ASSERTIONS

* primarily used to validate the behaviour of a design

used case:-

Q. if signal 'start' has become high in one clock cycle. Then, it should be in high level for at least 4 clock cycles. It should not go low before 4 clock cycles?

Ans:- By using verilog checker, lot of codings required, in SV only single line of code

```
assert property (
    @(posedge clk) $rose(start)
        |-> start[*4:$] );
```

Advantage:-

1. CODE Compact when compare to Verilog
2. easily catching the bugs
3. increase Controlability & observability.

disadvantage:-

1. Still stimulus are required.
2. Declare carefully.

1. types of assertions

Immediate Assertion

* evaluted current simulation time.
* multiple execution leads to glitch's
* to avoid glitch use 'Differed Assertions'
  - use #0 or final keyword
* Declare in procedural block

syntax :-

1. immediate Assertion

```
always begin
  ass: assert(a&b);
end
```

2. deffered immediate

```
always begin
  ass: assert #0 (b!=a)
end
```

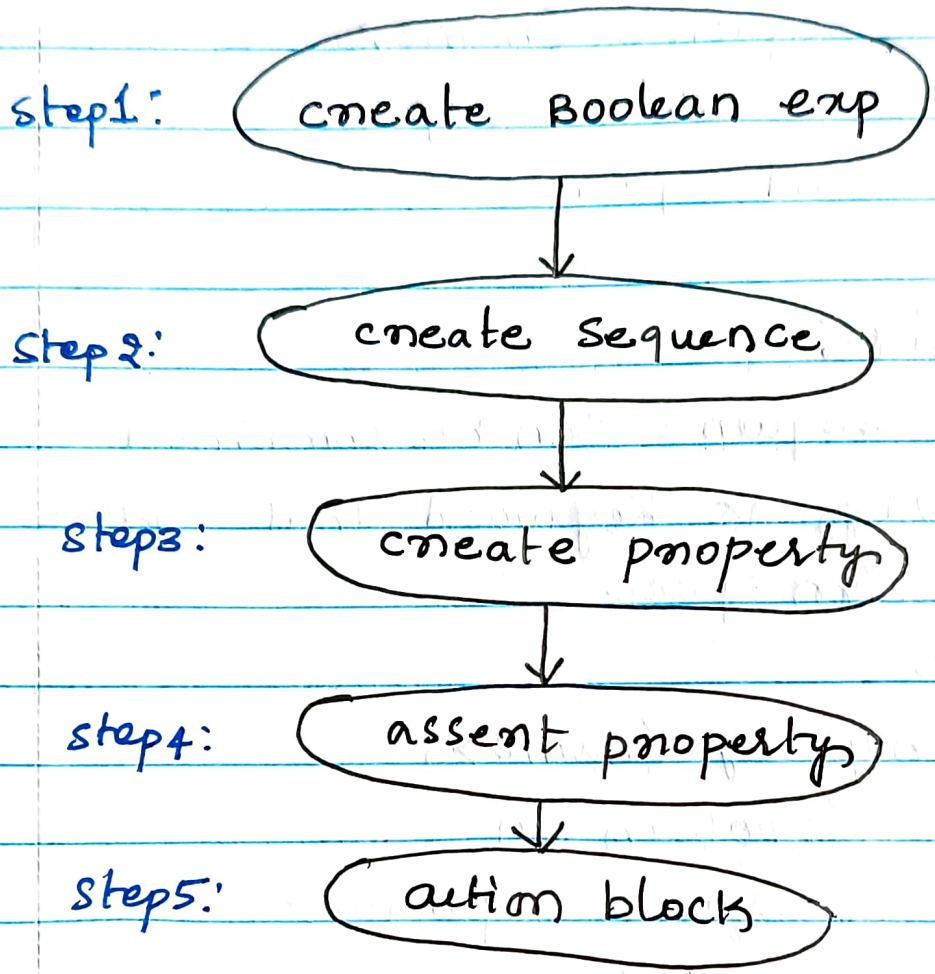
Concurrent Assertions

* Sampling of assertion done in preponed region evaluation happens in observed Region
* it required property keyword

syntax :-

```
property <name>;
  <Boolean or Sequence>
endproperty
```

2. Building Blocks of assertion.



1. Boolean exp :-

1. a ##2 b;
2. @(posedge clk) a ##3 b;
3. @(posedge clk) (A||B)[*3];

Sequence :-

1. Combination of multiple Boolean exp
2. clock tik delaly → "##"
3. It can be Declared in interface, clocking block, program or package.

