TEST bench code for task 2

```
// tb_ram_sync.v
// Testbench for Synchronous RAM Module

`timescale 1ns / 1ps // Define time units for simulation

module tb_ram_sync;

  // Parameters for the RAM module (must match the DUT)
  parameter DATA_WIDTH = 8;
  parameter ADDR_WIDTH = 4;

  // Testbench signals
  reg              clk;
  reg              rst_n;
  reg              we;
  reg  [ADDR_WIDTH-1:0] addr;
  reg  [DATA_WIDTH-1:0] data_in;
  wire [DATA_WIDTH-1:0] data_out;

  // Instantiate the Device Under Test (DUT) - our RAM module
  ram_sync #(
    .DATA_WIDTH(DATA_WIDTH),
    .ADDR_WIDTH(ADDR_WIDTH)
  ) dut (
    .clk(clk),
    .rst_n(rst_n),
    .we(we),
    .addr(addr),
    .data_in(data_in),
    .data_out(data_out)
  );

  // Clock generation
  parameter CLK_PERIOD = 10; // 10ns period, 5ns high, 5ns low
  initial begin
    clk = 0;
    forever #(CLK_PERIOD/2) clk = ~clk; // Toggle clock every half period
  end

  // VCD Dump setup
```

```
initial begin
    $dumpfile("dump.vcd"); // Specify the VCD file name
    $dumpvars(0, tb_ram_sync); // Dump all variables in the current scope (tb_ram_sync)
                               // The '0' indicates dumping all levels of hierarchy
end

// Test sequence
initial begin
    // Initialize inputs
    rst_n   = 0;   // Assert reset
    we      = 0;
    addr    = 0;
    data_in = 0;

    $display("Time | Action      | Addr | Data_In | Data_Out | Expected_Out");
    $display("-----------------------------------------------------------------");

    // Release reset after a few clock cycles
    #(CLK_PERIOD * 2) rst_n = 1; // De-assert reset

    // --- Test Case 1: Write and Read at Address 0 ---
    #(CLK_PERIOD); // Wait for a clock cycle after reset de-assertion
    $display("%0t | Reset De-asserted", $time);

    // Write to address 0
    we      = 1;
    addr    = 4'h0;
    data_in = 8'hAA;
    #(CLK_PERIOD);
    $display("%0t | Write       | %h    | %h     | %h      | --", $time, addr, data_in, data_out);

    // Read from address 0 (data_out will show previous value due to synchronous read)
    we      = 0; // Disable write
    addr    = 4'h0;
    #(CLK_PERIOD); // Wait for one clock cycle for data to propagate to data_out
    $display("%0t | Read        | %h    | %h     | %h      | %h", $time, addr, data_in, data_out, 8'hAA);
    if (data_out !== 8'hAA) $error("ERROR: Data mismatch at address 0. Expected AA, Got %h", data_out);

    // --- Test Case 2: Write to Address 5, then Read ---
