SV ASSERTIONS

Introduction

- There are two significant pieces of technology that are used by almost all verification engineers.
  1. A constrained random test bench
  2. Code coverage tool.
- The test benches normally perform 3 different tasks 1. Stimulus generation.
  2. Self-checking mechanisms.
  3. Functional coverage measurement.
- Self-checking processes usually targets two specific areas 1. Protocol checking.
  2. Data checking.

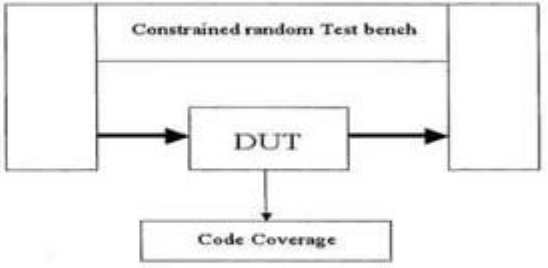


Fig: constrained random test bench without assertions.

- Functional coverage provides a measure of verification completeness. and it gives
  1. Protocol coverage. 2. Test plan coverage.

What is an Assertion?

- An assertion is a statement that a certain property must be true.
- Assertions are used to,
  - Document the functionality of the design
  - Check that the intent of the design is met over simulation time
  - Determine if verification tested the design (coverage)
- Assertions can be specified,
  - By the design engineer as part of the model
  - By the verification engineer as part of the test program
- Verilog does not provide an assertion construct.
  - Verification checks must be coded with programming statements.

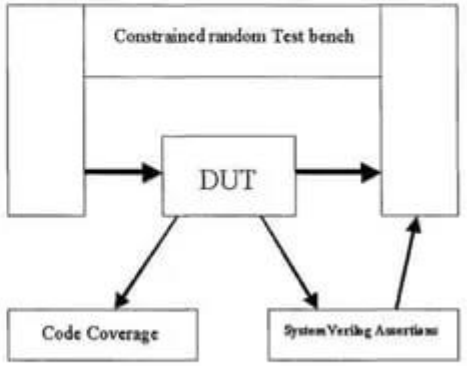


Fig: DUT with Assertions

Why using sv assertions important?

- It's a verification technique i.e. embedded in the language
  - Gives "white box" visibility into the design
- Enables specifying design requirement with assertions
  - can specify design requirements using an executable language
- Enables easier detecting of design problems
  - In simulation, design errors can be automatically detected
  - Error reports show when error occurred with insight as to why
  - Formal analysis tools can prove that the mode functionality does or does not match the assertion.
  - Can generate "counter-examples"(test cases) for assertion failures
- Enables constrained random verification with coverage.

➢ Assertion written in Verilog and SVA

➢ **VERILOG**

```
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
```

➢ **SV ASSERTION**

```
a_to_b_chk:assert property@(posedge elk) $rose(a) |-> ##[1:3] $ r o s e (b) );
```

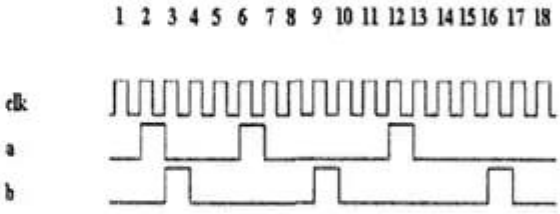


Fig. Waveform for sample assertion

System Verilog Scheduling

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 timeslot.

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.

