1. **When signal_a is asserted, signal_b must be asserted, and must remain up until one of the signals signal_c or signal_d is asserted.**

```
property signal_a_implies_signal_b_until_c_or_d;
  @(posedge clk) disable_iff(!reset)
    ($rose(signal_a) |-> (signal_b ##1 until (signal_c || signal_d)));
endproperty
```

$rose(signal_a) checks for a rising edge of signal_a.

|-> denotes implication, stating that if there is a rising edge of signal_a, then the following condition should hold.

(signal_b ##1 until (signal_c || signal_d)) specifies that signal_b must be asserted (signal_b) and must remain asserted (##1) until either signal_c or signal_d is asserted.

2. **After signal_a is asserted, signal_b must be deasserted, and must stay down until the next signal_a.**

```
property signal_b_deasserted_until_next_signal_a;
  @(posedge clk) disable_iff(!reset)
    ($rose(signal_a) |-> (!signal_b ##1 until ($stable(signal_a))));
endproperty
```

|-> denotes implication, stating that if there is a rising edge of signal_a, then the following condition should hold.

(!signal_b ##1 until ($stable(signal_a))) specifies that signal_b must be deasserted (!signal_b ##1 until) and must stay deasserted until the next rising edge of signal_a.

3. **A request should be followed two cycles later by a rising edge of acknowledge. The ack is only allowed to be high for one clock cycle.**

```
property request_followed_by_ack;
  @(posedge clk) disable_iff(!reset)
    (request |-> ##2 $rose(ack) && ##1 !$changed(ack));
endproperty
```

(request |-> ##2 $rose(ack) && ##1 !$changed(ack)) checks that if request is asserted, then two cycles later (##2), there should be a rising edge of ack ($rose(ack)) and ack should only be high for one clock cycle (##1 !$changed(ack)).

4. **If signal_a is received while signal_b is inactive, then on the next cycle signal_c must be inactive, and signal_b must be asserted.**

```
property signal_a_received_while_b_inactive;
  @(posedge clk) disable_iff(!reset)
    (($changed(signal_a) && !signal_b) |-> ##1 (!signal_c && signal_b));
endproperty
```

```
($changed(signal_a) && !signal_b) checks if signal_a has changed (gone from 0
to 1) and signal_b is currently inactive.

|-> denotes implication. It specifies that if the condition above is true,
then the following conditions should hold in the next cycle (##1):

(!signal_c && signal_b) states that signal_c must be inactive, and signal_b
must be asserted.
```

5. **signal_a must not rise before the first signal_b.**

```
property no_rise_before_first_b;
  @(posedge clk) disable_iff(!reset)
    (!($rose(signal_a) && !($past(signal_b, 1))));
endproperty
```

```
!($rose(signal_a) && !($past(signal_b, 1))) checks that signal_a does not
rise (transition from 0 to 1) before the first occurrence of signal_b.

$past(signal_b, 1) captures the value of signal_b in the previous cycle, and
!($rose(signal_a) && !($past(signal_b, 1))) ensures that signal_a doesn't
rise while signal_b is still inactive.
```

6. **Write an assertion to check divide by 2 circuit output.**

```
property divide_by_2_assertion;
  // Property triggered on the positive edge of the clock
  @(posedge clock)

  // Disable the property check during reset
  disable iff (reset)

  // Check that the current state 'q' is not equal to its previous state
  (q != $past(q, 1));
endproperty
```

```
(q != $past(q, 1)): This part of the property checks that the current state q
is not equal to its previous state ($past(q, 1)). This condition ensures that
the output toggles between different states on each rising edge of the clock.
```

7. **When the positive edge of signal "a" is detected, check signal "b" has to be high continuously until signal "c" goes low.**

```
property pt;
  // Assertion triggered on the positive edge of the clock
  @(posedge clock)

  // Check: When the positive edge of signal "a" is detected... // "b" has to
be high continuously until signal "c" goes low
  $rose(a) |->  (b throughout (!c[->1]));
endproperty
```

`@(posedge clock)`: This triggers the assertion on the positive edge of the clock.

`$rose(a) |->`: This checks that when the positive edge of signal "a" is detected, the condition following it should hold.

`(b throughout (!c[->1]))`: This specifies that "b" has to be high continuously (throughout) until signal "c" goes low (!c[->1]).

8. **Whenever valid signal goes high enable signal should be asserted in the next cycle & it should be stable till ready signal is asserted. The ready signal should be asserted after enable with in 4 to 6 cycles.**

```
property ppt;
  // Assertion triggered on the positive edge of the clock
  @(posedge clock)

  // Check: Whenever valid signal goes high...
  $rose(valid) |-> enable[*4:6] ##1 ready;
endproperty
```

`@(posedge clock)`: This specifies that the assertion is triggered on the positive edge of the clock.

`$rose(valid) |->`: This checks that whenever the valid signal goes from 0 to 1 (positive edge or rising edge), the conditions following it should hold.

`enable[*4:6] ##1 ready;`: This part specifies the conditions that should be true after the rising edge of the valid signal:

`enable[*4:6]`: The enable signal should be asserted continuously for a duration of 4 to 6 clock cycles.
`##1`: After this duration, there should be exactly one clock cycle.
`ready;`: The ready signal should be asserted in this clock cycle.

9. **Whenever the signal A goes high from the next cycle the signal B should repeat n no. of times, where n is equal to value of bit[3:0]C when signal A is asserted.**

```
property cyc;
  // Declare integer variables local and count
  int local, count;

  // Assertion triggered on the positive edge of the clock
  @(posedge clock)

  // Check: If this condition holds...  // Implication: Then this condition
must also hold
  ($rose(a), local = c, count = 0)  |=> $rose(b) ##1(b, count++)[*1 : $] ##1
(!b && (count == local-1));

endproperty
```

`@(posedge clock)`: This triggers the assertion on the positive edge of the clock.

`($rose(a), local = c, count = 0) |=>`: This sets up initial conditions when signal a rises. It initializes local to the value of c and sets count to 0.

`$rose(b) ##1 (b, count++)[*1 : $] ##1 (!b && (count == local-1))`: This part of the assertion checks that after the rising edge of b, there should be a sequence of rising edges of b `(##1 (b, count++)[*1 : $])` where count is incremented. The sequence should continue until there's a falling edge of b `(##1 (!b && (count == local-1)))`. This enforces a specific relationship between the count of rising edges of b and the value of local.

10. Req must eventually be followed by ack, which must be followed 1 cycle later by done.

```
property req_followed_by_ack_and_done;
  @(posedge clk) disable_iff(!reset)
    (req |-> ##[1:$] ack && ##1 done);
endproperty
```

`(req |-> ##[1:$] ack && ##1 done)` checks that if req is asserted, then eventually `(##[1:$])` it should be followed by ack, and ack should be followed 1 cycle later by done.