failing point is shown by marker 1e.

Success 1 – The only real success is shown by marker 2s. A valid start is detected at this point. After 2 clock cycles from this point, the checker expects that the sequence (a ##2 b) repeats three times. The sequence is repeated 3 times as expected. A valid stop is expected 2 clock cycles after the successful repetition of the sequence and it happens as expected. Hence, the checker succeeds and the succeeding point is shown by marker 2e.

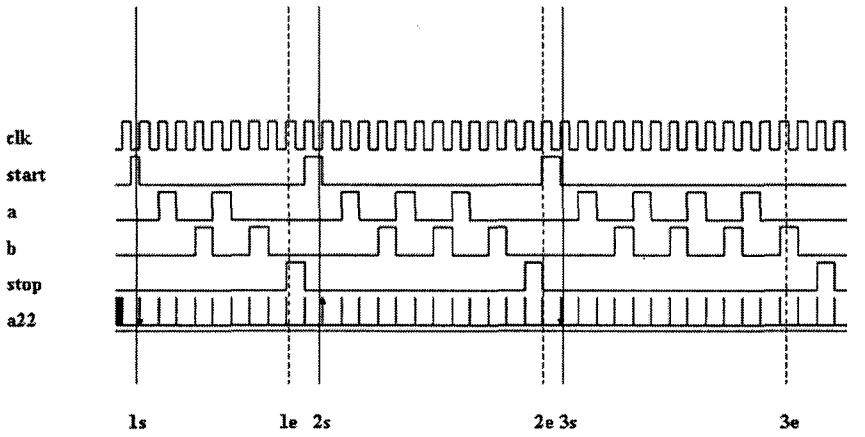

Figure 1-24. Waveform for SVA checker using consecutive repeat on a sequence

Failure 2 – The second failure is shown by marker 3s. A valid start is detected at this point. After 2 clock cycles from this point, the checker expects that the sequence (a ##2 b) repeats three times. The sequence repeats as expected. A valid stop is expected 2 clock cycles after the successful repetition of the sequence and it does not arrive. Hence, the checker fails and the failing point is shown by marker 3e.

1.21.3 Consecutive repetition operator [*] on a sequence with a delay window

Property p23 checks that two clock cycles after a valid start, sequence (a ##[1:4] b) repeats 3 times and two clock cycles after that, signal "stop" is high. The fact that the sequence has a timing window makes this check slightly complicated.

```
property p23;
  @(posedge clk) $rose(start) |->
      ##2 ((a ##[1:4] b)[*3]) ##2 stop;
endproperty

a23: assert property(p23);
```

The main sequence (a **##[**1:4] b) **[***3] expands as follows.

((a ##1 b) or (a ##2 b) or (a ##3 b) or (a ##4 b)) ##1
((a ##1 b) or (a ##2 b) or (a ##3 b) or (a ##4 b)) ##1
((a ##1 b) or (a ##2 b) or (a ##3 b) or (a ##4 b))

Figure 1-25 shows how property p23 reacts in a simulation. It shows 2 failures and one real success.

Failure 1 – The first failure is shown by marker 1s. A valid start is detected at this point. After 2 clock cycles from this point, the checker expects that the sequence (a ##[1:4] b) repeats three times. But in this case, the sequence is repeated only 2 times. Hence, the checker fails and the failing point is shown by marker 1e. Note that the 2 repeats that matched are (a ##1 b) and (a ##2 b) respectively.

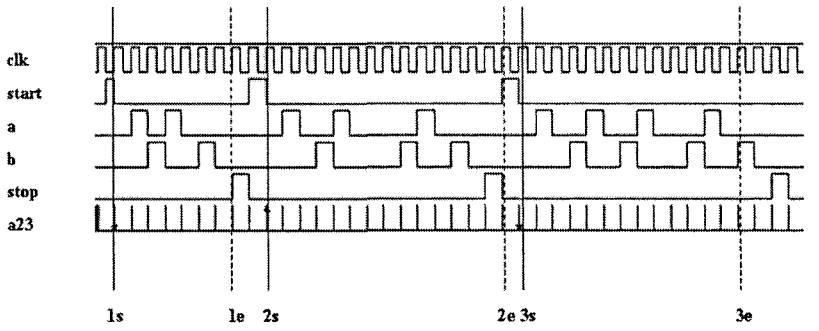

Figure 1-25. Waveform for SVA checker using consecutive repeat on a sequence with window of delay

Success 1 – The only real success is shown by marker 2s. A valid start is detected at this point. After 2 clock cycles from this point, the checker expects that the sequence (a ##[1:4] b) repeats three times. The sequence is repeated 3 times as expected. A valid stop is expected 2 clock cycles after the successful repetition of the sequence and it happens as expected. Hence, the checker succeeds and the succeeding point is shown by marker 2e. Note that the 3 repeats that matched are (a ##2 b), (a ##4 b) and (a ##2 b) respectively.

Failure 2 – The second failure is shown by marker 3s. A valid start is detected at this point. After 2 clock cycles from this point, the checker expects that the sequence (a ##[1:4] b) repeats three times. The sequence does repeat as expected. A valid stop is expected 2 clock cycles after the successful repetition of the sequence and it does not arrive as expected. Hence, the checker fails and the failing point is shown by marker 3e. Note that the 3 repeats that matched are (a ##2 b), (a ##2 b) and (a ##3 b) respectively.

1.21.4 Consecutive repetition operator [*] and eventuality operator

Property p23 specified a window of timing for the sequence that repeated itself. It is also possible to provide a window for the number of repetitions. For example, a [*1:5] means that signal "a" should repeat itself anywhere between 1 to 5 times. The definition can be expanded as follows.

```
a or
(a ##1 a) or
(a ##1 a ##1 a) or
```

```
(a ##1 a ##1 a ##1 a) or
(a ##1 a ##1 a ##1 a ##1 a)
```

The bounds of the repeat window follow the same rules as the delay windows. The left hand side value should be lesser than the right hand side value. The right hand side value can be a "**$**" sign indicating an unbounded number of repeats.

Property p24 shows an example of a finite check with an unbounded number of repeats defined. It checks that 2 cycles after a valid start signal, the signal "a" will stay high repeatedly until a valid stop arrives.

```
Property p24;
 @(posedge clk) $rose(start) |->
          ##2 (a[*1:$]) ##1 stop;
endproperty

a24: assert property(p24);
```

Figure 1-26 shows how property p24 reacts in a simulation. It shows one failure and one real success.

Failure 1 – A valid start occurs at clock cycle 3 shown by marker 1s. The check expects that 2 clock cycles from this point, signal "a" will stay high repeatedly until a valid stop arrives. Signal "a" detects high continuously until clock cycle 7. In clock cycle 8, it is detected low but the signal "stop" has not arrived yet. Hence, the check fails at clock cycle 8 shown by marker 1e.

Success 1 – A valid start occurs at clock cycle 11 shown by marker 2s. The check expects that 2 clock cycles from this point, signal "a" will stay high repeatedly until a valid stop arrives. Signal "a" stays high continuously until clock cycle 15. In clock cycle 16, it is detected low and the signal "stop" arrives as expected. Hence, the check succeeds at clock cycle 16 shown by marker 2e.

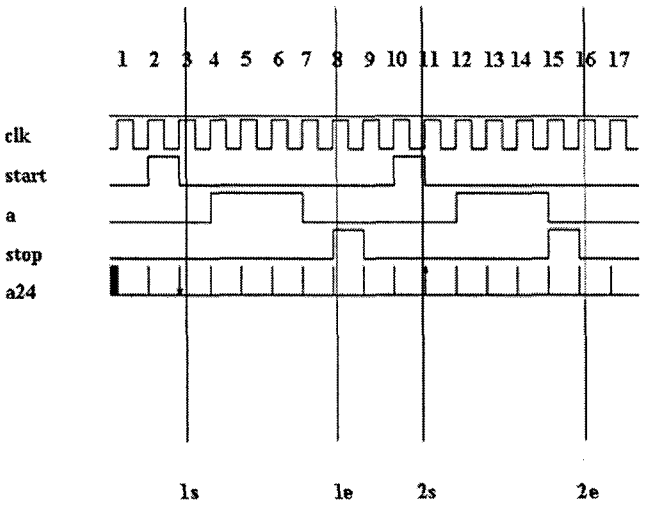

Figure 1-26. Waveform for SVA checker using consecutive repeat and eventuality

1.21.5 Go to repetition operator [->]

Property p25 checks that, if there is a valid start signal on any given positive edge of the clock, 2 clock cycles later, signal "a" will repeat three times continuously or intermittently before there is a valid stop signal.

```
property p25;
  @(posedge clk) $rose(start) |->
           ##2 (a[->3]) ##1 stop;
endproperty

a25: assert property(p25);
```

Figure 1-27 shows how property p25 reacts in a simulation. The figure shows that there is one failure, one real success and one incomplete check.