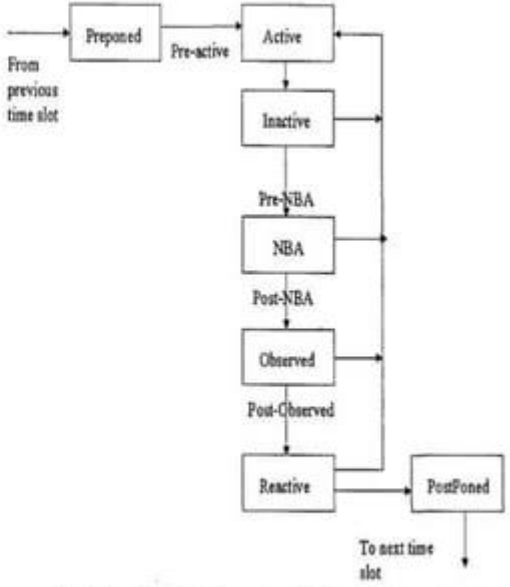


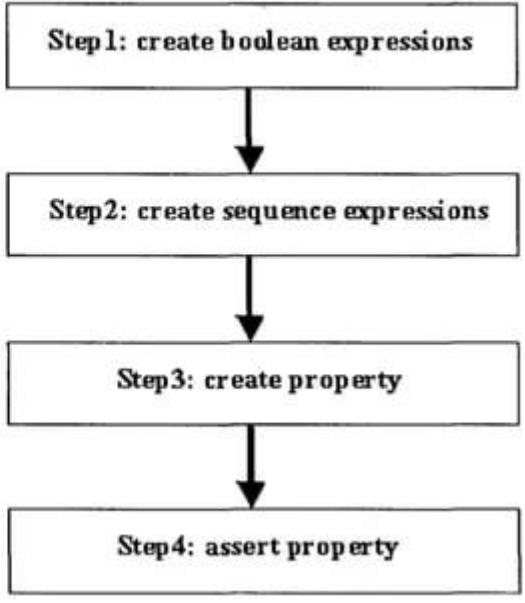
Fig. Simplified SV event schedule flow chart

Building blocks of SVA

sequence name_of_sequence;
<testexpression>;
endsequence

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

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



Types of assertions

Concurrent assertions

- Based on clock cycles.
- Used with both static&dynamic verification tools
- Placed in procedural block,module,an interface or program definition
- Evalution at clock edges
- Ex:

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

Immediate assertions

- Based on simulation event semantics.
- Used with dynamic verification tools
- Placed in procedural block definition
- evaluated just like any other Verilog expression within a procedural block.
- Ex:

```
always_comb
  b e g in
  a _ i a : a s s e r t (a && b ) ;
  end
```

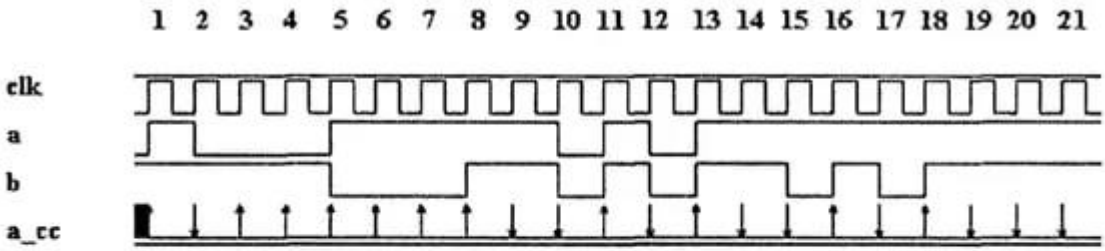


Fig: Waveform for a sample concurrent assertion

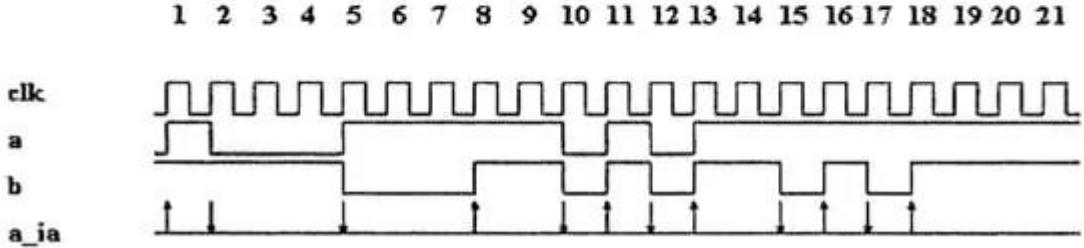


Fig: Waveform for a sample immediate assertion

A simple action block

- It can be used for controlling the simulation environment and gathering functional coverage data.

