IIT JODHPUR

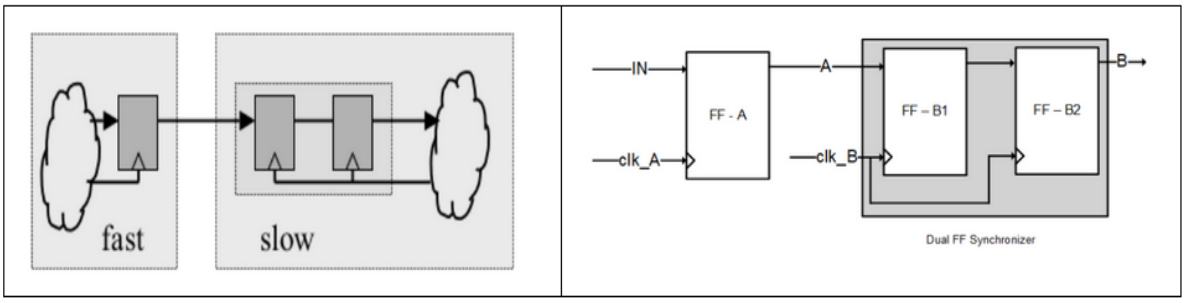
Major Examination: EEL7770 Formal Verification-Nov'25 (OPEN-BOOK)

Guidelines (Total time: 180 minutes, Maximum Marks: 40):

- Please read the question paper very carefully. **NO clarification is required in any question.** In case of any doubt, assume whatever you wish to and state that in your answer.
- Usage of assistance from internet/any LLM tool is NOT allowed.

---

1. Modern SoC designs utilize multiple clock domains (i.e., frequencies) to achieve higher performance and low power dissipation. However, timing problems may arise around the logic regions when there is crossing of clock from one frequency to another. This is referred to as CDC (clock domain crossing) in the industrial flows. We need to verify these crossings to ensure that we have data integrity (i.e., no data is lost). Left part of the following figure is a high level representation (so, no signal names are mentioned) of CDC showing transition from fast clock to slow clock as we move from left to right (i.e., input to output). Assume d_in (input data), d_out (output data) and other important signals as per your understanding/choice (here synchronization is achieved with the help of edge-triggered flip-flops and cloud-like structures represent the combinational logic). **How we can carry out this verification for this logic module?** [3]



   You can analyze the diagram shown in the right portion, which is also representing CDC (clk_A is the first clock where as clk_B is the second clock).

2. You have developed a synchronous hardware module in Verilog HDL as a finite-state machine (FSM) with three 4-bit registers: state_reg, count_reg, flag_reg. [2+2+2+2]

   - state_reg encodes one of 10 valid states (S0–S9)
   - count_reg is a 4-bit counter that increments until it reaches a programmable threshold, after which it resets to 0.
   - flag_reg is a 6-bit register where only the lower 3 bits are used, and upper 3 bits are "don't care" in functionality.

   a) Compute the total number of possible states in the state space through explicit state enumeration? Hint: Explicit enumeration means all bit-level combinations of the three registers.
   b) Compute the number of semantically valid states given the conditions in the above scenario (i.e., constraints on the bits of registers).
   c) What property you can write for verification of above logic module?
   d) Explain how state-space reduction techniques can bring down the number of states to be verified as you might have reported in (a) and (b) above?

3. Consider that a local deeptech startup named as *MewarNetworkDesign* develops custom network traffic controller that performs packet length accumulation and generates alerts when specific thresholds are reached. You have joined this company for an internship role in the design verification team. Note that the traffic amount is measured in the unit of packets. Devise a formal verification methodology for this design? [4]

```
module packet_counter(
   input  wire       clk,
   input  wire       rst,
   input  wire       pkt_valid,
   input  wire [7:0]  pkt_len,
   output reg  [15:0] total_len,
   output reg        warn,
   output reg        overflow
);
   always @(posedge clk) begin
      if (rst) begin
         total_len <= 0;
         warn      <= 0;
         overflow  <= 0;
      end else if (pkt_valid) begin
         total_len <= total_len + pkt_len;

         if (total_len + pkt_len > 16'h0FFF)
            warn <= 1;
         else
            warn <= 0;

         if (total_len + pkt_len > 16'hFF00)
            overflow <= 1;
      end
   end
endmodule
```