Syntax :- sequence <name>;
Boolean exp;
endsequence.

iii. property :-

* No. of sequence can be combined
* property is keyword that captures the design specs.

Syntax :-

Property <name>;
<Boolean or
sequence exp name>
endproperty.

IV. Assert keyword :-

* to check property 'Assert' keyword used

Syntax :-

<assertion_name> : assert property
(<property name>);

V. Action Block :-

Syntax :-

```
sequence seq1;
    a##b;
endsequence

property prop;
    @(posedge clk) seq1;
endproperty

assert property (prop);
    $display(" passed");
else
    $display(" failed");
```

action block

Note :- Severity levels

1. $info :- sample information, like $display
2. $warning :- it give Runtime warning
3. $error :- Runtime error & this is default severity of assertions
4. $fatal :- it give runtime fatal & quit simulation.

3. System function (SVA)

1. Sequence with edge detection :-

* SVA have built in edge detection that allows the user to monitor the transition of signal value from one cycle to the next cycle.

1. $rose();
2. $fell();
3. $stable();
4. $changed();

1. $rose :- (0,x or z) anything to 1 transition

syntax:-

```
sequence seq;
  @(posedge clk) $rose(A);
endsequence
```

2. $fell :-

→ anything to 0 transition (1,x or z)

Syntax:-

```
sequence seq();
  @(posedge clk) $fell(A);
endsequence.
```

3. $stable :-

* This returns if the value of exp did not change in the current cycle and previous cycle

* syntax :-

```
sequence seq;
  @(posedge clk) $stable(A);
endsequence
```

4. $changed :- opposite to $stable.

Note :- some other system functions.

5. $countones ;-

* it will return the No. of 1's in A
* any value other than 1 is ignored.

6. $isunknown :-

* checks whether the bit of vector with x, z

7. $onehot :-

* Return 1 if, only 1-bit is high in an exp

8. $onehot0 :-

* Return 1, if all bits are zero only one bit is high.

2. Sequence with logical relationship :-

* Sequence can be developed by using logical operations like
  → &&, ||, ^

Syntax :-

```
sequence seq;
  @(posedge clk) A&&B;
endsequence
```

3. Sequence with expression :-

i. a ##1 b → A is high, after 1 clock B high

ii. a ##n b => n, after n clock B high.

iii. a ##[1:3] b => A is high, after 1 to 3 clocks B is high anywhere

iv. a ##[1:$] b => a is high, after 1 to ∞ clock b is high

v. a ##1 b[*3] => a is high, after 1 clock B is high & B should be const for 3 clock cycles

④ Forbidding a property :-

```
sequence seq;
    A ##2 B;
endsequence

property prop;
    not seq;
end property
```

→ this combination will not happen

---

4. Operator (SVA)

1. Implication operators :-

i. overlapping :-

* denoted by '|->'
* same cycle implication

ii. non-overlaping :-

* denoted with "|=>"
* Next cycle implication

```
property prop;
@(posedge clk) A |-> B;
endpropelty
```

```
property prop;
@(posedge clk)
    B |=> C;
endproperty
```

2. Repetition operaton :-

i. Consecutive :-

By Num :- [*n]

By Range :- [*min:max]

ii. non Consecutive :-

By Num :- [=n]

By Range :- [=min:max]

Note :-

1. @(posedge clk) $rose(A)

1 => ##1 (b[=2]) ##c

↓

c become high, anywhere after ~~(2 clock cycles) of~~ B high. often 2 clock cycles to avoid it we use go to operaton

3. go-to operator :-

* denoted with " -> "
* By Num : [->2]
* By range : [->min:max]

Syntax :-

```
property prop;
  @(posedge clk)
  $rose(A) |=> ##1(B[->2])##1 S;
endproperty
```

Path: SV/ Functional Coverage/

Functional Coverage :-

* use to measure the Quality of design verification.

→ To create coverage model we use covergroup and as part of coven group we define covenpoint for every variables, we cross the covenpoints and can define bins explicitly, if not simulator will explicitly creates itself (we call it as automatically bins (default = 64)

coven group :-

* To create covenage model we use coven group
* used inside module or named block.
* It may contain :
  1. clocking event
  2. coverpoints
  3. cross covenage b/w them.

covenpoints :-

* we define a covenagepoints for the model (each variable)

bins :-

* keep track of Number of times usen hits specific value of covenpoint expression.