- user can also print a custom error or success message using the **"action block"** in the assert statement

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

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

- The checker ass shown above uses simple display statements in the action block to print successes and failures.

Assertion Severity Levels

- The assertion failure behavior can be specified

  $fatal: Implicit $finish call.

  $error: A run-time error severity; software continues execution.

  $warning: A run-time warning; software continues execution.

  $info: No severity, just print the message.

- Severity level is included in assertion failure message.
- Severity can optionally include further message information using $display syntax

```
always@(posedge clk)
ack_chk:assert!(acka&&ackb)
else
$error("%m failure");
```

```
always@(posedge enable)
Assert !(valid)
Else
Begin
 $warning;
$display("invalid enable");
 err_count++;
end
```

Sequence with edge definitions

- $rose:

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

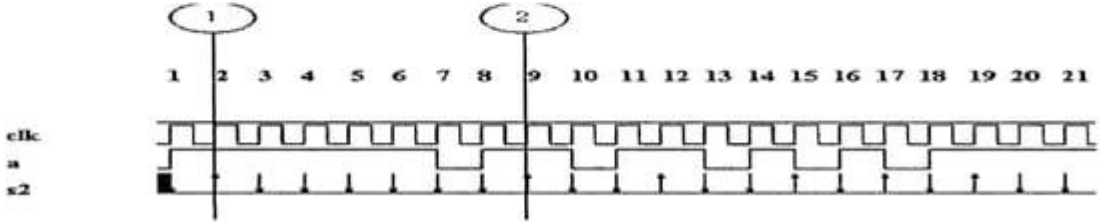


Fig: Waveform for $rose

Sequence with edge definitions

- **$fell:** $fell(Boolean expression or signal name) returns true if the least significant bit of the expression changed to 0. Otherwise, it returns false.

```
sequence seq_fell;
 @(posedge clk) $fell(a);
endsequence
```

Sequence seq_fell checks that the signal "a" transitions to a value of 0 on every positive edge of the clock. If the transition does not occur, the assertion will fail.

- **$stable:** $stable(Boolean expression or signal name) returns true if the value of the expression did not change. Otherwise, it returns false.

```
sequence seq_stable;
  @(posedge clk) $stable(a);
endsequence
```

Sequence seq_stable checks that the signal "a" is stable on every positive edge of the clock. If the there is any transition occur, the assertion will fail.

Sequence with edge definitions

- **$past:** $past(signal_name, number of clock cycles) provides the value of the signal from the previous clock cycle.
  -Below Property checks that, in the given positive clock edge, if the "b" is high, then 2 cycles before that, a was high.

```
property p;
 @(posedge clk) b |-> ( $past(a,2) == 1);
endproperty
a: assert property(p);
```

- **$past construct with gating signal:**

The $past construct can be used with a gating signal. on a given clock edge, the gating signal has to be true even before checking for the consequent condition.

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

Below Property checks that, in the given positive clock edge, if the "b" is high, then 2 cycles before that, a was high only if the gating signal "c' is valid on any given positive edge of the clock.

```
Property p;
 @(posedge clk) b |-> ($past(a,2,c) == 1);
endproperty
a: assert property(p);
```

Sequence with logical relationship

- Sequence h, 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.

- Ex: sequence h;
  @(posedge clk) a||b;
  endsequence

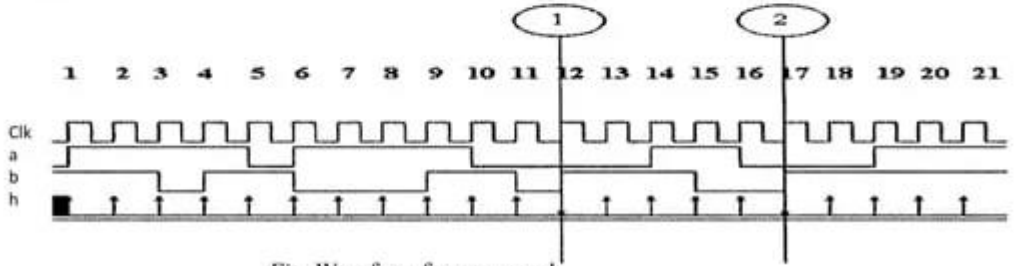


Fig: Waveform for sequence h

Clock definitions in SVA

- In general, 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.

