Detailed SystemVerilog Assertions (SVA) Guide for Beginners

SystemVerilog Assertions (SVA) are a powerful way to specify design behavior and check it during simulation or formal verification. This guide will explain SVA in detail with examples to help you understand how to write effective assertions.

1. Types of Assertions in SystemVerilog

-
  ```
  Syntax:
  assert (condition) [pass_action] else
  [fail_action];
  ```

-
  ```
  Example :
  always @(posedge clk) begin
      assert (a && b)
          $display("Assertion passed: a and b are
  high");
      else
          $error("Assertion failed: a or b is low");
  end
  ```

  - If a && b is true, the pass message is printed.
  - If false, an error is reported.

1.2 Concurrent Assertions

- Evaluated over time based on clock edges.
- Used to check temporal (time-based) behavior.

-
  ```
  Syntax:
  assert property (@(event) property_expression);
  ```

-
  ```
  Example:
  assert property (@(posedge clk) req |-> ##[1:3]
  ack);
  ```

  - This means:
    - If req is high, then (|→) within 1 to 3 cycles (## [1:3]), ack must be high.
    - If not, the assertion fails.

2. Building Blocks of SVA

Category	Key Elements	Description

Category	Key Elements	Description
Sequences	##n, [*n], [→n], and, or	Temporal behavior over clock cycles (e.g., req ##2 ack).
Properties	`	→,⇒, always, eventually` Logical conditions built from sequences (e.g., `req -> ##[1:3] ack`).
Assertions	assert, assume, cover	- assert: Checks if property holds. - cover: Tracks occurrence.
Clocking	@(posedge clk)	Specifies when assertions are evaluated (e.g., clock edges).
System Functions	$rose(), $fell(), $past()	Helper functions for edge detection and past values.
let & Recursion	let declarations	Reusable expressions or recursive properties for complex checks.

SVA Flow Diagram:

```
+---------------------+
|  Boolean Expressions |  ◄┐
```

We use cookies to provide you a better user experience on this website.

Cookie Policy

Only essentials I agree

```
  +---------------------+    |
            |                |
            ▼                |
                             |
  +---------------------+    |
  |      Properties     |    | (Implication,
repetition)
  | (|->, |=>, always)  |    |
  +---------------------+    |
            |                |
            ▼                |
                             |
  +---------------------+    |
  |     Assertions      |    | (Check/Enforce)
  | (assert, cover)     |    |
  +---------------------+    |
            |                |
            ▼                |
                             |
  +---------------------+    |
  |    Clocking &       |    | (Timing control)
  |  System Functions   |────┘
  | ($rose, $past, @clk)|
  +---------------------+
```

Key Takeaways:

1. Boolean expressions form the base conditions.
2. Sequences define multi-cycle behaviors.
3. Properties combine sequences with operators like |→.
4. Assertions enforce properties in simulation/formal verification.
5. Clocking & functions tie everything to the design's timing.

3. Key SVA Operators

3.1 Sequence Operators

Sequences describe a series of events over time.

Operator	Description	Example

Operator	Description	Example
##[min:max]	Variable delay range	a ##[1:3] b → a now, b in 1-3 cycles
[*n]	Consecutive repetition	a[*3] → a must be true for 3 cycles in a row
[→n]	Non-consecutive "goto" repetition	a[→3] → a must be true exactly 3 times (not necessarily consecutive)
[=n]	Non-consecutive, allows extra occurrences	a[=3] → a must be true at least 3 times

Example:

```
sequence s_data_ready;
    start ##1 data_valid[*2] ##1 done;
endsequence
```

- This means:
  1. start is high.
  2. After 1 cycle, data_valid must stay high for 2 cycles.
  3. Then, after 1 more cycle, done must be high.