    we      = 1;
    addr    = 4'h5;
```

```
data_in = 8'h55;
#(CLK_PERIOD);
$display("%0t | Write      | %h    | %h     | %h      | --", $time, addr, data_in, data_out);

we     = 0; // Disable write
addr   = 4'h5;
#(CLK_PERIOD); // Wait for one clock cycle for data to propagate to data_out
$display("%0t | Read       | %h    | %h     | %h      | %h", $time, addr, data_in, data_out, 8'h55);
if (data_out !== 8'h55) $error("ERROR: Data mismatch at address 5. Expected 55, Got %h", data_out);

// --- Test Case 3: Write to Address F, then Read ---
we     = 1;
addr   = 4'hF;
data_in = 8'hFF;
#(CLK_PERIOD);
$display("%0t | Write      | %h    | %h     | %h      | --", $time, addr, data_in, data_out);

we     = 0; // Disable write
addr   = 4'hF;
#(CLK_PERIOD); // Wait for one clock cycle for data to propagate to data_out
$display("%0t | Read       | %h    | %h     | %h      | %h", $time, addr, data_in, data_out, 8'hFF);
if (data_out !== 8'hFF) $error("ERROR: Data mismatch at address F. Expected FF, Got %h", data_out);

// --- Test Case 4: Read from an unwritten address (should be 0) ---
addr   = 4'h2; // Address 2 was not written
#(CLK_PERIOD); // Wait for one clock cycle for data to propagate to data_out
$display("%0t | Read (unwritten) | %h    | %h     | %h      | %h", $time, addr, data_in, data_out, 8'h00);
if (data_out !== 8'h00) $error("ERROR: Data mismatch at unwritten address 2. Expected 00, Got %h", data_out);

// --- Test Case 5: Overwrite Address 0 and Read ---
we     = 1;
addr   = 4'h0;
data_in = 8'h12;
#(CLK_PERIOD);
$display("%0t | Overwrite   | %h    | %h     | %h      | --", $time, addr, data_in, data_out);

we     = 0; // Disable write
addr   = 4'h0;
#(CLK_PERIOD); // Wait for one clock cycle for data to propagate to data_out
$display("%0t | Read       | %h    | %h     | %h      | %h", $time, addr, data_in, data_out, 8'h12);
if (data_out !== 8'h12) $error("ERROR: Data mismatch at address 0 after overwrite. Expected 12, Got %h", data_out);
```

```
    // End simulation
    #(CLK_PERIOD);
    $display("----------------------------------------------------------------");
    $display("Simulation Finished at %0t", $time);
    $finish; // Terminate simulation
  end

endmodule
```

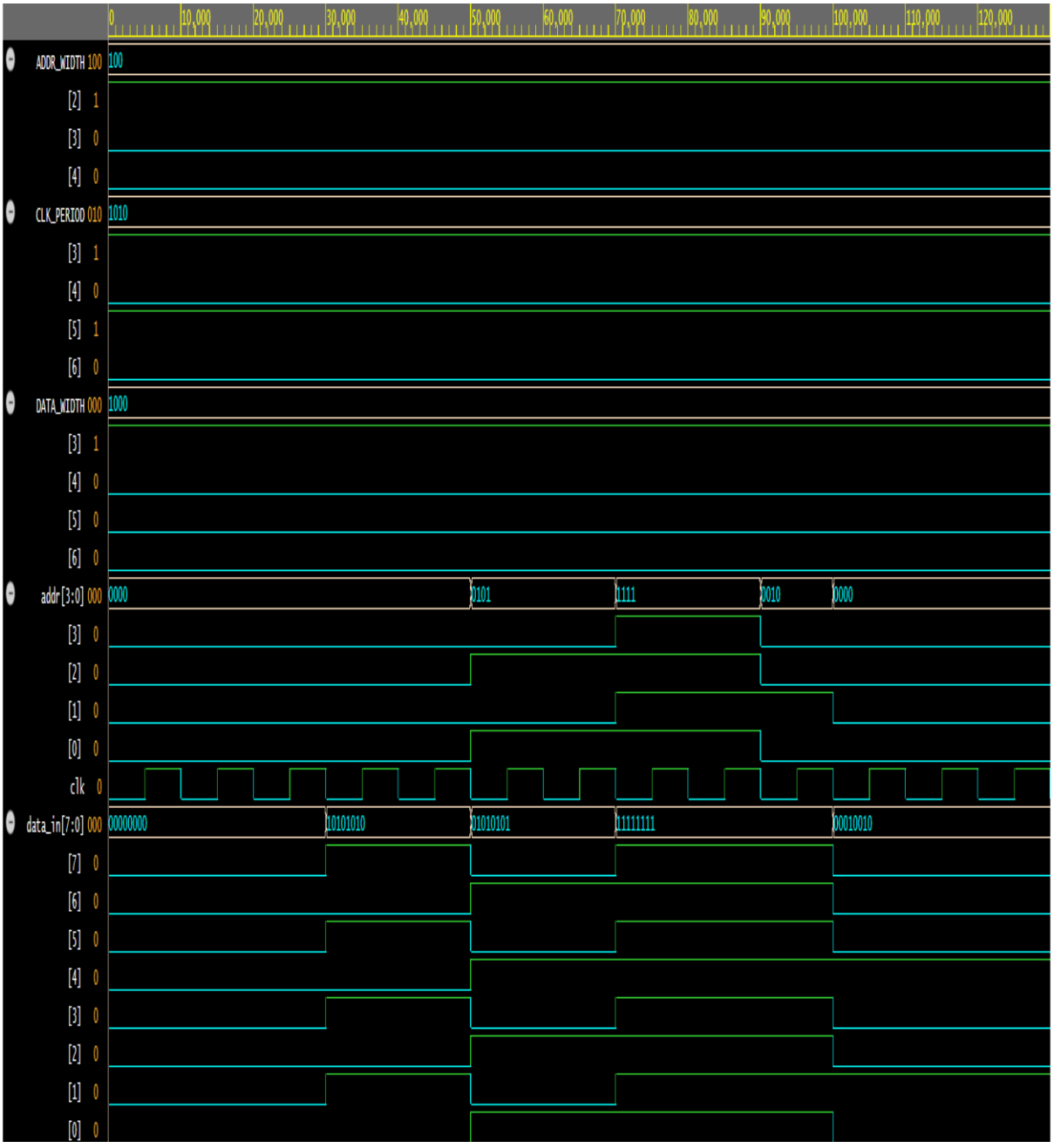
DESIGN CODE FOR TASK 2

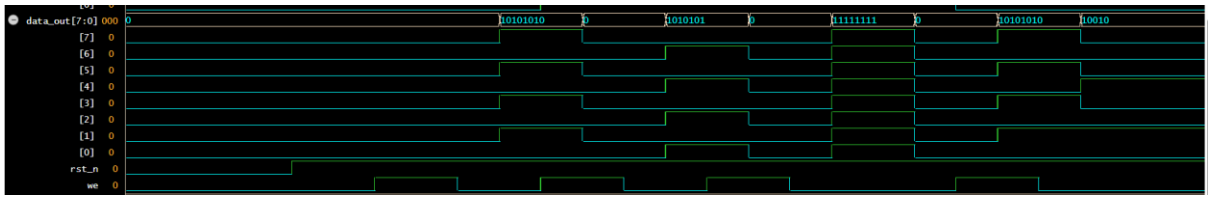
```
// ram_sync.v
// Synchronous RAM Module with Read and Write Operations
// Designed for synthesizability with EDA tools.