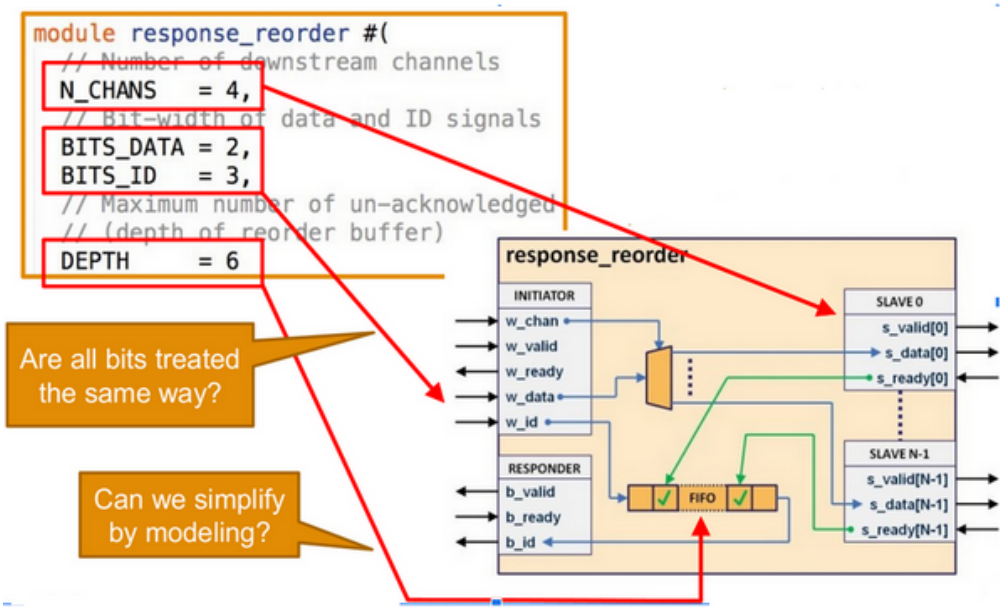
4. Consider the logic block shown below. What could be one way of formal representation of this design in formal semantics so that verification techniques like model checking could be utilized on this model. Please find out which order out of A→B→C→D or C→A→D→B is the most efficient for its canonical representation? [6]

```
module dut (
   input  logic A,
   input  logic B,
   input  logic C,
   input  logic D,
   output logic Y
);
   logic t1, t2, t3;
   assign t1 = (A & B);
   assign t2 = (~A & C);
   assign t3 = (B & D);
   assign Y  = (t1 | t2) ^ t3;
endmodule
```

Hint: ROBDD is one way of canonical representation of digital design blocks.

5. The figure below shows an abstracted response_reorder module that accepts responses from multiple downstream slave channels and reorders them before delivering them to the initiator. The module is parameterized by the following variables:

   - N_CHANS (number of downstream channels) = 4
   - BITS_DATA (bit-width of data signals) = 2
   - BITS_ID (bit-width of ID signals) = 3
   - DEPTH (depth of the internal FIFO used for reordering) = 6



a) Based on the abstraction shown, explain whether all bits of the response are treated uniformly by the reorder logic or whether some fields require special handling. [2]
b) With BITS_DATA and BITS_ID as unchanged, how does verification complexity change when N_CHANS increases from 4 to 16 ? Show with suitable example. [2]
c) After abstraction, how may we verify the logical functionality of the reordering block? [2]