3.1.1 Basic Delay Operators

Operator	Description	Example	Meaning

Operator	Description	Example	Meaning
			exactly 3 cycles later
##[min:max]	Variable delay range	a ##[1:4] b	a true now, b must be true between 1-4 cycles later
##0	Zero-cycle delay (same cycle)	a ##0 b	Both a and b must be true simultaneously

Example:

```
sequence s_pipeline;
    instr_decode ##2 execute ##1 writeback;
endsequence
// Decode now, execute after 2 cycles, writeback after
1 more cycle
```

3.1.2 Repetition Operators

Key Difference:

- a[*3] = a ##1 a ##1 a (strictly consecutive)
- a[→3] = ... a ... a ... a ... (anywhere, but exactly 3 times)
- a[=3] = allows extra as (e.g., a a a a still matches [=3])

Example:

```
sequence s_interrupt;
    $rose(irq) ##1 irq[->3] ##1 $fell(irq);
endsequence
```

We use cookies to provide you a better user experience on this website.

Cookie Policy

Only essentials I agree

```
// Consecutive
sequence s_burst;
    burst_start ##1 data_valid[*4] ##1 burst_end;
endsequence

// Non-consecutive
sequence s_sparse_events;
    event1[->2] ##1 event2[=1];
endsequence
```

3.2 Property Operators(Logical Relationships)

Properties define expected behavior using sequences.

Operator	Description	Example		
`	→`	Overlapping implication (check immediately)	`a	→ b→ Ifais true,b` must be true same cycle
`	⇒`	Non-overlapping implication (next cycle)	`a	⇒ b→ Ifais true,b` must be true next cycle
and	Both sequences must hold	seq1 and seq2		

Operator	Description	Example	
not	Sequence must never occur	not (a ##1 b)	

Example:

```
property p_valid_transaction;
    @(posedge clk)
    start |=> (data_valid[*2] ##1 done);
endproperty
```

- When start is high, then in the next cycle, data_valid must stay high for 2 cycles, followed by done.

3.2.1 Implication Operators

Operator	Description	Example	Timing		
`	→`	Overlapping implication	`a	→ b`	If a is true, check b immediately
`	⇒`	Non-overlapping implication	`a	⇒ b`	If a is true, check b next cycle

Visualization:

```
|-> (Overlapping):
Cycle: 1   2   3
       a   b
```

```
        a       b
        |_______|
```

Example:

```
// Overlapping
property p_overlap;
    @(posedge clk) req |-> grant;
endproperty

// Non-overlapping
property p_next_cycle;
    @(posedge clk) req |=> grant;
endproperty
```

3.2.2 Logical Operators

Operator	Description	Example
and	Both sequences must succeed	seq1 and seq2
or	Either sequence must succeed	seq1 or seq2
not	Sequence must never occur	not (a ##1 b)
intersect	Both sequences must succeed simultaneously	seq1 intersect seq2

Example:

```
property p_mutex;
    @(posedge clk) not (bus_request1 and
property p_data_hold;
    @(posedge clk)
    $rose(valid) |-> $stable(data) throughout
ready[*4];
endproperty
```

3.3. System Functions

3.3.1 Edge Detection

Function	Description	Example
$rose(sig)	True on 0→1 transition	$rose(enable)
$fell(sig)	True on 1→0 transition	$fell(reset)
$stable(sig)	True if no change	$stable(address)

3.3.2 Past Values

Function	Description	Example
$past(signal, n)	Value of signal n cycles ago	$past(data, 2)

Example:

```
assert property (@(posedge clk)
    (data_valid && $past(data_valid, 2)) |-> data ==
$past(data, 1));
// If valid now and 2 cycles ago, current data must
```

```
sequence s_axi_burst;
    $rose(arvalid) ##0 arready ##1
    arvalid[*4] intersect arready[*4] ##1
    $fell(arvalid);
endsequence
```

Sequence Breakdown:

1. $rose(arvalid) ##0 arready:
   - The sequence starts when arvalid has a rising edge (transition from 0 to 1)
   - At the exact same cycle (##0), arready must also be high
   - This represents the handshake that initiates the address phase of the transaction
2. ##1 arvalid[*4] intersect arready[*4]:
   - After 1 clock cycle (##1), we check for:
     - arvalid staying high for exactly 4 consecutive cycles ([*4])
     - Simultaneously (intersect), arready must also stay high for exactly 4 consecutive cycles
   - This represents a burst transfer of 4 data beats where both valid and ready remain asserted
3. ##1 $fell(arvalid):
   - After the 4-cycle burst, in the next cycle (##1), arvalid must fall (transition from 1 to 0)
   - This indicates the end of the transaction

What This Sequence Checks:

This sequence verifies a complete 4-beat burst read transaction on the AXI AR channel:

1. Proper handshake initiation (arvalid and arready both high)
2. Continuous transfer for exactly 4 cycles (both signals remain high)
3. Proper termination (arvalid goes low after the burst)

```
    s_axi_burst;
endproperty

assert property (p_axi_burst);
```

This ensures that when a 4-beat burst read starts, it follows this specific protocol pattern.