Ex:

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

        property p;
        @(posedge clk) ha;
        endproperty
ast : assert property(p);
```

Forbidding a property

- sometimes we expect the property to be false always. If the property is true, the assertion fails.
- Ex: sequence ha;

```
@(posedge clk) a ##2 b;
 endsequenee
property v;
   not ha;
endproperty
vh : assert property(v);
```

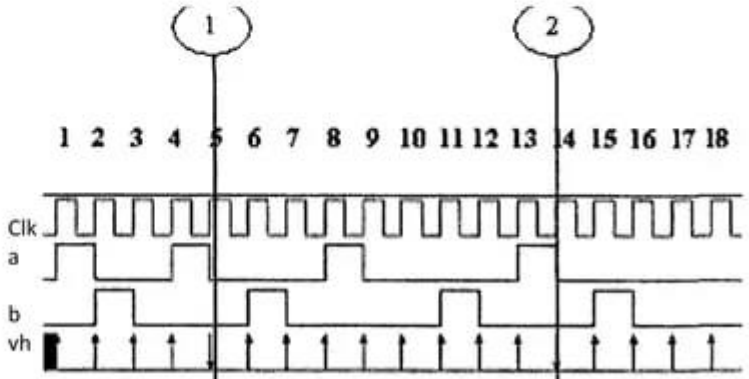


Fig: Waveform of SVA checker forbidding a property

Implication operator

- *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.
- The *implication construct* can be used only with property definitions. It *cannot be used in sequences*.
- There are 2 types of implication:1.Overlapped implication(|->) 2.Non-overlapped implication(|=>)

Overlapped implication

- If there is a match on the antecedent, then the consequent expression is evaluated in the same clock cycle.

- Ex: property pr;
  @(posedge elk) A |-> B;
  end property
  ass : assert property(pr);

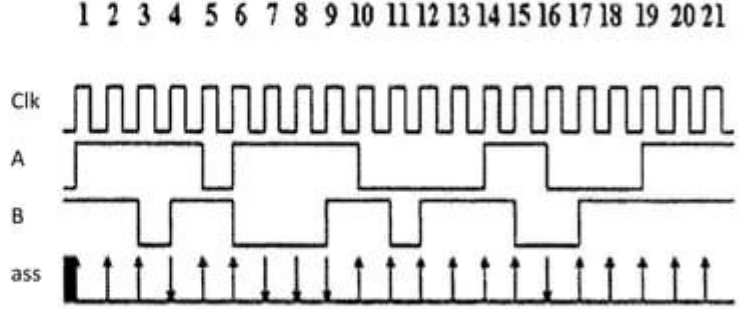


Fig: Waveform for property pr

- 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.

Non-overlapped implication(|=>)

- If there is a match on the antecedent, then the consequent expression is evaluated in the next clock cycle.

- Ex: property hr;

