1. What are the differences between synchronous and asynchronous state machines? How do you choose one?
   Synchronous State Machines:
   - All state transitions occur at specific clock edges.
   - State updates are synchronized with a clock signal.
   - Typically simpler to design and analyze because timing is controlled.
   - Easier to implement in digital circuits.

   Asynchronous State Machines:

   - State transitions can occur at any time based on input changes.
   - Not synchronized to a clock signal, which can lead to timing issues (like race conditions).
   - More complex and harder to analyze, but can be faster for specific applications where immediate response is critical.

   Choosing Between Them:

   - Use synchronous for designs requiring reliability and ease of design (like most digital systems).
   - Use asynchronous for low-latency applications where speed is critical, and the design can handle the complexity.
2. Illustrate the differences between binary encoding and onehot encoding mechanism state machines.
   Binary Encoding:

   - Each state is represented by a unique binary number.
   - More states can be represented with fewer flip-flops (e.g., 4 flip-flops can represent 16 states).
   - Complexity increases with the number of states (requires combinational logic for state transitions).

   One-Hot Encoding:

   - Each state is represented by one flip-flop, where only one flip-flop is 'hot' (set to 1) at any time.
   - Simpler transition logic, as only one flip-flop needs to be toggled.
   - More flip-flops are needed for more states (e.g., 4 states require 4 flip-flops).
3. Illustrate how a multi-dimensional array is implemented in Verilog.

```
module multi_dim_array_example;
  // Declare a 2D array
  reg [7:0] my_array [0:3][0:3]; // 4x4 array of 8-bit registers

  initial begin
    // Assign values to the 2D array
    my_array[0][0] = 8'hAA;
    my_array[1][2] = 8'hBB;
    // Accessing a value
    $display("Value at (1, 2): %h", my_array[1][2]);
  end
endmodule
```

4. What are the considerations in instantiating technology-specific memories?
   - Memory Type: Understand the differences between SRAM, DRAM, ROM, etc., and their use cases.
   - Access Speed: Evaluate the read/write speeds required for your application.
   - Density and Size: Choose based on storage needs and area constraints.

- Power Consumption: Assess power requirements based on your design's needs.
- Technology Compatibility: Ensure compatibility with the existing technology and design flow.
- Initialization Requirements: Some memories need specific initialization procedures.

5. What are the factors that dictate the choice between synchronous and asynchronous memories?
   - Speed Requirements: Synchronous memories typically offer higher performance and are used in high-speed applications.
   - Design Complexity: Synchronous memories are easier to interface with a synchronous system.
   - Power Consumption: Asynchronous memories may have lower power consumption in low-speed applications.
   - Cost and Availability: Evaluate cost implications and availability of specific memory types.
6. Design 4-bit binary counter using Verilog.

```
module binary_counter (
   input wire clk,
   input wire rst,
   output reg [3:0] count
);
   always @(posedge clk or posedge rst) begin
      if (rst)
         count <= 4'b0000; // Reset the count to 0
      else
         count <= count + 1; // Increment the count
   end
endmodule
```

7. Given the state-assigned table shown below implement the finite-state machine rest state is 000.

Present state y[2:0]	Next state(x=0) Y[2:0] (x=1)		Output z
000	000	001	0
011	001	100	0
010	010	001	0
011	001	010	1
100	011	100	1

```
module fsm (
   input wire clk,
   input wire rst,
   input wire x,
   output reg [2:0] y,
   output reg z
);
   // State encoding
   typedef enum reg [2:0] {
      S0 = 3'b000,
      S1 = 3'b001,
      S2 = 3'b010,
      S3 = 3'b011,
```

```
        S4 = 3'b100
    } state_t;

    state_t current_state, next_state;

    always @(posedge clk or posedge rst) begin
        if (rst)
            current_state <= S0; // Reset state
        else
            current_state <= next_state; // Next state
    end

    always @(*) begin
        case (current_state)
            S0: begin
                z = 0;
                next_state = (x == 1) ? S1 : S0;
            end
            S1: begin
                z = 0;
                next_state = (x == 1) ? S3 : S2;
            end
            S2: begin
                z = 0;
                next_state = (x == 1) ? S1 : S2;
            end
            S3: begin
                z = 1;
                next_state = S4;
            end
            S4: begin
                z = 1;
                next_state = (x == 1) ? S3 : S0;
            end
            default: begin
                z = 0;
                next_state = S0;
            end
        endcase
    end

    // Output the current state
    always @(*) begin
        y = current_state;
    end
endmodule
```