Failure 1 - A valid start of the checker is shown by marker 1s. The check expects that 2 clock cycles after the valid start, signal "a" will repeat three times. Signal "a" repeats 3 times as expected. After the third match on signal "a," a valid "stop" signal is expected on the next clock cycle. This does not happen and hence the check fails as shown by marker 1e.

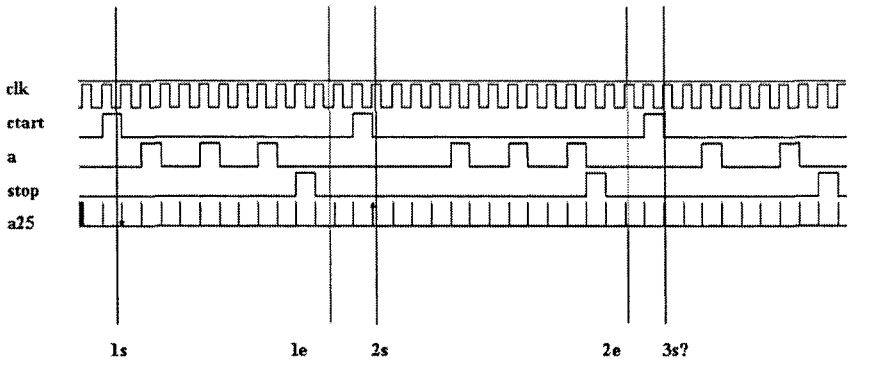

Figure 1-27. Waveform for SVA checker using go to repetition operator

Success 1 - A valid start of the checker is shown by marker 2s. The check expects that 2 clock cycles after the valid start, signal "a" will repeat three times. Signal "a" repeats 3 times as expected. After the third match on signal "a," a valid "stop" signal is expected on the next clock cycle. The "stop" signal arrives as expected and hence the check succeeds at marker 2e.

Incomplete 1 - A valid start of the checker is shown by marker 3s. The check expects that 2 clock cycles after the valid start, signal "a" will repeat three times. Signal "a" repeats 2 times; before the third one arrives, the simulation is finished. Also note that a valid "stop" signal arrives before the end of the simulation cycles. This "stop" will not have any effect since the repeat statement has not completed. The 3 expected repeats act as a blocking statement before the "stop" signal. Hence, the check is incomplete at the end of the simulation.

1.21.6 Non-consecutive repetition operator [=]

Property p26 checks that if there is a valid start signal on any given positive edge of the clock, 2 clock cycles later, signal "a" will repeat three times continuously or intermittently before there is a valid stop signal. One clock cycle later, the signal "stop" should be detected low. It checks for the exact same thing as property p25 except that it uses a "non-consecutive" repeat operator in the place of a "go to" repeat operator. This means that, in property p26, there is no expectation that there is a valid match on signal "a" in the previous cycle of a valid match on "stop" signal.

```
Property p26;
 @(posedge clk) $rose(start) |->
```

```
                    ##2 (a[=3]) ##1 stop ##1 !stop;
endproperty

a26: assert property(p26);
```

Figure 1-28 shows how property p26 reacts in a simulation. The figure shows that there are two successes and one incomplete check.

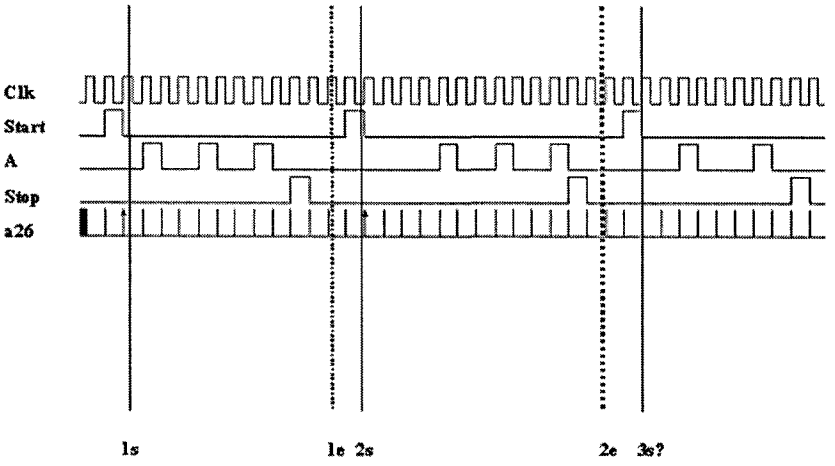

Figure 1-28. Waveform for SVA checker using non-consecutive repetition operator

Success 1 - A valid start of the checker is shown by marker 1s. The check expects that 2 clock cycles after the valid start, signal "a" will repeat three times. Signal "a" repeats 3 times as expected. After the third match on signal "a," a valid "stop" signal is expected, not necessarily on the next clock cycle. A valid "stop" signal arrives 2 clock cycles after the third match on signal "a" and hence the check succeeds as shown by marker 1e. This is the main difference between "go to" repetition and "non-consecutive" repetition. Property p25 failed for the same condition since a "go to" repetition was used.

Success 2 - A valid start of the checker is shown by marker 2s. The check expects that 2 clock cycles after the valid start, signal "a" will repeat three times. Signal "a" repeats 3 times as expected. After the third match on signal "a," a valid "stop" signal is expected, not necessarily on the next clock cycle. A valid "stop" signal arrives 1 clock cycle after the third match on signal "a" and hence the check succeeds as shown by marker 2e.

Incomplete 1 - A valid start of the checker is shown by marker 3s. The check expects that 2 clock cycles after the valid start, signal "a" will repeat three times. Signal "a" repeats 2 times; before the third one arrives, the simulation is finished. Also note that a valid "stop" signal arrives before the end of the simulation cycles. This "stop" will not have any effect since the repeat statement has not completed. The 3 expected repeats act as a blocking statement before the "stop" signal. Hence, the check is incomplete at the end of the simulation. This behavior is the same as in "go to" repetition.

1.22 The "and" construct

The binary operator "**and**" can be used to combine two sequences logically. The final property succeeds when both the sequences succeed. ***Both sequences must have the same starting point but they can have different ending points.*** The starting point of the check is when the first sequence succeeds and the end point is when the other sequence succeeds, ultimately making the property succeed.

Sequence s27a and s27b are two independent sequences. The property p27 combines them with an **and** operator. The property succeeds when both the sequences succeed.

```
sequence s27a;
 @(posedge clk) a##[1:2] b;
endsequence

sequence s27b;
 @(posedge clk) c##[2:3] d;
endsequence

property p27;
 @(posedge clk)  s27a and s27b;
endproperty

a27: assert property(p27);
```

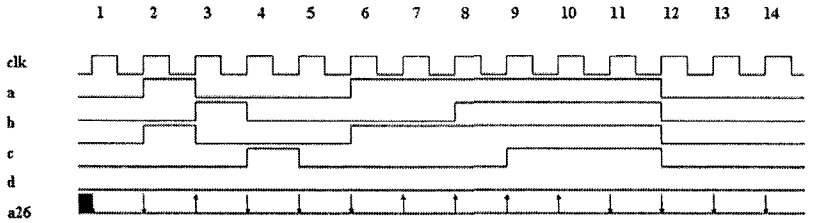

Figure 1-29. Waveform for SVA checker using "and" construct

Figure 1-29 shows how property p27 reacts in a simulation. Table 1-13 summarizes the sampled values of all relevant signals and the status of the assertion a27. There are 3 types of results. There can be a failure due to lack of a valid start. This happens on a given clock edge if signal "a" is not high or signal "c" is not high (clock cycles 1, 2, 4, 5, 6, 13, 14).

Table 1-13. Evaluation table for SVA checker using "and" construct

Clock cycle	Sampled value of "a"	Sampled value of "b"	Sampled value of "c"	Sampled value of "d"	Valid start	a27 status
1	0	0	0	0	No	Fail
2	0	0	0	0	No	Fail
3	1	0	1	0	Yes	Success (start at 3, end at 5)
4	0	1	0	0	No	Fail
5	0	0	0	1	No	Fail
6	0	0	0	0	No	Fail
7	1	0	1	0	Yes	Success (start at 7, end at 10)
8	1	0	1	0	Yes	Success (start at 8, end at 10)
9	1	1	1	0	Yes	Success (start at 9, end at 11)
10	1	1	1	1	Yes	Success (start at 10, end at 12)
11	1	1	1	1	Yes	Fail (start at 11, end at 14)
12	1	1	1	1	Yes	Fail (start at 12, end at 14)

Clock cycle	Sampled value of "a"	Sampled value of "b"	Sampled value of "c"	Sampled value of "d"	Valid start	a27 status
13	0	0	0	0	No	Fail
14	0	0	0	0	No	Fail

There are 5 different successes and each one of them has a different length. The valid checks that started at clock cycle 7 and clock cycle 8 both finish at clock cycle 10. For the check that starts at clock cycle 7, signal "b" is true in clock cycle 9, and signal "d" is true in clock cycle 10. For the check that starts at clock cycle 8, signal "b" is true in clock cycle 9, and signal "d" is true in clock cycle 10.