6. RTL designs could be functionally equivalent but structurally different. Typically, RTL designs could be optimized through different pipelines, transforms, and decomposition. With any formal verification technique, check if the two logic blocks are equivalent or not? [6]

```
module block_A (
   input  logic       clk,
   input  logic [15:0] a,
   input  logic [15:0] b,
   input  logic [15:0] c,
   output logic [17:0] y
);


   logic [17:0] a_mul4_reg, b_shift_reg, c_reg;

   always_ff @(posedge clk) begin
      a_mul4_reg <= a * 4;      // multiply
      b_shift_reg <= b << 2;    // shift-left
      c_reg <= c;
   end

   // Final output combinational
   always_comb begin
      y = a_mul4_reg + b_shift_reg + c_reg;
   end

endmodule
```

```
module block_B (
   input  logic       clk,
   input  logic [15:0] a,
   input  logic [15:0] b,
   input  logic [15:0] c,
   output logic [17:0] y
);

   // Compute partial results
   logic [17:0] a_shl1, a_shl2;
   logic [17:0] b_shl2;

   assign a_shl1 = {a, 1'b0};       // a << 1
   assign a_shl2 = {a, 2'b00};      // a << 2
   assign b_shl2 = {b, 2'b00};      // b << 2

   logic [17:0] a_reg, b_reg;
   logic [15:0] c_stage1, c_stage2;

   always_ff @(posedge clk) begin
      a_reg <= a_shl2;
      b_reg <= b_shl2;
      c_stage1 <= c;
      c_stage2 <= c_stage1;
    end

     logic [17:0] sum1;
   assign sum1 = a_reg + b_reg;

   always_ff @(posedge clk) begin
           y <= sum1 + c_stage2;
   end
endmodule
```

Hint: The above code is written in System Verilog. logic data-type is similar to reg data-type.

always_ff and always_comb is similar to always procedural block in Verilog.

7. A round robin arbiter is a digital circuit module that grants access to a shared resource to multiple requesters in a fair, rotating sequence. It ensures that no requester is starved of access by cycling through them, giving each a fixed time slot in turn. This mechanism is commonly used in system-on-chips and routers where multiple components need to share a common resource like a memory bus. Therefore, verification of this arbiter module is centrally important.
   In the below figure, a RTL is shown (the RTL description spans from left to right side of the figure), **please develop a sound and complete formal verification scheme for this design? What abstraction technique you could possibly utilize for this design?** [5 + 2]

```
module rr_arbiter #(
   parameter GRANT_CYCLES = 4
)(
   input  logic       clk,
   input  logic       reset,
   input  logic       req0,
   input  logic       req1,
   output logic       gnt0,
   output logic       gnt1,
   output logic [2:0]  counter   );

   typedef enum logic [1:0] {IDLE, GNT0, GNT1} state_t;
   state_t state, next_state;

   logic [2:0] cycle_cnt;
   logic      last_winner;  // 0 = client0 was last, 1 = client1
was last

   // State transition logic
   always_comb begin
      next_state = state;
      case (state)

         IDLE: begin
            if (req0 & req1)
               next_state = (last_winner == 0) ? GNT1 :
GNT0;
            else if (req0)
               next_state = GNT0;
            else if (req1)
               next_state = GNT1;
         end
```

```
GNT0: if (cycle_cnt == GRANT_CYCLES-1)
next_state = IDLE;
         GNT1: if (cycle_cnt ==
GRANT_CYCLES-1) next_state = IDLE;

      endcase
   end
always_ff @(posedge clk or posedge reset) begin
      if (reset) begin
         state      <= IDLE;
         cycle_cnt   <= 0;
         last_winner <= 1'b0;
      end else begin
         state <= next_state;
         if (state == GNT0 || state == GNT1)
            cycle_cnt <= cycle_cnt + 1'b1;
         else
            cycle_cnt <= 0;

         if (state == GNT0 && next_state == IDLE)
last_winner <= 0;
         if (state == GNT1 && next_state == IDLE)
last_winner <= 1;
      end
   end

   assign gnt0 = (state == GNT0);
   assign gnt1 = (state == GNT1);

   assign counter = cycle_cnt;
endmodule
```

Hint: You may considering the logical functionality from RTL description for analysis.