3.4.2 PCIe TLP Packet

```
sequence s_tlp_header;
    sof ##1 header[->3] ##1 eof;
endsequence
```

Sequence Breakdown:

1. sof:
   - The sequence starts with the Start of Frame (SOF) signal being asserted
   - This marks the beginning of a TLP packet
2. ##1 header[→3]:
   - After 1 clock cycle (##1), we look for:
     - The header signal occurring 3 times ([→3])
     - The → is a "goto" operator that matches the specified number of occurrences regardless of the number of intervening cycles
   - This represents the TLP header which typically spans 3 or 4 DWORDS (depending on packet type) in PCIe
3. ##1 eof:
   - After the header is complete, in the next cycle (##1), we expect:
     - End of Frame (EOF) signal to be asserted
   - This marks the completion of the TLP packet

What This Sequence Checks:

This sequence verifies the basic structure of a TLP packet:

Key Notes:

- The → operator means there can be any number of cycles between header occurrences, but they must appear in order
- This is more flexible than [*3] which would require consecutive cycles
- The sequence doesn't specify what happens with the payload (if any) - it just checks the header structure

Typical Usage:

This would likely be used in PCIe verification to ensure proper TLP header formation:

```
property p_tlp_header_correct;
    @(posedge clk) disable iff (!reset_n)
    s_tlp_header;
endproperty

assert property (p_tlp_header_correct);
```

This ensures that when a TLP packet starts, it follows this header structure pattern before ending.

3.4.3 FIFO Overflow Protection

```
property p_fifo_safe;
    @(posedge clk)
    fifo_count > (DEPTH-2) |-> not fifo_write;
endproperty
```

This is a SystemVerilog Assertion (SVA) property that enforces a safety condition for a FIFO (First-In-First-Out) buffer to prevent overflow.

Property Breakdown:

1. Clock Specification:
    - @(posedge clk): The property is evaluated on every rising

- DEPTH is presumably a parameter defining the total capacity of the FIFO
- The condition triggers when the FIFO is nearly full (either 1 or 2 slots remaining)

3. Implication Operator:
   - |->: The overlapping implication operator means:
     - If the antecedent is true at a given clock edge, then the consequent must also be true at the same clock edge
4. Consequent (Required Behavior):
   - not fifo_write:
     - When the FIFO is nearly full, the fifo_write signal must be low (not asserted)
     - This prevents further writes that could cause overflow

What This Property Checks:

This assertion ensures that:

- When the FIFO reaches a near-full state (either 1 or 2 empty slots remaining, depending on whether the inequality is strict)
- The system must not attempt to write new data (fifo_write must be deasserted)
- This prevents FIFO overflow conditions

Practical Implications:

- DEPTH-2 creates a 2-entry safety margin (could be adjusted based on design needs)
- The property helps catch bugs where:
  - The write logic doesn't properly respect FIFO capacity
  - The fifo_count isn't being calculated correctly
  - Flow control isn't working properly

Typical Usage:

This would be used with:

```
assert property (@(posedge clk) disable iff (!reset_n)
p_fifo_safe);
```

This is a common type of assertion for FIFO control logic verification.