There are two failures, one at clock cycle 11 and one at clock cycle 12. Each one of them has the same length but they fail due to different reasons. For the check that starts at clock cycle 11, signal "b" is true in clock cycle 12. But signal "d" is never true in clock cycles 13 or 14 and hence the check fails at clock cycle 14. For the check that starts at clock cycle 12, signal "b" is not true in clock cycle 13. Both signal "b" and signal "d" are not true in clock cycle 14 and hence the check fails in clock cycle 14.

1.23 The "intersect" construct

The "**intersect**" operator is very similar to the "**and**" operator with one additional requirement. ***Both the sequences need to start at the same time and complete at the same time. In other words, the length of both sequences should be the same.***

Property p28 checks for the same condition as property p27. The only difference is that it uses the **intersect** construct instead of the **and** construct.

```
sequence s28a;
 @(posedge clk) a##[1:2] b;
endsequence

sequence s28b;
 @(posedge clk) c##[2:3] d;
endsequence

property p28;
 @(posedge clk)  s28a intersect s28b;
endproperty
```

```
a28: assert property(p28);
```

Figure 1-30 shows how property p28 reacts in a simulation. Table 1-14 summarizes the sampled values of all the relevant signals and the status of the assertion a28. Figure 1-30 also shows the results of assertion a27 that uses the **and** construct on the same set of design conditions. This helps understand the differences between the **and** construct and the **intersect** construct.

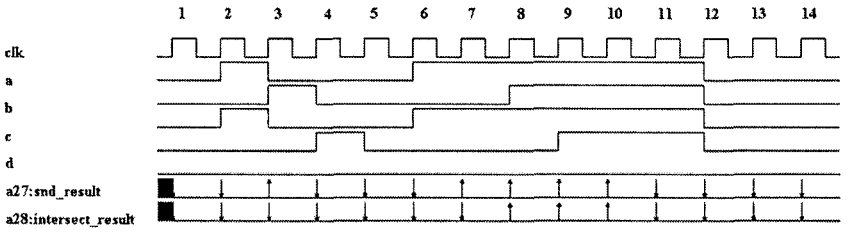

Figure 1-30. Waveform for SVA checker using "intersect" construct

The failures due to lack of a valid start remain the same. The second set of failure happens from the fact that the individual sequences do not match as expected. This kind of failure happens at clock cycles 11 and 12. The third set of failure happens even though the individual sequences match as expected. ***These failures happen since the individual sequence did not take the same length of time to match.*** In the failure shown in clock cycle 3, sequence s28a takes one clock cycle to match ("a" is true in clock cycle 3 and "b" is true in clock cycle 4) and sequence s28b takes 2 clock cycles to match ("c" is true in clock cycle 3 and "d" is true in clock cycle 5). In the failure shown in clock cycle 7, s28a takes two clock cycles to match ("a" is true in clock cycle 7 and "b" is true in clock cycle 9) and sequence s28b takes three clock cycles to match ("c" is true in clock cycle 7 and "d" is true in clock cycle 10).

The three successes happen at clock cycles 8, 9 and 10 respectively. In all these three cases the sequences match with the same length of time.

In the success shown in clock cycle 8, sequence s28a matches twice, at clock cycles 9 and 10. The sequence s28b also matches twice, at clock cycles 10 and 11. The common length is 2 clock cycles for both the sequences to match. Hence, the **intersect** succeeds with s28a matching at

clock cycle 10 and s28b also matching at clock cycle 10. Each of them has a length of 2 clock cycles.

Table 1-14. Evaluation table for SVA checker using "intersect" construct

Clock tick	Sampled value of "a"	Sampled value of "b"	Sampled value of "c"	Sampled value of "d"	Valid start	a28 status
1	0	0	0	0	No	Fail
2	0	0	0	0	No	Fail
3	1	0	1	0	Yes	**Fail (sequences succeed with different length)**
4	0	1	0	0	No	Fail
5	0	0	0	1	No	Fail
6	0	0	0	0	No	Fail
7	1	0	1	0	Yes	**Fail (sequences succeed with different length)**
8	1	0	1	0	Yes	Success (start at 8, end at 10)
9	1	1	1	0	Yes	Success (start at 9, end at 11)
10	1	1	1	1	Yes	Success (start at 10, end at 12)
11	1	1	1	1	Yes	Fail (start at 11, end at 13)
12	1	1	1	1	Yes	Fail (start at 12, end at 14)
13	0	0	0	0	No	Fail
14	0	0	0	0	No	Fail

In the success shown in clock cycle 9, sequence s28a matches twice, at clock cycles 10 and 11. The sequence s28b also matches twice, at clock cycles 11 and 12. The common length is 2 clock cycles for both the sequences to match. Hence, the **intersect** succeeds with s28a matching at clock cycle 11 and s28b also matching at clock cycle 11. Each of them has a length of 2 clock cycles.

In the success shown in clock cycle 10, sequence s28a matches twice, at clock cycles 11 and 12. The sequence s28b matches at clock cycle 12. The common length is 2 clock cycles for both the sequences to match. Hence, the

intersect succeeds with s28a matching at clock cycle 12 and s28b also matching at clock cycle 12. Each of them has a length of 2 clock cycles.

1.24 The "or" construct

The binary operator "**or**" can be used to combine two sequences logically. ***The final property succeeds when any one of the sequence succeeds.***

Sequence s29a and s29b are two independent sequences. The property p29 combines them with an **or** operator. The property succeeds when any one of the sequence succeeds.

```
sequence s29a;
 @(posedge clk) a##[1:2] b;
endsequence

sequence s29b;
 @(posedge clk) c##[2:3] d;
endsequence

property p29;
 @(posedge clk)  s28a or s28b;
endproperty

a29: assert property(p29);
```

Figure 1-31 shows how property p29 reacts in a simulation. Table 1-15 summarizes the sampled values of all the relevant signals and the status of the assertion a29. Figure 1-31 also shows the results of assertion a27 that uses the **and** construct on the same set of design conditions. This helps understand the differences between the **and** construct and the **or** construct. The failures due to the lack of a valid start remain the same. The second set of failure happens from the fact that the individual sequences do not match as expected. This kind of failure happens at clock cycle 12. Both sequences never match within their timing window and hence the check fails.

The successes are almost the same for the **and** operator and **or** operator. The main difference is the duration of the match. The **or** operator matches as soon as a match is found on sequence s29a and hence does not wait for sequence s29b to finish.

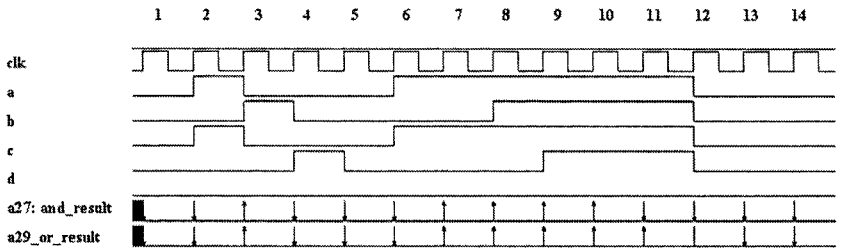

Figure 1-31. Waveform for SVA checker using "or" construct

Table 1-15. Evaluation table for SVA checker using "or" construct

Clock tick	Sampled value of "a"	Sampled value of "b"	Sampled value of "c"	Sampled value of "d"	Valid start	a29 status
1	0	0	0	0	No	Fail
2	0	0	0	0	No	Fail
3	1	0	1	0	Yes	Success (start at 3, end at 4)
4	0	1	0	0	No	Fail
5	0	0	0	1	No	Fail
6	0	0	0	0	No	Fail
7	1	0	1	0	Yes	Success (start at 7, end at 9)
8	1	0	1	0	Yes	Success (start at 8, end at 9)
9	1	1	1	0	Yes	Success (start at 9, end at 10)
10	1	1	1	1	Yes	Success (start at 10, end at 11)
11	1	1	1	1	Yes	Success (start at 11, end at 12)
12	1	1	1	1	Yes	Fail (start at 12, end at 14)
13	0	0	0	0	No	Fail
14	0	0	0	0	No	Fail

One of the failures with the **and** construct at clock cycle 11 becomes a success with the **or** construct. The reason for this is that the first part of sequence s29a matches at clock cycle 12 and this immediately makes the property succeed. In the **and** construct this alone is not enough. The second part of the sequence has to match, but it does not occur within the specified time window. Therefore, the same condition makes the property p27 fail at clock cycle 14.

