Develop the algorithm for a Moore/Mealy Machine implementation of the state diagram from problem statement

Assign each location around the ISRO campus with a unique state in 3-bit binary representation:

Room0: 000
Room1: 001
Room2: 010
Room3: 011
Room4: 100
Room5: 101
Room6: 110
Room7: 111

Define the inputs: In this case, the input is a binary sequence representing the travel plans for Figo. Each '0' or '1' in the sequence corresponds to a move from the current location to the next location.

Implement the FSM based on the state diagram:

For a Moore machine:

Initialize the current state to Room0.

Define the output action associated with each state, which is the current location.

Loop through the binary sequence of travel plans.

For each input, transition to the next state based on the state diagram.

Output the current location/state associated with the current state.

For a Mealy machine:

Initialize the current state to Room0.

Define the output action associated with each transition, which is the next location.

Loop through the binary sequence of travel plans.

For each input, transition to the next state based on the state diagram and output the next location/state associated with the current transition.

Here's the algorithm in pseudocode for the Mealy machine implementation:

```
current_state = Room0
output = "Room0"

function land_rover_figo_fsm(binary_sequence):
    for input in binary_sequence:
        if input == "0":
            next_state = get_next_state(current_state, "0")
            output = get_output(current_state, "0")
        else if input == "1":
            next_state = get_next_state(current_state, "1")
            output = get_output(current_state, "1")

        current_state = next_state
        print(output)

function get_next_state(current_state, input):
    // Define the state transitions based on the state diagram
    transition_table = {
        "000": {"0": "001", "1": "000"},
        "001": {"0": "010", "1": "001"},
        "010": {"0": "011", "1": "010"},
        "011": {"0": "100", "1": "011"},
        "100": {"0": "101", "1": "100"},
        "101": {"0": "110", "1": "101"},
        "110": {"0": "111", "1": "110"},
        "111": {"0": "000", "1": "111"}
    }

    return transition_table[current_state][input]

function get_output(current_state, input):
    // Define the output associated with each transition
    output_table = {
        "000": {"0": "Room1", "1": "Room0"},
        "001": {"0": "Room2", "1": "Room1"},
        "010": {"0": "Room3", "1": "Room2"},
        "011": {"0": "Room4", "1": "Room3"},
        "100": {"0": "Room5", "1": "Room4"},
        "101": {"0": "Room6", "1": "Room5"},
        "110": {"0": "Room7", "1": "Room6"},
        "111": {"0": "Room0", "1": "Room7"}
    }

    return output_table[current_state][input]
```

Develop the Verilog code for the design

```
module land_rover_figo_fsm(
  input wire clk,        // Clock input
  input wire reset,      // Reset input
  input wire travel_plan,  // Binary sequence input
  output reg [2:0] location  // Current location output
);

// Define the states
parameter Room0 = 3'b000;
parameter Room1 = 3'b001;
parameter Room2 = 3'b010;
parameter Room3 = 3'b011;
parameter Room4 = 3'b100;
parameter Room5 = 3'b101;
parameter Room6 = 3'b110;
parameter Room7 = 3'b111;

// Define the state register
reg [2:0] current_state;
reg [2:0] next_state;

// Define the outputs
reg [2:0] output_location;

// State transition and output logic
always @(posedge clk or posedge reset) begin
```

```
  if (reset) begin
    current_state <= Room0;
    output_location <= Room0;
  end
  else begin
    current_state <= next_state;
    output_location <= current_state;
  end
end

// Next state and output logic
always @(current_state or travel_plan) begin
  case (current_state)
    Room0:
      if (travel_plan == 1'b0)
        next_state = Room1;
      else if (travel_plan == 1'b1)
        next_state = Room0;
    Room1:
      if (travel_plan == 1'b0)
        next_state = Room2;
      else if (travel_plan == 1'b1)
        next_state = Room1;
    Room2:
      if (travel_plan == 1'b0)
        next_state = Room3;
      else if (travel_plan == 1'b1)
        next_state = Room2;
    Room3:
      if (travel_plan == 1'b0)
        next_state = Room4;
      else if (travel_plan == 1'b1)
        next_state = Room3;
    Room4:
      if (travel_plan == 1'b0)
        next_state = Room5;
      else if (travel_plan == 1'b1)
        next_state = Room4;
    Room5:
      if (travel_plan == 1'b0)
        next_state = Room6;
      else if (travel_plan == 1'b1)
        next_state = Room5;
    Room6:
      if (travel_plan == 1'b0)
        next_state = Room7;
      else if (travel_plan == 1'b1)
        next_state = Room6;
    Room7:
      if (travel_plan == 1'b0)
        next_state = Room0;
      else if (travel_plan == 1'b1)
```

```
        next_state = Room7;
  endcase
end

endmodule
```

Verify the functionality with possible test cases

```
module land_rover_figo_fsm_tb;

  // Parameters
  parameter CLOCK_PERIOD = 10;  // Clock period in simulation time units

  // Inputs
  reg clk;
  reg reset;
  reg travel_plan;

  // Outputs
  wire [2:0] location;

  // Instantiate the Land Rover Figo FSM
  land_rover_figo_fsm uut(
    .clk(clk),
    .reset(reset),
    .travel_plan(travel_plan),
    .location(location)
  );

  // Clock generation
  always begin
    clk = 0;
    # (CLOCK_PERIOD / 2);
    clk = 1;
    # (CLOCK_PERIOD / 2);
  end

  // Test case 1
  initial begin
    reset = 1;
    travel_plan = 0;
    # (2 * CLOCK_PERIOD);
    reset = 0;
    travel_plan = 1;
    # (5 * CLOCK_PERIOD);
    $display("Test case 1: Location = %b", location);
    $finish;
  end
  // Test case 2
  initial begin
    reset = 1;
    travel_plan = 0;
    # (2 * CLOCK_PERIOD);
    reset = 0;
    travel_plan = 1;
    # (CLOCK_PERIOD);
    travel_plan = 0;
    # (2 * CLOCK_PERIOD);
    travel_plan = 1;
    # (3 * CLOCK_PERIOD);
    $display("Test case 2: Location = %b", location);
    $finish;
  End

endmodule
```

Draw suitable conclusion based on the results obtained.

The interpretation drawn from the test case findings will be influenced by the expected behavior of the Land Rover Figo FSM. The following conclusions are conceivable.

We may say that the Land Rover Figo FSM is working properly if the output locations seen in the test cases match the expected locations based on the supplied travel plans. This shows that the FSM provides the appropriate output locations and transitions between states accurately.

A potential problem with the Land Rover Figo FSM implementation can be seen if the output locations do not match the predicted locations. To find and fix the issue in this instance, more investigation and debugging are needed.

Full coverage testing: If the test cases cover a variety of trip scenarios, including various binary input sequences, and the FSM consistently generates the expected output locations, it increases confidence in the FSM's functionality. This indicates that the FSM is capable of managing a variety of travel arrangements as anticipated.

Partial coverage testing: It's necessary to take into account the coverage achieved if the test cases only cover a small number of trip arrangements. Even though the FSM may function properly in the tested settings, problems could still occur in untested ones. To guarantee the FSM is accurate in all instances, extra testing and analysis are advised in these circumstances..