3.5. Operator Precedence Rules

1. ##, [*], [→], [=] (highest)
2. not, and, or
3. |→, |⇒ (lowest)

Use parentheses to clarify:

```
// Ambiguous:
a ##1 b or c ##1 d
// Clear:
(a ##1 b) or (c ##1 d)
```

3.6. Common Pitfalls

1. Overlapping vs Non-overlapping Confusion
   - |→ checks immediately, |⇒ checks next cycle
2. Repetition Operator Misuse
   - a[→3] requires exactly 3 occurrences (no more, no less)
3. Clock Domain Issues
   - Never mix clocks in a single assertion
4. Reset Handling
   - Always use disable iff for asynchronous resets

```
assert property (@(posedge clk) disable iff (reset) (a
|-> b));
```

3.7. Practical Tips

1. Debugging Assertions
   - Use $display in sequences:

2. Parameterized Assertions

```
property p_delay_check(int min, max);
    a ##[min:max] b;
endproperty
```

3. Assertion Libraries
    - Create reusable assertion templates for common protocols (AXI, APB, etc.)

4.Handshake Assertions Example

Protocol Rules:

1. When VALID goes high, it must stay high until READY occurs.
2. READY can occur only when VALID is high (no spurious READY).
3. Data must remain stable while VALID is high and READY is low.

SVA Implementation

4.1. Boolean Expressions

```
logic valid, ready;
logic [31:0] data;
```

4.2. Sequences

- Sequence for valid-hold:

```
sequence valid_hold_seq;
  valid ##1 (valid throughout (ready [->1]));
endsequence
```

  (Means: Once valid is high, it stays high until ready arrives.)
- Sequence for data stability:

```
property axi_handshake_prop;
  // Rule 1: VALID must hold until READY
  $rose(valid) |-> valid_hold_seq;

  // Rule 2: No READY without VALID
  ready |-> valid;

  // Rule 3: Data stability during handshake
  valid && !ready |=> $stable(data);
endproperty
```

4.4. Assertions & Cover

```
// Assert the handshake rules
assert property (@(posedge clk) axi_handshake_prop)
  else $error("AXI handshake violation!");

// Cover successful handshake
cover property (@(posedge clk) valid ##1 ready);
```

4.5. System Functions & Clocking

```
// Check if VALID rises without past READY
assert property (@(posedge clk) $rose(valid) |->
$past(!ready));
```

Flow Diagram for Handshake SVA

```
  +---------------------+
  |   Boolean Signals   |  (valid, ready, data)
  +---------------------+
            |
            ▼
  +---------------------+
  |      Sequences      |  ◄— valid_hold_seq,
data_stable_seq
  +---------------------+
            |
            |
            ▼
  +---------------------+
  |     Assertions      |  ← assert, cover
  +---------------------+
            |
            ▼
  +---------------------+
  |   Clock & Functions |  ← @(posedge clk), $rose,
$stable
  +---------------------+
```

Key SVA Operators Used

Operator	Purpose	Example in AXI		
##n	Delay	valid ##1 ready		
`	→`	Overlapped implication	`$rose(valid) →	valid_hold_seq`
throughout	Condition holds during sequence	valid throughout (ready [->1])		
$stable	No signal change	$stable(data) throughout (valid && !ready)		
$rose	Rising edge	$rose(valid)		

- Verify data changes during valid && !ready.

2. Use cover to ensure all handshake scenarios occur.

5. Cover Properties (Coverage Checks)

Cover properties track whether certain scenarios occur in simulation.

```
cover property (@(posedge clk) req ##[1:10] ack);
```

- Monitors if ack arrives 1-10 cycles after req.

```
cover property (@(posedge clk) $rose(interrupt));
```

- Tracks how many times an interrupt occurs.

6. Best Practices for Writing Good SVAs

1. Name Assertions Clearly
   - Example: assert_fifo_not_overflow instead of assert1.
2. Use disable iff for Resets
   ```
   assert property (@(posedge clk) disable iff (reset)
   (a |-> b));
   ```
   - Disables the check during reset.
3. Keep Assertions Simple
   - Break complex checks into smaller sequences.
4. Use Cover Points
   - Ensure all important scenarios are tested.
5. Bind Assertions to Modules
   ```
   bind fifo_module fifo_assertions
   fifo_assert_inst(.*);
   ```

- Immediate Assertions → Simple checks inside procedural blocks.
- Concurrent Assertions → Time-based checks (most common in SVA).
- Sequences → Describe a series of events.
- Properties → Define expected behavior using sequences.
- Cover Properties → Track scenario occurrences.

in **SVA**

P **Prachi Goyal** **5 April 2025**

SHARE THIS POST

TAGS

OUR BLOGS

PCIE

AXI

perforce

SVA

coverage

Home About Us

Get in touch