```
    @(posedge clk)  A|=>B;
  endproperty
ass : assert property(hr);
```

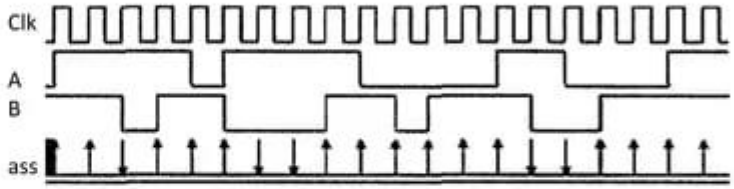


*Fig:*Waveform for property ass

- In above 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.

Timing windows in SVA Checkers

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

is : **assert property**(vs);

Follows like below

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

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

(a && b) |->##[3] c

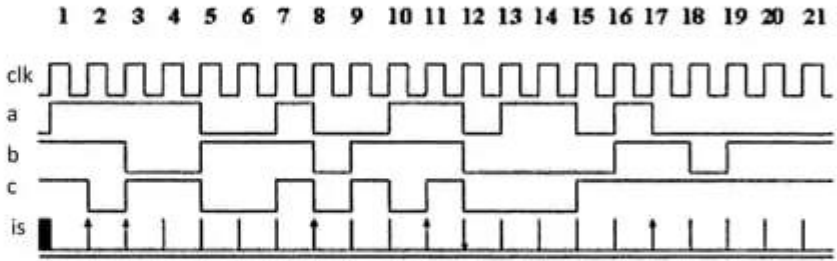


Fig: Waveform for property "is"

- Overlapping window gives `0' instead of 1 from above example(##[0:3]). So The value of signal "c" is detected high in the same clock edge as the valid match on the antecedent.
- Indefinite window gives $ instead of 3 from above example(##[1:$]). "$" sign which implies that there is no upper bound for timing.

Repetition operators

➢SVA language provides three different types of repetition operators.

1.**Consecutive repetition**: This allows the user to specify that a signal or a sequence will match continuously for the number of clocks specified. The simple syntax shown below.

signal or sequence [*n];

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

2.**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 main requirement of "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 shown below.

Signal [->n]

3.**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 shown as **Signal [=n]**

Consecutive repetition operator [*]

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

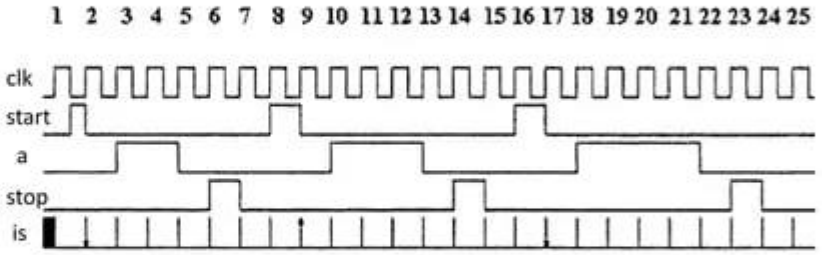


Fig: Waveform for SVA checker using consecutive repeat

- Property sr 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.

Go to repetition operator [->]

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

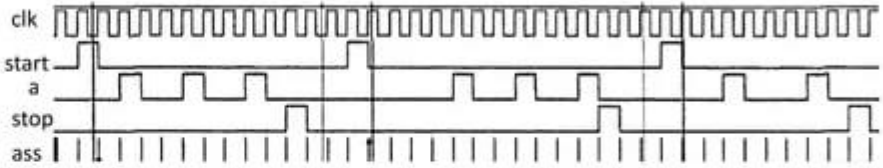


Fig: Waveform for SVA checker using go to repetition operator

- Property hr checks that, if there is a vahd 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.

Non-consecutive repetition operator [=]

```
Property pr;
@(posedge clk) $rose(start)|->##2 (a[=3]) ##1 stop ##1 !stop;
endproperty
ass: assert property(pr);
```

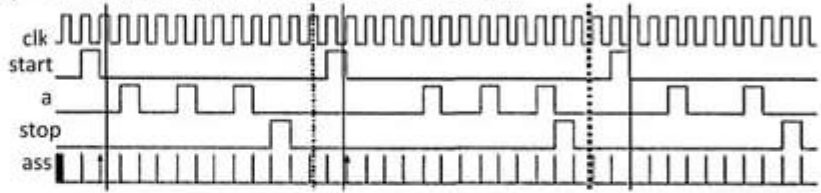


Fig: Waveform for SVA checker using non-consecutive repetition operator

- Property pr 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.

- Here there is no expectation that there is a valid match on signal "a" in the previous cycle of a valid match on "stop" signal.

The "and" ,"or" constructs

- The binary operators **"and", "or"** can be used to combine two sequences logically.
- Both sequences must have the same starting point but they can have different ending points.