1.25 The "first_match" construct

Whenever a timing window is specified in sequences along with binary operators such as **and** and **or,** there is a possibility of getting multiple matches for the same check. The construct "**first_match**" ensures that only the first sequence match is used and the others are discarded. This becomes very helpful when combining multiple sequences together wherein only the first match in the timing window is required to evaluate the remaining part of the property.

In the example shown below, two sequences are combined with an **or** operator. There are several possible matches for this property and they are as follows.

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

When the property p30 gets evaluated, the first one to match will be kept and every other match will be discarded.

```
sequence s30a;
  @(posedge clk) a ##[1:3] b;
endsequence

sequence s30b;
  @(posedge clk) c ##[2:3] d;
endsequence

property p30;
  @(posedge clk) first_match(s30a or s30b);
endproperty
```

```
a30: assert property(p30);
```

Figure 1-32 shows how property p30 reacts in a simulation. There are 2 successes shown in the figure, one at clock cycle 3 and another at clock cycle 9. The success at clock cycle 3 is based on the match on the sequence (c ##2 d). The success at clock cycle 9 is based on the match on the sequence (a ##1 b). In both cases, the first sequence match made the property succeed.

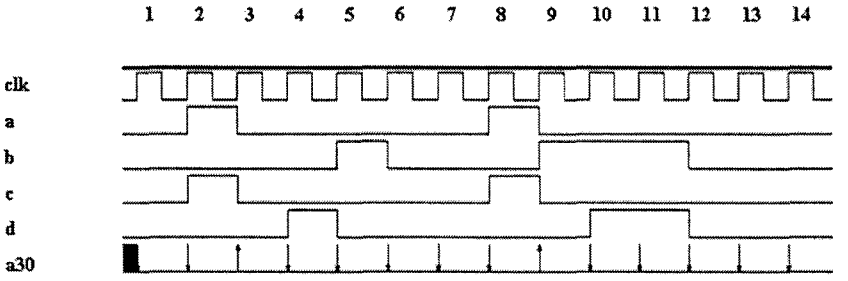

Figure 1-32. Waveform for SVA checker using "first_match" construct

1.26 The "throughout" construct

Implication is one technique discussed so far that allows defining pre-conditions. For example, for a specific sequence to be tested, a certain pre-condition must be true. There are also situations wherein the condition must hold true until the entire test sequence completes. Implication checks for pre-condition once on the clock edge and then starts evaluating the consequent part. Hence, it does not care if the antecedent remains true or not. To make sure that certain condition holds true during the evaluation of the entire sequence, "**throughout**" operator should be used. The simple syntax of a **throughout** operator is shown below.

```
(expression) throughout (sequence definition)
```

Property p31 checks the following.

a. The check starts when signal "start" has a falling edge.
b. Test the expression ((!a&&!b) ##1 (c[->3]) ##1 (a&&b)).
c. The sequence checks that between the falling edge of signals "a"

and "b," and the rising edge of signals "a" and "b," signal "c" should repeat itself 3 times continuously or intermittently.

d. During the entire test expression, signal "start" should always be low.

```
property p31;
 @(posedge clk) $fell(start) |->
       (!start) throughout
       (##1 (!a&&!b) ##1 (c[->3]) ##1 (a&&b));
endproperty

a31: assert property(p31);
```

Figure 1-33 shows how property p31 reacts in a simulation. The check succeeds at clock cycle 3 and fails in clock cycle 16.

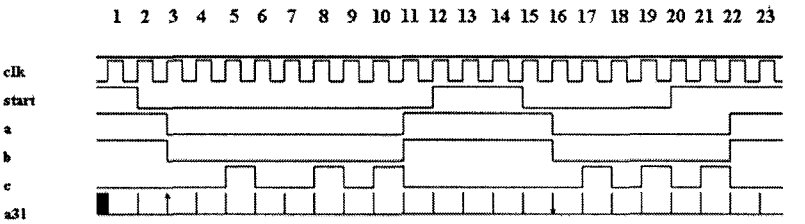

Figure 1-33. Waveform for SVA checker using "throughout" construct

Success 1 – The antecedent of the property succeeds on clock cycle 3 when a falling edge is detected on the start "signal." One cycle after that, signals "a" and "b" are expected to be low, and they are as expected in clock cycle 4. From this point, signal "c" is expected to repeat itself three times. It does repeat three times, once each in clock cycles 6, 9 and 11. In clock cycle 12, it is expected that both signals "a" and "b" are high, and they are as expected. Hence, the property starts at clock 3 and succeeds at clock 12. ***Note that signal "start" was detected low from the clock cycles 3 through 12. That is the key for the success of this check.***

Failure 1 – The antecedent of the property succeeds on clock cycle 16 when a falling edge is detected on the "start" signal. One cycle after that, signals "a" and "b" are expected to be low, and they are in clock cycle 17. From this point signal, "c" is expected to repeat itself three times. We get two repeats on clock cycles 18 and 20. But on clock 21, before the third

repeat on signal "c" arrives, the signal "start" is detected high and the check fails at clock cycle 21. The "throughout" condition was violated here and hence the check fails.

1.27 The "within" construct

The "**within**" construct allows the definition of a sequence contained within another sequence.

```
seq1 within seq2
```

This means that seq1 happens within the start and completion of seq2. The starting matching point of seq2 must happen before the starting matching point of seq1. The ending matching point of seq1 must happen before the ending matching point of seq2. Property p32 checks that the sequence s32a happens within the rise and fall of signal "start." The rise and fall of signal "start" is defined as a sequence in s32b.

```
sequence s32a;
 @(posedge clk)
       ((!a&&!b) ##1 (c[->3]) ##1 (a&&b));
endsequence

sequence s32b;
 @(posedge clk)
       $fell(start) ##[5:10] $rose(start);
endsequence

sequence s32;
 @(posedge clk) s32a within s32b;
endsequence

property p32;
 @(posedge clk) $fell(start) |-> s32;
endproperty

a32: assert property(p32);
```

The same set of design conditions used to describe the **throughout** operator is used in Figure 1-34 to show how property p32 reacts in a simulation. There are two valid starts for this check, one at clock cycle 3 and

another at clock cycle 16. In both these clocks, a falling edge of the signal "start" is detected.

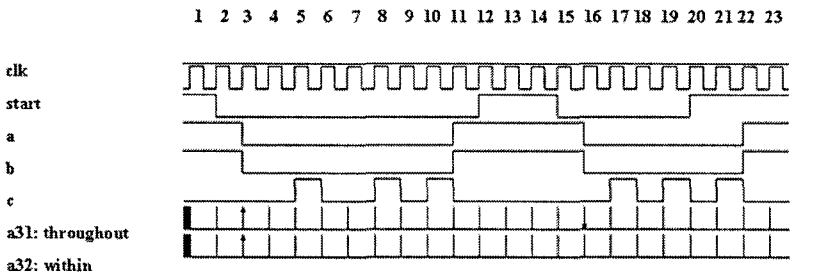

Figure 1-34. Waveform for SVA checker using "within" construct

Success 1 – The check starting at clock cycle 3 succeeds. The falling edge of signal "start" is at clock cycle 3 and the rising edge of the signal "start" is at clock cycle 13. Within these clock cycles, signal "c" is detected high three times in clock cycles 6, 9 and 11. Hence, the check succeeds.

Incomplete 1 - The check starting at clock cycle 16 never finishes. The falling edge of signal "start" is at clock cycle 16 and the rising edge of the signal "start" is at clock cycle 21. Within these clock cycles, signal "c" is detected high two times in clock cycles 18 and 20 respectively. The third repeat of signal "c" comes at clock cycle 22 but signal "start" is detected high at clock cycle 21. This is a failure but since a "go to" repetition operator is used to check for signal "c," it acts as a blocking sequence. This makes the check fail and issues an incomplete message during simulation.

1.28 Built-in system functions

SVA provides several built-in functions to check for some of the most common design conditions.

$onehot(expression) – checks that the expression is one-hot, in other words, only one bit of the expression can be high on any given clock edge.

$onehot0(expression) – checks that the expression is zero one-hot, in other words, only one bit of the expression can be high or none of the bits can be high on any given clock edge.

$isunknown(expression) – checks if any bit of the expression is X or Z.

$countones(expression) – counts the number of bits that are high in a vector.

