madanrajcr | madan raj c r | madanrajcr@gmail.com

System Verilog assertions examples.

Asserts that if in changes from 0 to 1 between one rising clock and the next, detect must be 1 on the following clock.(with and without using the active low reset).

```
property example1;
(@posedge clk) $rose(in) |=>  detect;
endproperty

assert property(example1);

property example2;
(@posedge clk) disable_iff(!resetn)
$rose(in) |=> detect;
endproperty

assert property(example2);
```

Active-low reset is preferred because it is hardware-friendly, safer at power-up, more noise-immune, and aligns with industry-standard reset architectures. Not using it can increase area, risk glitches, and cause unreliable initialization.

When enable is low, the input in must remain stable and must not change across clock cycles.
When enable is high, a valid 0→1 transition on in between two rising clock edges must cause detect to assert high on the following clock cycle.

```
property example3;
(@posedge clk) disable_iff(!resetn)
(!enable) |-> $stable(in)
and
((enable && $rose(a)) |=> detect);
endproperty

assert property(example3);
```

***"and" does not merge the sequences, it simply requires both sub-properties to hold independently at the same clock evaluation.**
if either sub-property fails, the whole combined assertion fails. This is why it's safe to combine protocol and functional rules in one property.*

When the request signal req is asserted, it shall remain high for exactly three consecutive clock cycles, during which the acknowledge signal ack must remain low. Immediately after these three cycles, ack shall assert high for exactly two clock cycles, independent of req. Following the acknowledgment period, both req and ack must return to their inactive (low) states.

```
property example4;
(@posedge clk)$rose(req) |-> (req  && (!ack))[*3] ##1
ack [*2] ##1
(!req && !ack);
endproperty

assert property(example4);
```

When the request signal req is asserted, the design shall monitor the ready signal rdy. Upon the first occurrence of rdy after req is asserted, the acknowledge signal ack must be asserted within 10 clock cycles.

```
property example5;
(@posedge clk) disable_iff(! resetn)
req |-> first_match [0:$] rdy |-> ##[0:10] ack;
endproperty

assert property(example5);
```

At any positive edge of the clock, if signal b is asserted high, then signal a must have been high exactly two clock cycles earlier, provided that the gating signal c is valid (high) on the corresponding clock edge.

```
property example6;
(@posedge clk) disable_iff(! resetn)
$rose(b) |-> $past ((a ,2 ,c) ==1);
endproperty

assert property(example6);
```

At any positive edge of the clock, if signal b is asserted high, then signal a must have been high exactly two clock cycles earlier, provided that the gating signal c is valid (high) on the same clock edge.

```
property example7;
(@posedge clk) disable_iff ( ! resetn )
( b && c ) |-> $past ( (a,2) == 1 );
endproperty
```

assert property(example7);

Same clock edge:
On a given positive clock edge, all signals in the assertion are sampled simultaneously. The condition and its check occur on the same edge, with any history referenced using $past.

Corresponding clock edge:
An event on one clock edge is related to behavior on a different (past or future) clockedge. Temporal operators like $past, ##, or $rose define how the check spans across clock cycles.

At a positive edge of the clock, if signal a is asserted high, then signal b must be asserted high for the next three consecutive clock cycles. After b has been high for the third consecutive cycle, signal c must assert high on the immediately following clock edge. If reset becomes high at any time during this entire sequence, the assertion checking shall be disabled and terminated.

```
property example8;
( @posedge clk) disable_iff( !resetn)
a |-> b[*3] ##1 c ;
endproperty

assert property(example8);
```

The sequence begins on the rising edge of the clock when the enable signal en rises, and once started, en must remain high. Throughout every two-clock cycle interval in which both req and valid are true, en is required to stay high for all clock ticks within that interval. This check must be repeated for five consecutive intervals, ensuring that the enable signal consistently remains high throughout all periods where valid requests occur.

```
property example9;
(@posedge clk) disable_iff ( !resetn )
$rose( en ) |-> (en throughout (req && valid) [*2]) [*5]
endproperty

assert property(example9);
```

even you can make use of the repeat statement also.

***In SystemVerilog Assertions, the until construct specifies that a sequence continues to hold until a particular condition becomes true. The key point is that the terminating condition must eventually occur, but it does not need to coincide with the last element of the sequence. For example, consider a signal req that represents a request and a signal ack that represents acknowledgment.**

```
property p_until;
@(posedge clk) (req [*0:$]) until ack;
endproperty

assert property(p_until);
```

This means that req can remain high for any number of clock cycles, but the sequence is considered complete only when ack eventually goes high. The ack signal does not have to align with the last high cycle of req; it can occur afterward. This is useful when we are only concerned that the request persists until acknowledgment eventually arrives, without strict timing constraints.

***The until_with construct is similar to until but introduces stricter timing. Here, the terminating condition must occur on the same clock cycle as the last element of the sequence. Using the same signals, the property can be written as.**

```
property p_until_with;
@(posedge clk) (req [*0:$]) until_with ack;
endproperty

assert property(p_until_with);
```

This property ensures that the last asserted req coincides exactly with ack on the same rising edge of the clock. Unlike until, ack cannot occur later; it must be synchronized with the final repetition of req. This is particularly important in scenarios where simultaneous assertion is required, such as a handshake that must complete on the same clock cycle.

***The throughout construct, by contrast, enforces that a signal remains true continuously for the duration of another sequence. Unlike until or until_with, it does not have a termination requirement but ensures persistent behavior. For example, if we want an enable signal en to remain high whenever both req and valid are high, the property can be written as.**

```
property p_throughout;
@(posedge clk) en throughout (req && valid);
endproperty

assert property(p_throughout);
```

This guarantees that en stays high for every clock cycle during which both req and valid are asserted. It is useful when continuous stability of a signal is required during certain operational conditions.