```
sequence s1;
  @(posedge clk) a##[l:2] b;
endsequence
sequence s2 ;
  @(posedge clk) c##[2:3] d;
endsequence
property And;
  @(posedge clk) s1 and s2;
endproperty
and: assert property(And);
property Or;
  @(posedge clk) s1 or s2;
endproperty
or: assert property(Or);
```

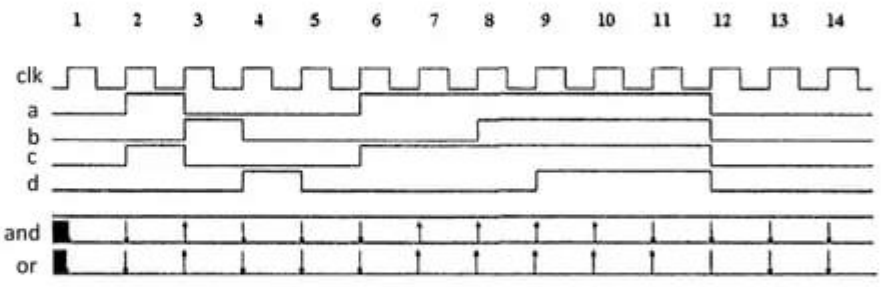


Fig: Waveform for SVA checker using "and" , "or" constructs

- Sequence s1 and s2 are two independent sequences. The property pr combines them with an **and** operator. The property succeeds when both the sequences succeed.

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.

```
sequence s1;
  @(posedge clk) a##[1:2] b;
endsequence
sequence s2 ;
  @(posedge clk) c##[2:3] d;
endsequence
property And;
  @(posedge clk) s1 and s2;
endproperty
and: assert property(And);
property Int;
@(posedge clk) s1 intersect s2;
• endproperty
int: assert property(Int);
```

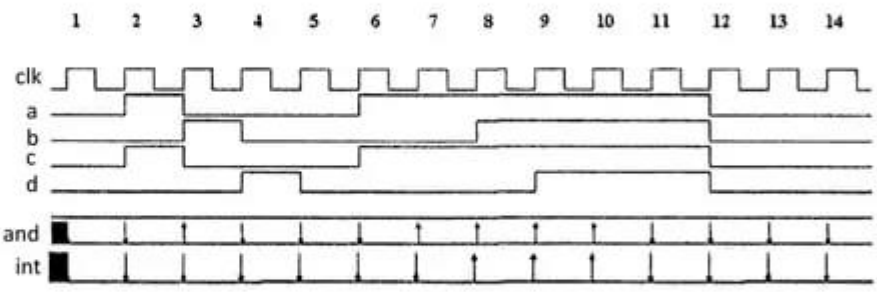


Fig: Waveform for SVA checker for intersect constructs

The "firstmatch" construct

- It is sometimes convenient to discard all sequence matches but the first one.
- The construct "**first_match**" ensures that only the first sequence match is used and the others are discarded.

```
sequence s1;
  @(posedge clk) a ##[1:3] b;
endsequence
sequence s2;
  @(posedge clk) c ##[2:3] d;
endsequence
property pr;
  @(posedge clk) first_match(s1 or s2);
endproperty
ass: assert property(pr) ;
```

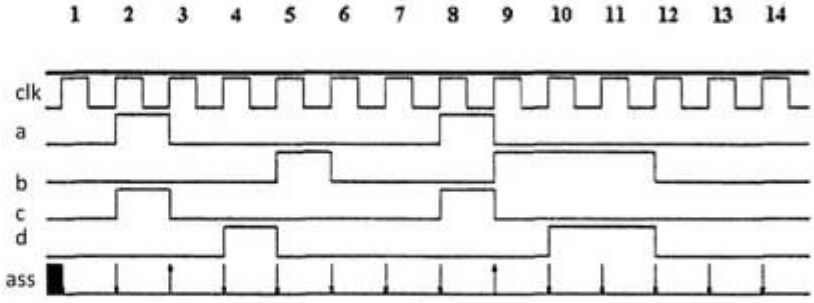


Fig: Waveform for SVA checker using "first_match" construct

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

The "throughout" construct

- Implication checks for precondition 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

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

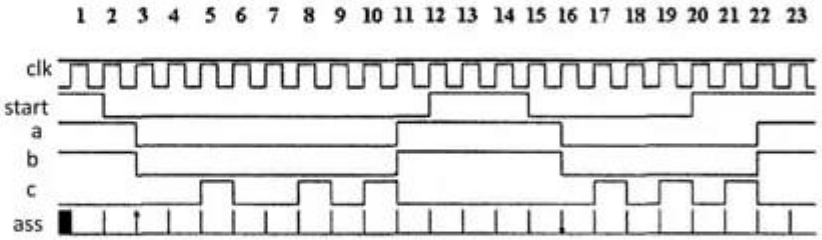


Fig: Waveform for SVA checker using "throughout" construct

- 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 below.

The "within" construct

- The containment of a sequence within another sequence can be expressed using within operator.
- syntax is given as seq1 **within** seq2.
- This means that seq1 happens within the start and completion of seq2.

- My_seq1 start with “rdy” and end with “done”, the design must have 8 “read”s.