Assert statement a33a checks that the bit vector "state" is one-hot. Assert statement a33b checks that the bit vector "state" is zero one-hot. Assert statement a33c checks if any bit of the vector "bus" is X or Z. Assert statement a33d checks that the number of ones in the vector "bus" is greater than one.

```
a33a: assert
      property(@(posedge clk) $onehot(state));
a33b: assert
      property(@(posedge clk) $onehot0(state));
a33c: assert
      property(@(posedge clk) $isunknown(bus));
a33d: assert
      property(@(posedge clk)$countones(bus)> 1);
```

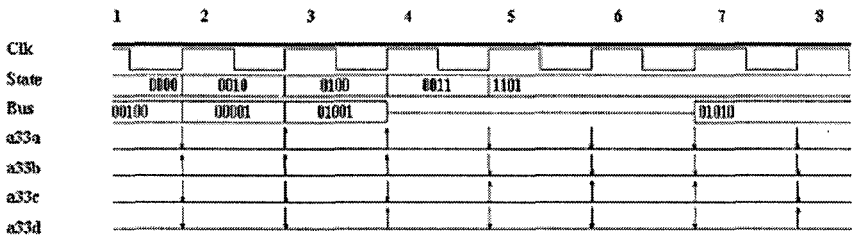

Figure 1-35. Waveform for SVA checker using built-in system functions

Figure 1-35 shows how the assert statements react in a simulation. Table 1-16 summarizes the sampled values of vector "state" and "bus" and the status of each assertion. Note that assertion a33a fails in clock cycle 2 since all bits are zero. The one-hot condition requires that one bit be high on all positive edges of the clock. On the other hand, assertion a33b passes since it checks for zero one-hot and all bits being zero is legal for this construct. Both a33a and a33b fail in clock cycles 5, 6, 7 and 8 wherein more than one bit is high. Assertion a33c fails anytime the value of the vector "bus" is not Z or X. It passes on clock cycles 5, 6 and 7 wherein the value is Z. Assertion

a33d fails on clock cycles 2, 3, 5, 6, and 7 wherein no more than one bit is high. Assertion a33d passes in clock cycles 4 and 8 since 2 bits are high in the vector "bus" on these two clock cycles.

Table 1-16. Evaluation table for SVA checker using built-in functions

Clock tick	Sampled value of "state"	Sampled value of "bus"	a33a - $onehot status	a33b - $onehot0 status	a33c - $isunknown status	a33d - $countones status
2	0000	00100	Fail	Success	Fail	Fail
3	0010	00001	Success	Success	Fail	Fail
4	0100	01001	Success	Success	Fail	Success
5	0011	Z	Fail	Fail	Success	Fail
6	1101	Z	Fail	Fail	Success	Fail
7	1101	Z	Fail	Fail	Success	Fail
8	1101	01010	Fail	Fail	Fail	Success

1.29 The "disable iff" construct

In certain design conditions, we don't want to proceed with the check if some condition is true. In other words, it is like an asynchronous reset that will make the check currently being evaluated void. SVA provides a construct called "**disable iff**" that acts like an asynchronous reset for the checker. The simple syntax for a **disable iff** is as follows.

disable iff (expression) < property definition>

Property p34 checks that after a valid start, signal "a" repeat 2 times and 1 cycle after that, signal "b" repeats 2 times and one cycle later signal "start" becomes low. During this entire sequence, if reset is detected high at any point, the checker will stop and issue a vacuous success by default.

```
property p34;
   @(posedge clk)
  disable iff (reset)
  $rose(start) |=> a[=2] ##1 b[=2] ##1 !start ;
endproperty

a34: assert property(p34);
```

Figure 1-36 shows how property p34 reacts in a simulation. A valid start is shown with marker 1s. After the valid start, signal "a" repeats two times and then signal "b" repeats two times. Signal "start" becomes low after that as expected.

During this entire sequence, the signal "reset" is inactive as expected and hence the check succeeds at marker 1e. A second valid start is shown with marker 2s. After the valid start, signal "a" repeats two times and then the "reset" signal becomes active before signal "b" could repeat two times. This nullifies the check and the property succeeds vacuously.

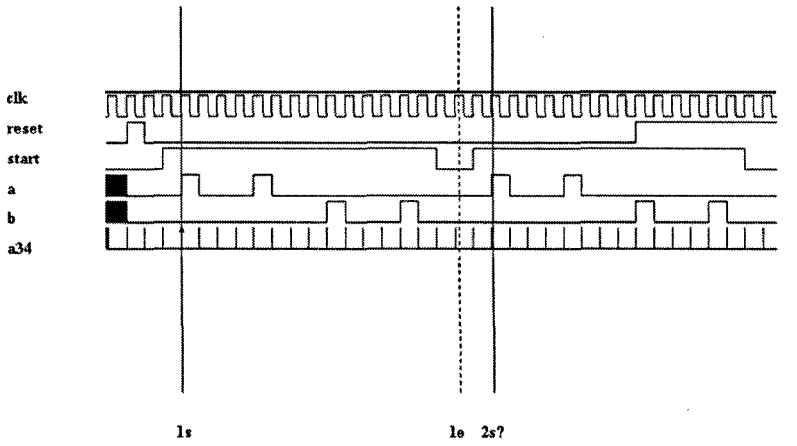

Figure 1-36. Waveform for SVA checker using "disable iff" construct

1.30 Using "intersect" to control length of the sequence

The **intersect** operator discussed in Section 1.23 can be used effectively to control the length of sequences, particularly in cases where the upper bound of the timing window is not defined. Whenever an eventuality operator is used, there is no restriction on the number of clock cycles that can be used by the checker to succeed. The **intersect** operator provides a mechanism to define the minimum and maximum number of clock cycles that can be used by the eventuality operator to succeed.

Property p35 defines a sequence that checks that on a given clock edge if signal "a" is high then eventually signal "b" should go high starting from the next clock cycle and eventually signal "c" should go high starting from the next clock cycle. This sequence will start whenever signal "a" is high and can take until the end of the simulation time to succeed. This is restricted by using the **intersect** operator 1[*2:5]. This **intersect** definition checks that from the starting point of the sequence match (high on signal "a") to the ending point of the sequence match (high on signal "c") it can take anywhere between 2 to 5 clock cycles.

```
Property p35;
   (@(posedge clk)  1[*3:5] intersect
                 (a ##[1:$] b ##[1:$] c));
endproperty

a35: assert property(p35);
```

Figure 1-37 shows how property p35 reacts in a simulation. Table 1-17 summarizes the sampled values of the relevant signals and shows the status of assertion a35. On a given clock edge if signal "a" is not detected high, it is a failure. This happens in several clock cycles (1, 3, 4, 5, 11 and 13) and these are not valid starts.

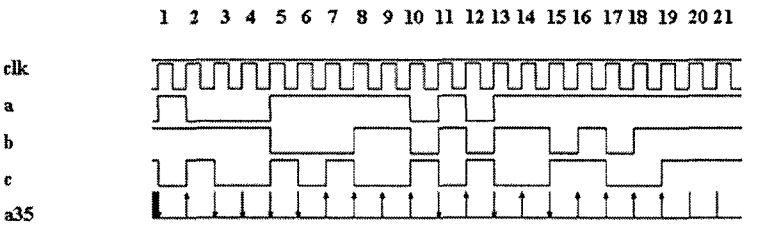

Figure 1-37. Waveform for SVA checker using intersect to control the length of the sequence

The check succeeds in several clock cycles (2, 6, 7, 8, 9, 10, 12 and 14). Note that the sequence takes 5 clock cycles or less from the start to the end point. The check has a real failure at clock cycle 6. Signal "a" is detected high at clock cycle 6 and signal "b" arrives at clock cycle 9. Signal "c" does not arrive at clock cycle 10, which completes the upper limit allowed for the length of the entire check. Hence, the check fails at clock 10. Note that signal "c" does arrive at clock cycle 11, but it's too late.

Table 1-17. Evaluation table for SVA checker using intersect operator to control the length of the sequence

Clock tick	Sampled value of "a"	Sampled value of "b"	Sampled value of "c"	Valid start	a35 status
1	0	1	1	No	Fail
2	1	1	0	Yes	Success (start at 2, end at 6)
3	0	1	1	No	Fail
4	0	1	0	No	Fail
5	0	1	0	No	Fail
6	1	0	1	Yes	Fail (start at 6, end at 10)
7	1	0	0	Yes	Success (start at 7, end at 11)
8	1	0	1	Yes	Success (start at 8, end at 11)
9	1	1	0	Yes	Success (start at 9, end at 11)
10	1	1	0	Yes	Success (start at 10, end at 13)
11	0	0	1	No	Fail
12	1	1	0	Yes	Success (start at 12, end at 16)
13	0	0	1	No	Fail
14	1	1	0	Yes	Success (start at 14, end at 16)
15	1	1	0	Yes	Fail
16	1	0	1	Yes	Success
17	1	1	1	Yes	Success

1.31 Using formal arguments in a property

Some of the common properties that can be re-used can be defined with formal arguments. Property "arb" takes 4 formal arguments and has a check defined on these formal arguments. The property is also bound to a specific clock. SVA allows clock definition as one of the formal arguments to the property. This way, the property can be bound to similar design block working with different clocks. Also, the timing delays specified can be parameterized to make the property definition very generic.

