1. **Write an assertion to make sure that a 5-bit grant signal only has one bit set at any time?**

```
// Property: Ensure only one bit is set in the 5-bit grant signal
property one_bit_set_in_grant;
  @(posedge clk)
  disable_iff(!reset)
  $onehot(grant);
endproperty
```

property one_bit_set_in_grant;: This line starts the definition of the property named one_bit_set_in_grant.

@(posedge clk): The assertion is triggered on the positive edge of the clk signal.
disable_iff(!reset): The assertion is disabled when the reset signal is active (low).

$onehot(grant): This is a SystemVerilog function that checks if only one bit is set (1) in the 5-bit signal grant. If this condition is true, the property is satisfied.

2. **Write an assertion which checks that once a valid request is asserted by the master, the arbiter provides a grant within 2 to 5 clock cycles.**

```
// Property: Once a valid request is asserted, the arbiter provides a grant
within 2 to 5 clock cycles
property grant_timing_after_request;
  @(posedge clk)
  disable_iff(!reset)
  $rose(master_request) |-> ##[2:5] arbiter_grant;
endproperty
```

@(posedge clk): The assertion is triggered on the positive edge of the clk signal.

disable_iff(!reset): The assertion is disabled when the reset signal is active (low).

$rose(master_request): This part of the assertion checks that the master_request signal rises (goes from low to high) at the current clock edge.

|->: This is the "assertion implication operator," which means "if the condition on the left-hand side is true, then the action on the right-hand side should be executed."

##[2:5] arbiter_grant;: This part of the assertion checks that the arbiter_grant signal is asserted within 2 to 5 clock cycles after the rising edge of master_request.

3. How can you disable an assertion during active reset time?

```
// Property: Once a valid request is asserted, the arbiter provides a grant
within 2 to 5 clock cycles
property grant_timing_after_request;
  @(posedge clk)
  disable_iff(!reset)
  $rose(master_request) |-> ##[2:5] arbiter_grant;
endproperty
```

`@(posedge clk)`: The assertion is triggered on the positive `edge` of the clk signal.

`disable_iff(!reset)`: The assertion is disabled when the reset signal is active (low).

`$rose(master_request)`: This part of the assertion checks that the master_request signal rises (goes from low to high) at the current clock `edge`.

`|->`: This is the "assertion implication operator," which means "if the condition on the left-hand side is true, then the action on the right-hand side should be executed."

`##[2:5] arbiter_grant;`: This part of the assertion checks that the arbiter_grant signal is asserted `within` 2 to 5 clock cycles after the rising `edge` of master_request.

4. How can all assertion be turned off during simulation (with active assertions)?

```
//design_module
module design_module(
    input logic enable,
    input logic reset,
    input logic clk,
    output logic active
);

    logic [1:0] fsm_cs; // current state
    logic [1:0] fsm_ns; // next state

    always_ff @(posedge clk or posedge reset) begin
        if (reset)
            fsm_cs <= 2'b00;
        else
            fsm_cs <= fsm_ns;
    end

    always_comb begin
        case (fsm_cs)
            2'b00: if (enable && !reset)
                       fsm_ns = 2'b01;
            2'b01: if (!enable || reset)
```

```
                        fsm_ns = 2'b00;
            default: fsm_ns = 2'b00;
        endcase
    end

    assign active = (fsm_ns == 2'b01) ? 1'b1 : 1'b0;

    // Property: Assert on the rising edge of the clock
    property p_dummy();
        @(posedge clk)
        $display("Property p_dummy Passed");
    endproperty

    // Assertion: Assert on the rising edge of the clock using the property
    DUMMY_ASSERT : assert property (p_dummy)
        else $error("Assertion DUMMY_ASSERT failed");
endmodule


//top
module top();
    logic en, reset, tester_clk, active_out;

    initial begin
        tester_clk = 1'b0;
        forever #10 tester_clk = ~tester_clk;
    end

    design_module mydut(
        .enable(en),
        .reset(reset),
        .clk(tester_clk),
        .active(active_out)
    );

    // Switching off specific named assertion
    initial begin
        $assertoff(0, top.mydut.DUMMY_ASSERT);
    end
endmodule
```

The initial block in the top module uses $assertoff to disable the specific named assertion DUMMY_ASSERT within the design_module. This means that this assertion will not be active during simulation.

$assertoff takes two arguments: the first is the instance number (0 in this case, assuming there's only one instance), and the second is the assertion handle (top.mydut.DUMMY_ASSERT), which uniquely identifies the specific assertion.