```
sequence my_seq1;
        read[=8] within (rdy ##[9:15] done);
endsequence: my_seq1
```

The "disable iff " construct

- In certain design conditions, we don't want to proceed with the check if some condition is true. Like an asynchronous reset for the checker so SVA providing "disable iff".

```
property pr;
  @(posedge clk)
  disable iff (reset)
  $rose(start) |=> a[=2] ##1 b[=2] ##1 !start
endproperty
a34: assert property(pr);
```

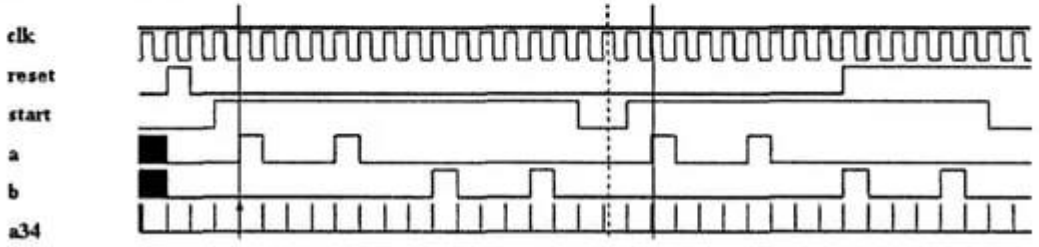
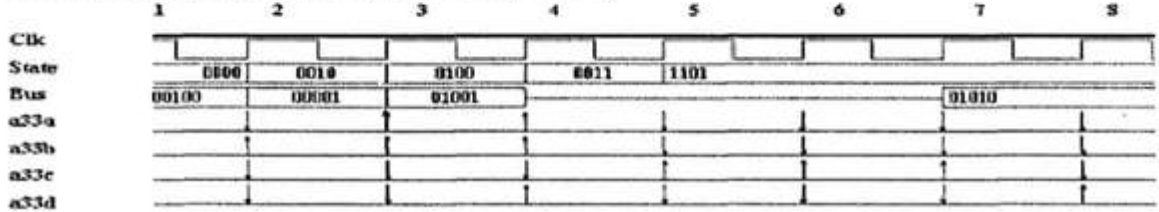


Fig: Waveform for SVA checker using "disable iff construct

Built-in system functions

- **$onehot(expression)** - checks that only one bit of the expression can be high on any given clock edge.
- **$onehot0(expression)** - checks only one bit of the expression can be high or none of the bits can be high on any given clock edge.
- **$isunknown(expresslon)** - checks if any bit of the expression is X or Z.
- **$countones(expression)**- counts the number of bits that are high in a vector.

- 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);



Connecting SVA to the design

- The bind directive can be specified in any of the following:
  - A module
  - An interface
  - A compilation-unit scope

- 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.

- The syntax for binding is as follows.
  - **bind** <module_name or instance name> <checker name> <checker instance name> <design signals>;

SVA for functional coverage

- SVA provides a keyword **"cover"** to specify protocol coverage.

- The basic syntax of a **cover** statement is as follows.
  - <cover_name> : **cover property**(property_name)

- 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 "mutexchk" is shown below.
  - c_mutex, 12 attempts, 12 match, 0 vacuous match