The property checks for a valid start first. On a given positive edge of the clock, if a falling edge of signal "a" is followed by the falling edge of signal "b" within 2 to 5 clock cycles, then it is a valid start.. If the antecedent matches, then the property checks for a falling edge on signal "c" and signal "d" on the next clock cycle and makes sure that these two signals stay low for 4 consecutive cycles. One cycle later, signal "c" and signal "d" should be detected high and one cycle after that signal b should be detected high.

Assuming that this is a protocol followed by an arbiter that deals with three different master devices with similar signals, the property can be re-used easily to check all three master interfaces. Assertions a36_1, a36_2 and a36_3 define the assertions for each master interface, using the signals relevant to each interface as the arguments for the property.

```
property arb (a, b, c, d);
  @(posedge clk) ($fell(a) ##[2:5] $fell(b)) |->
      ##1 ($fell(c) && $fell(d)) ##0
      (!c&&!d) [*4] ##1 (c&&d) ##1 b;
endproperty

a36_1: assert property(arb(a1, b1, c1, d1));
a36_2: assert property(arb(a2, b2, c2, d2));
a36_3: assert property(arb(a3, b3, c3, d3));
```

Figure 1-38 shows how the assertions defined for each interface react in a simulation. Assertion a36_1 has one valid start and it succeeds. Assertion a36_3 has one valid start and it fails.

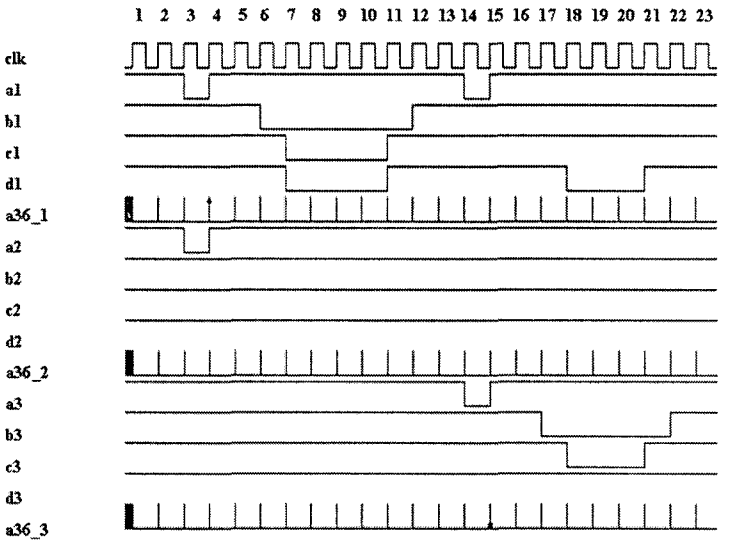

Figure 1-38. Waveform for SVA checker using formal arguments in a property

Success 1 (a36_1) - The check begins when a falling edge arrives on signal "a1" on clock cycle 4. This expects that a falling edge arrives on signal "b1" within 2 to 5 clock cycles and it does arrive on clock cycle 7. In the next clock cycle, signal "c1" and "d1" are low as expected. They should remain low for four cycles. They remain low from clock cycle 8 to 11. At clock cycle 12, both the signals "c1" and "d1" are high as expected. At clock cycle 13, signal "b1" is high as expected. Hence, the signal starts at clock cycle 4 and succeeds at clock cycle 13.

Failure 1 (a36_3) - The check begins when a falling edge arrives on signal "a3" on clock cycle 15. This expects that a falling edge arrives on signal "b3" within 2 to 5 clock cycles and it does arrive on clock cycle 18. In the next clock cycle, signal "c3" and "d3" are expected to be low. Since signal "d3" is not detected to be low, the check fails at clock cycle 19.

1.32 Nested implication

SVA allows having nested implications. These are useful when we have multiple gating conditions leading to a single final consequent.

Property p_nest checks that a valid start occurs if there is a falling edge on signal "a," then one cycle later, signals "b," "c" and "d" should all be active low to keep the valid start alive. If the second condition matches, then it is expected that within 6 to 10 cycles the condition "free" is true. Note that the consequent condition "true" is evaluated if and only if the signals "b," "c" and "d" match as expected.

```
`define free (a && b && c && d)

property p_nest;
   @(posedge clk) $fell(a) |->
                  ##1 (!b && !c && !d) |->
                                ##[6:10] `free;
endproperty

a_nest: assert property(p_nest);
```

The same property can be re-written without using the nested implication as follows.

```
property p_nest1;
   @(posedge clk) $fell(a) ##1 (!b && !c && !d)
```

```
                    |-> ##[6:10] `free;
endproperty

a_nest1: assert property(p_nest1);
```

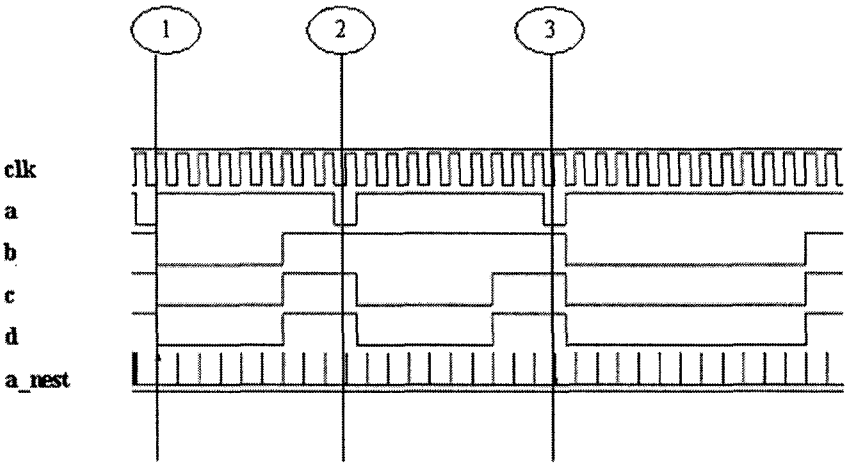

Figure 1-39. SVA checker with nested implication

Note that the nested implication property p_nest has no "else" condition and hence, the property can be easily re-written as shown in p_nest1.

Figure 1-39 shows how the assertion a_nest behaves in a simulation. Marker 1 shows the first success of the checker. A valid start occurs when a falling edge is detected on signal "a." One cycle later, signals "b," "c" and "d" are detected low as expected and hence the check is kept alive and the consequent gets evaluated. The condition "free" is detected true 6 clock cycles later and hence the check succeeds.

The second marker indicates the next valid start wherein a falling edge of signal "a" is detected. One cycle later, signals "c" and "d" are detected low but signal "b" is not low. Hence, the check is not active anymore and the check succeeds vacuously.

The third marker indicates a valid start wherein a falling edge of signal "a" is detected. One cycle later, signals "b," "c" and "d" are detected low as expected and hence the check is still active and the consequent gets

evaluated. The condition "free" is not detected true within 6 to 10 clock cycles after and hence the check fails.

1.33 Using if/else with implication

SVA allows the use of an "**if/else**" statement on the consequent of an implied property. Property p_if_else checks that a valid start occurs if a falling edge is detected on signal "start" and one clock cycle later either signal "a" or signal "b" is detected high. On a successful match of the antecedent, the consequent can take two possible paths.

1. If signal "a" is detected high, then, signal "c" should repeat twice intermittently and one cycle later signal "e" should be high.
2. If signal "a" is not high, then, signal "d" should repeat twice intermittently and one cycle later signal "f" should be high.