This usage of $assertoff is specific to simulation and does not affect synthesis.

Disabling assertions selectively using $assertoff can be useful in situations where certain assertions are not relevant for a particular test scenario or when debugging specific issues.

5. **In a RESP operation, request must be true immediately, grant must be true 3 clock cycles later, followed by request being false, and then grant being false.**

```
property resp_operation_property;
  @(posedge clk) disable_iff(!reset)
    ($rose(request) && ##3 $rose(grant) && ##1 !$changed(request) && ##1
!$changed(grant));
endproperty
```

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

##3 $rose(grant) specifies that three clock cycles later, there should be a rising edge of grant.

##1 !$changed(request) ensures that after the rising edge of grant, request should become false in the next cycle.

##1 !$changed(grant) ensures that after the falling edge of request, grant should become false in the next cycle.

6. **The signals signal_a and signal_b may only be asserted if signal_c is asserted.**

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

(signal_c |-> (signal_a && signal_b) && (!$changed(signal_a) && !$changed(signal_b))) expresses that if signal_c is asserted, then both signal_a and signal_b must be asserted (signal_a && signal_b).

Additionally, it ensures that once signal_c is asserted, there are no changes ($changed) in signal_a and signal_b until the next positive clock edge.

7. **Request must true at the current cycle; grant must become true sometime between 1 cycle after request and the end of time.**

```
property request_and_later_grant;
  @(posedge clk) disable_iff(!reset)
    (request && ##1 grant) throughout $past(request, 1);
endproperty
```

(request && ##1 (grant)) throughout $past(request, 1) checks that request is true at the current cycle, and there exists a future cycle where grant becomes true, and this condition holds throughout the time from the previous cycle ($past(request, 1)) until the end of time.

8. Enable must remain true throughout the entire ack to done sequence.

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

`(req |-> ##[1:$] ack && ##1 done) |-> (enable throughout (ack && done))` checks that **if** req is asserted, then **eventually** `(##[1:$])` it should be followed by ack, **and** ack should be followed 1 cycle later by done. The entire **sequence** is then followed by the assertion that enable must remain true **throughout** the entire ack to done **sequence**.

9. Write an assertion: A high for 5 cycles and B high after 4 continuous highs of A and finally both A and B are high?


```
property a_high_for_5_and_b_after_4_continuous_highs;
  @(posedge clk) disable_iff(!reset)
    (a[*5] && (a |=> ##4 b) && (a && b));
endproperty
```

`(a[*5] && (a |=> ##4 b) && (a && b))` checks that a is high **for** 5 cycles `(a[*5])`, **and** after 4 continuous highs of a, b becomes high `(a |=> ##4 b)`, **and** finally, both a **and** b are high `(a && b)`.

10. Write an assertion to verify the clock with 50% duty cycle of any given time period.

```
property time_period(int clk_period);
  time current_time;

  @(posedge clock)

  // Check: If this condition holds...  // Implication: Then this condition
must also hold
  (`1, current_time = $time)  |=> (clk_period/2 == $time - current_time);

endproperty
```

`(`1, current_time=$time) |=>`: This sets the current **time** (current_time) to the simulation **time** (`$time`) at the positive **edge** of the clock.

`(clk_period/2 == $time - current_time)`: This checks that the **time** period between consecutive positive edges of the clock is equal to half of the specified clock period `(clk_period)`.

11. When signal "d" changes to 1, on next cycle, if signal "b" is true, then signal "c" should be high continuously or intermittently for 2 clock cycles, followed by high on signal "a" in the next cycle else the signal "a" should be high continuously or intermittently for 2 clock cycles, followed by high on signal "c" in the next cycle.

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

  // Check: If signal "d" rises...   // If-Else Block:
  $rose(d) |=>  if (b)
    // If "b" is true, then "c" should be high continuously or intermittently
for 2 clock cycles, followed by a rising edge on "a"
    c[->2] ##1 a;
  else
    // If "b" is not true, then "a" should be high continuously or
intermittently for 2 clock cycles, followed by a rising edge on "c"
    a[->2] ##1 c;
endproperty
```

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

`$rose(d) |=>`: This checks that when the signal "d" rises, the condition following it should hold.

`if (b) c[->2] ##1 a;`: If the condition is true (i.e., if "b" is true), then "c" should be high continuously **or** intermittently **for** 2 clock cycles, followed by a rising **edge** on "a".

`else a[->2] ##1 c;`: If "b" is **not** true, then "a" should be high continuously **or** intermittently **for** 2 clock cycles, followed by a rising **edge** on "c".