1. implicit bins :-

* If we do not specify any bins, SV simulator will create bins automatically

Eg :- (i) bit [3:0] data;

↓

2^4 = 16 bins → 0 to 15

(ii) bit [7:0] data;

↓

2^8 = 256 bins → 0 to 255

Note :- if → option.auto_bin_max = 256

otherwise if → B > 64 → grouping

B <= 64 → individual

grouping :- $\frac{256}{64}$ = 4 bins each

data[0:3] → 1st bin

data[4:7] → 2nd bin

⋮

data[252:255] → 64th bin

CODE :-

```
covergroup cg;
 cp: coverpoint A;
 cp: coverpoint B;
endgroup
```

2. Explicit bins :-

* We can create :
  → separate bin for each value in a given range
  → A single bin for all the values in a range/s

*** Scalar bins :-**
→ single bin for all values

→ ~~data[ ] = {1,2,5};~~

* data = {1, 2, 5};

Vector bins :-
→ unique bin is created for each value

* data[ ] = {[0:5]};

Other exp :-

1. b = {1, 2, 3, 4, 5}; → scalar bin
2. b[ ] = {[1:5]}; → vector bin
3. b[ ] = {[$:7]}; → min to 7
4. b[ ] = {[0:$]}; → 0 to max
5. b = {[1:7], 8, 9}; → scalar with num's

(iii) ignore bins & illegal bins :-

1. ignorebins :-

* ignorebins are bins which is ignored from coverage

* Syntax :-

```
covergroup cg;
  cp: Coverpoint A {
        ignore_bins ib = {1,2};
                    }
endgroup
```

(ii) Illegal bins :-

* Illegal bins also same ignore bins, but when bins are hitting we will get an error message. (for every bit)

* Syntax :-

```
coven group cg;
  cp : Coverpoint A {
        illegal_bins ilb = {1,2};
                    }
endgroup
```

(iv) Cross Coverage:

* Keyword : cross
* A cross is defined to track the values of two or more Coverpoints as a group.
* Syntax :-

```
cross A, B ;
cross cp1, cp2;
cross cp1, A ;
```

* crosses can be applied:
  - → predefined Coverpoints
  - → Variables which are visible in the scope.
  - → Combinations of Coverpoint & Variable.

CODE :-

```
covergroup cg;
    cp1 : coverpoint A ;
    cp2 : Coverpoint B;
    cp1xcp2 : cross A, B;
endgroup : cg
```

Reusable Covergroup :-

```
module reusable;
  logic [3:0] mode;

  Covergroup cg (ref logic[3:0] Var_name,
                 input int lower, higher,
                 input string instance_name);
    option.per_instance = 1;
    option.name = instance_name;

    cp: coverpoint mode {
          bins b[] = {[lower:higher]};
        }
  endgroup: cg

  cg c1, c2, c3, c4;
  initial begin
    c1 = new(mode, 0, 3, "Arth");
    c2 = new(mode, 4, 7, "logical");
    c3 = new(mode, 8, 11, "shift");
    c4 = new(mode, 12, 15, "Jump");
  end
endmodule
```

conditional exp iff :-

```
covergroup cg;
  cp: covenpoint queue iff(rst);
  cp1: covenpoint Arrays iff(!empty);
endgroup: cg
```

(✓) transition bins :-

* The transition of covenage point can be covened by specifying the sequence
  → val1 => val2 => val3
* it nepnesent transition of covenage point value from value1 to value2.

syntax :-

bins b1 = (10 => 20 => 30); → 10 → 20 → 30

bins b2[] = (40 => 50), (80 => 90 => 100);

bins b3 = (1,5 => 6,7);

b[0] = 40 → 50

b[1] = 80 → 90 → 100

b3 = 1 → 6, or
1 → 7, or
5 → 6, or
5 → 7

Covergroup built-in methods :-

* sample();
* get_coverage();
* get_inst_coverage();
* get_inst_name();
* start();
* stop();

(VI) binsof & intersect :-

bin of

if particular coverpoint in cond

value of of bins of point

→ cp1-cp2 : cross A, nst {

binsof(nst) intersect {1};

}

—: CODE COVERAGE :—

* it measures how much of the "design code" is exercised.

* This includes the execution of design blocks, Number of lines, conditions, FSM, Toggles and path

* the simulator will automatically extract the code coverage from the design code