Note that there is a priority in the evaluation of the consequent for signal "a."

```
property p_if_else;
  @(posedge clk)
  ($fell(start) ##1 (a||b)) |->
    if(a)
      (c[->2] ##1 e)
    else
      (d[->2] ##1 f);
endproperty

a_if_else: assert property(p_if_else);
```

To re-write this property without using an "if/else" construct, three separate properties are required. A priority based "if/else" on two signals leads to three different possibilities as shown below.

a	b	Leaf
1	0	a
0	1	b
1	1	a

Note that if both signals "a" and "b" are high, then the "if" block of signal "a" is executed since it has priority. The three properties are shown below.

```
property p_if_else_leaf1;
 @(posedge clk)
       ($fell(start) ##1 a) |->
                          (c[->2] ##1 e);
endproperty

a_if_else_leaf1:
      assert property(p_if_else_leaf1);

property p_if_else_leaf2;
 @(posedge clk)
       ($fell(start) ##1 b) |->
                          (d[->2] ##1 f);
endproperty

a_if_else_leaf2:
      assert property(p_if_else_leaf2);

property p_if_else_leaf3;
 @(posedge clk)
       ($fell(start) ##1 (a &&b)) |->
                          (c[->2] ##1 e);
endproperty

a_if_else_leaf3:
      assert property(p_if_else_leaf3);
```

1.34 Multiple clock definitions in SVA

SVA allows a sequence or a property to have multiple clock definitions for sampling individual signals or sub-sequences. SVA will automatically synchronize between the clock domains used in the signals or sub-sequences. The following code shows a simple example of a sequence using multiple clocks.

```
sequence s_multiple_clocks;
     @(posedge clk1) a ##1 @(posedge clk2) b;
```

```
endsequence
```

The sequence s_multiple_clocks checks that, on a given positive edge of clock "clk1," signal "a" is high and then on a give positive edge of clock "clk2," signal "b" is high. The sequence matches when signal "a" is high on any given positive edge of clock "clk1." The ##1 delay construct will move the evaluation time to the nearest positive clock edge of clock "clk2" and then will check for signal "b" being high. ***When multiple clocked signals are used in a sequence, only ##1 delay construct is allowed.*** Re-writing the sequence s_multiple_clocks as follows is not allowed.

```
sequence s_multiple_clocks_illegal1;
     @(posedge clk1) a ##0 @(posedge clk2) b;
endsequence

sequence s_multiple_clocks_illegal2;
     @(posedge clk1) a ##2 @(posedge clk2) b;
endsequence
```

The use of ##0 will create confusion on which one is the nearest clock after the match on signal "a." This will create race conditions; hence, it is not allowed. The use of ##2 is not allowed since it is not possible to synchronize to the nearest positive clock edge of clock "clk2."

Similar techniques can be used to create properties with multiple clocks. The following code shows an example.

```
property p_multiple_clocks;
  @(posedge clk1) s1 ##1 @(posedge clk2) s2;
endproperty
```

It is assumed that the sequence s1 is not clocked or it has the same clock definition as "clk1." It is assumed that the sequence s2 is not clocked or it has the same clock definition as "clk2." The property can also have a non-overlapping implication operator in between the sequence definitions. A sample code is shown below.

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

The use of an overlapping implication operator between two multiple clocked sequences is not allowed. Since the end of the antecedent and the

beginning of the consequent overlaps, it can lead to race conditions; hence, it is illegal. The following code shows the illegal coding style.

```
property p_multiple_clocks_implied_illegal;
  @(posedge clk1) s1 |-> @(posedge clk2) s2;
endproperty
```

1.35 The "matched" construct

Whenever a sequence is defined with multiple clocks, the construct "**matched**" is used to detect the endpoint of the first sequence. Sequence s_a looks for a rising edge on the signal "a." Signal "a" is sampled based on the clock "clk1." Sequence s_b looks for a rising edge on the signal "b." Signal "b" is sampled based on the clock "clk2." The property p_match verifies that on a given positive edge of clock "clk2," if there is a match on sequence s_a, then one cycle later sequence s_b should be true.

```
sequence s_a;
  @(posedge clk1) $rose(a);
endsequence

sequence s_b;
  @(posedge clk2) $rose(b);
endsequence

property p_match;
  @(posedge clk2) s_a.matched |=> s_b;
endproperty

a_match: assert property(p_match);
```

Figure 1-40 shows how the assertion a_match behaves in a simulation. The property gets a valid start when there is a match on sequence s_a. Note that we are looking for this match on every positive edge of clock "clk2," though sequence s_a is sampled based on clock "clk1."

A valid rise on signal "a" happens at clock cycle 3 of "clk1." This updates the match value on sequence s_a to true. This value will be held until the nearest positive clock edge of "clk2." The nearest positive edge of "clk2" is at clock cycle 2 of "clk2." At this point the property becomes active and one clock cycle of "clk2" later, it is expected that the sequence

s_b matches. Hence, the first success of the property starts at clock cycle 2 of "clk2" and ends at clock cycle 3 of "clk2."

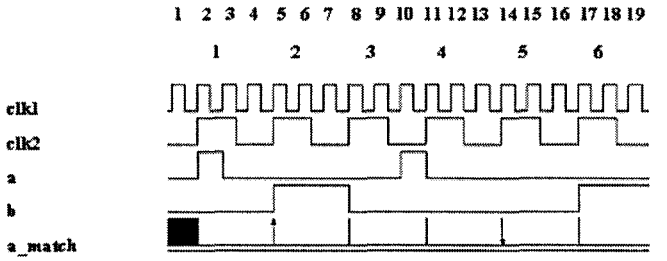

Figure 1-40. SVA checker using "matched" construct

Another valid rise on signal "a" happens at clock cycle 11 of "clk2" and this is sampled by the property at clock cycle 5 of "clk2." The property becomes active at this point and it is expected that in clock cycle 6 of "clk2," the sequence s_b match. But in this case, a rising edge of signal "b" does not occur and hence the property fails. ***The key concept to understand in using "matched" construct is that, the sampled match value is stored only until the next nearest clock edge of the other sequence.***

1.36 The "expect" construct

SVA supports a construct called "**expect,**" which is similar to the wait construct in Verilog, with the key difference being that the expect statement waits on the successful evaluation of a property. It acts as a blocking statement for the code that follows the **expect** construct. The syntax of the **expect** construct is very similar to the **assert** construct. ***The expect statement is allowed to have an action block upon the success or failure of the property.*** A sample code using the **expect** construct is shown below.

```
initial

begin
@(posedge clk);
#2ns cpu_ready = 1'b1;
expect(@(posedge clk) ##[1:16]
                    memory_ready == 1'b1)
```

```
    $display("Hand shake successful\n");
  else
    begin
    $display("Hand shake failed: exiting\n")
    $finish();
    end

  for(i=0; i<64; i++)
  begin
    send_packet();
    $display("PACKET %0d sent\n", i);
  end

  end
```

Note that after the signal "cpu_ready" is asserted, the **expect** statement waits for anywhere between 1 to 16 cycles for the signal "memory_ready" to be asserted. If the signal "memory_ready" is asserted as expected, a success message is displayed and the "for" loop code starts executing. If the signal "memory_ready" is not asserted as expected, then a failure message is displayed and the simulation exits.

1.37 SVA using local variables

A variable can be declared locally within a sequence or a property and an assignment can be made on that variable. The variable is placed next to a sub-sequence separated by a comma. If the sub-sequence matches, then the variable assignment is executed. Every time the sequence is attempted, a new copy of the variable is created.

```
property p_local_var1;
int lvar1;
@(posedge clk)
($rose(enable1), lvar1 = a) |->
          ##4 (aa == (lvar1*lvar1*lvar1));
endproperty

a_local_var1: assert property(p_local_var1);
```

The property p_local_var1 looks for a rising edge on the signal "enable1." Upon a match on this, the local variable "lvar1" stores the value

of the design vector "a." After 4 cycles, it is checked that the value of the design output vector "aa" is equal to the cubed value of the local variable. The consequent of the property waits for the design to satisfy the latency (4 clock cycles) and then compares the original design output with the locally calculated value. Figure 1-41 shows how the check reacts in a simulation.

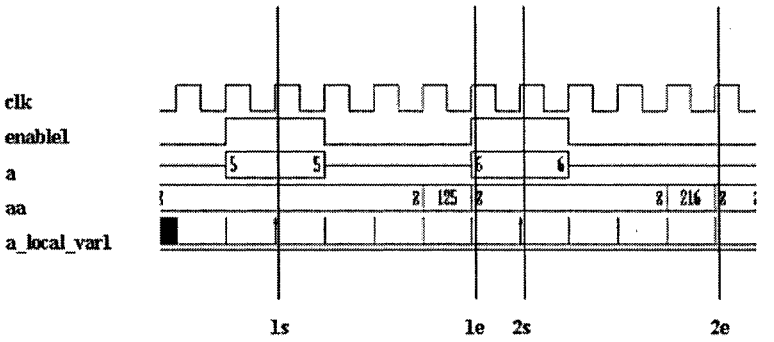

Figure 1-41. Waveform for SVA with local variables

The marker 1s shows the point where the rising edge of the signal "enable1" is sampled. At this point, vector "a" has a value of 5 and this is stored in the local variable "lvar1." The marker 1e shows the point where the output is sampled. This is 4 clock cycles after the input value was stored. At marker 1e, since the output value (125) equals that of the cube of the local variable "lvar1," the assertion succeeds. Similarly, marker 2s shows when the next input data is stored and marker 2e shows when the output is sampled and compared with the cubed value of local variable "lvar1."

The local variables can be stored and manipulated inside SVA.

```
property p_lvar_accum;
int lvar;
@(posedge clk) $rose(start)|=>
(enable1 ##2 enable2, lvar = lvar +aa) [*4]
##1 (stop && (aout == lvar));
endproperty

a_lvar_accum : assert property(p_lvar_accum);
```

The property p_lvar_accum checks for the following.

1. A valid start occurs if a rising edge is detected on the signal "start" on any given positive edge of the clock.
2. One cycle later, a specific pattern or a sub-sequence is looked for. The signal "enable1" should be detected high and 2 cycles later signal "enable2" should be detected high. This sub-sequence should repeat itself 4 times continuously.
3. For every repeat of the sub-sequence, the value of the vector "aa" is accumulated locally. At the end of the repetition, the local variable holds a value accumulated from the vector "aa" four times.
4. One cycle after the repetition, it is expected that the signal "stop" is detected high and the value held by the local variable is equal to the value held by the output vector "aout."

Figure 1-42 shows how the check reacts in a simulation.

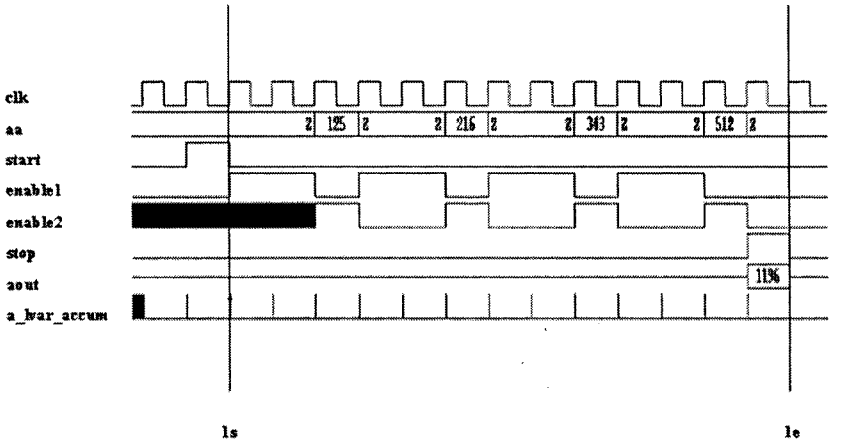

Figure 1-42. SVA with local variable assignment

The marker 1s shows a valid start of the check wherein the signal "start" is detected high. Marker 1e shows the end of the check. The repetitions of the enable signals complete successfully and one cycle later the signal "stop" is detected high as expected. The local variable holds the same value as that of the output vector "aout" and hence the check succeeds at the marker 1e.

1.38 SVA calling subroutine on a sequence match

SVA can also call a subroutine on every successful match of a sequence. The local variables defined in the same sequence can be passed as arguments to these subroutine calls. For each match on the sequence, the subroutine calls are executed in the same order as they are listed in the sequence definition.

```
sequence s_display1;
@(posedge clk)
($rose(a), $display("Signal a arrived at %t\n",
$time));
endsequence

sequence s_display2;
@(posedge clk)
($rose(b), $display("Signal b arrived at %t\n",
$time));
endsequence

property p_display_window;
@(posedge clk)
s_display1 |-> ##[2:5] s_display2;
endproperty

a_display_window :
      assert property(p_display_window);
```

Sequence s_display1 looks for a rising edge on the signal "a." Upon a match on this event, it executes the display statement. Sequence s_display2 does a similar action on signal "b." The property p_display_window checks that if sequence s_display1 occurs then the sequence s_display2 should occur anywhere between 2 and 5 clock cycles. By using display statements, the user can get information on exactly how many cycles the consequent sequence completed. Figure 1-43 shows how the check reacts in a simulation.

The marker 1s shows a valid start of the checker since a rising edge of the signal "a" is detected. At this point, SVA executes the display statement relevant to this sequence (s_display1). The marker 1e shows the point when a rising edge arrives on signal "b." Since this arrives after 3 cycles, the

checker succeeds. At this point, the display statement relevant to this sequence (s_display2) is executed.

The marker 2s shows a valid start of the checker since a rising edge of the signal "a" is detected. At this point, SVA executes the display statement relevant to this sequence (s_display1). The marker 2e shows the ending point of the checker. A valid rising edge never arrived on signal "b" within 2 and 5 clock cycles and hence, the checker failed. Since the second sequence never matched, the relevant display statement is not executed. A default error is issued by SVA.

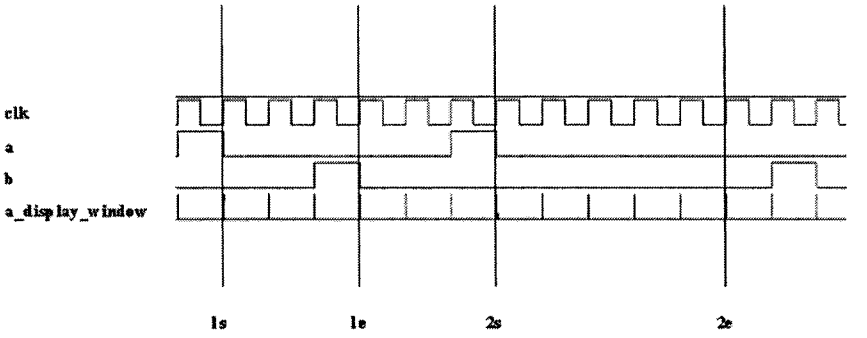

Figure 1-43. SVA using subroutines on sequence match

A sample simulation log is shown below.

```
Signal a arrived at                    125

Signal b arrived at                    275

"sub.v", 45: sub.a_display_window:
started at 125s succeeded at 275s

Signal a arrived at                    425

"sub.v", 45: sub.a_display_window:
started at 425s failed at 675s
        Offending '$rose(b)'
```

1.39 Connecting SVA to the design

SVA checkers can be connected to the design by two different methods.

1. Embed or in-line the checkers in the module definition.
2. Bind the checkers to a module, an instance of a module or multiple instances of a module.

Some engineers don't like adding any verification code within the design. In this case, binding the SVA checkers externally is the choice. SVA code can be embedded anywhere in a module definition. The following example shows SVA being in-lined within the module.

```
module inline(clk, a, b, d1, d2, d);

input logic clk, a, b;
input logic [7:0] d1, d2;
output logic [7:0] d;

always@(posedge clk)
begin
  if(a)
    d <= d1;
  if(b)
    d <= d2;
end

property p_mutex;
  @(posedge clk) not (a && b);
endproperty

a_mutex: assert property(p_mutex);

endmodule
```

If the user decides to keep the SVA checkers separate from the design, then he has to create a separate checker module. ***By defining a separate checker module, the re-usability of the checker increases. The following code shows a checker module.***

```
module mutex_chk(a, b, clk);
```

```
input logic a, b, clk;

property p_mutex;
  @(posedge clk) not (a && b);
endproperty

a_mutex: assert property(p_mutex);

endmodule
```

Note that when a checker module is defined, it is an independent entity. The checker is written for a generic set of signals. The checker can be bound to any module or instance in the design. The syntax for binding is as follows.

```
bind <module_name or instance name>
     <checker name> <checker instance name>
     <design signals>;
```

For the example checker shown above, the binding can be done as follows.

```
bind inline mutex_chk i2 (a, b, clk);
```

When the binding is done, the actual design signal names are used.

Let's say we have a top-level module as follows.

```
module top (..);

inline u1 (clk, a, b, in1, in2, out1);
inline u2 (clk, c, d, in3, in4, out2);

endmodule
```

The checker mutex_chk can be bound to the two instances of the module "inline" in the top-level module as follows.

```
bind top.u1 mutex_chk i1(a, b, clk);
bind top.u2 mutex_chk i2(c, d, clk);
```

The design signals that are bound can contain cross module reference to any signal within the scope of the bound instance.

1.40 SVA for functional coverage

Functional coverage is a metric for measuring verification status against design specification. It is classified into two categories:

a. Protocol coverage.
b. Test plan coverage.

Assertions can be used to get exhaustive information on protocol coverage. SVA provides a keyword "**cover**" to specify this. The basic syntax of a **cover** statement is as follows.

```
<cover_name> : cover property(property_name)
```

"cover_name" is a name provided by the user to identify the coverage statement and "property_name" is the name of the property on which the user wants to get coverage information. For example, the checker "mutex_chk" defined in Section 1.39 can be covered as follows.

```
c_mutex: cover property(p_mutex);
```

The results of the **cover** statement will provide the following information:

1. Number of times the property was attempted.
2. Number of times the property succeeded.
3. Number of times the property failed.
4. Number of times the property succeeded vacuously.

A sample coverage log from a simulation for the checker "mutex_chk" is shown below.

```
c_mutex, 12 attempts, 12 match, 0 vacuous match
```

Just like the assert statement, the cover statement can also have an action block. Upon a successful coverage match, a function or a task can be called or a local variable update can be performed.