`timescale 1ns / 1ps // Added timescale for consistency and to remove warning

module ram_sync #(
  parameter DATA_WIDTH = 8,  // Data bus width
  parameter ADDR_WIDTH = 4   // Address bus width (e.g., 4 bits for 16 locations)
) (
  input  wire                 clk,       // Clock input (essential for synchronous logic)
  input  wire                 rst_n,     // Asynchronous active-low reset
  input  wire                 we,        // Write Enable (active high) - controls write operation
  input  wire [ADDR_WIDTH-1:0] addr,      // Address input for both read and write
  input  wire [DATA_WIDTH-1:0] data_in,   // Data input for write operation
  output reg  [DATA_WIDTH-1:0] data_out   // Data output for read operation (registered for synchronous read)
);

  // Declare the memory array
  // This 'reg' array is the standard way to infer Block RAM or Distributed RAM
  // depending on the target technology and memory size.
  reg [DATA_WIDTH-1:0] mem [0:(1<<ADDR_WIDTH)-1];

  // Synchronous Write Operation
  // This 'always' block describes the write behavior.
  // The sensitivity list includes 'posedge clk' for synchronous operation
  // and 'negedge rst_n' for asynchronous reset.
```

```
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        // Asynchronous Reset:
        // For synthesis, this loop will typically translate to logic that clears
        // the memory contents on reset. For large memories, EDA tools might
        // optimize this or expect initial values to be loaded via a separate
        // memory initialization file (e.g., .mem, .hex) during bitstream generation.
        for (int i = 0; i < (1<<ADDR_WIDTH); i = i + 1) begin
            mem[i] <= {DATA_WIDTH{1'b0}}; // Initialize all memory locations to 0
        end
    end else if (we) begin
        // Synchronous Write:
        // When 'we' is high, 'data_in' is written to 'mem[addr]' on the
        // positive edge of the clock. This infers a write port.
        mem[addr] <= data_in;
    end
end

// Synchronous Read Operation
// This 'always' block describes the read behavior.
// The output 'data_out' is registered, meaning its value changes only on the
// positive edge of the clock. This is crucial for synchronous RAMs to ensure
// stable output and meet timing requirements in hardware.
// The output 'data_out' will reflect the data at 'addr' from the *previous*
// clock cycle (after the write has propagated internally).
always @(posedge clk or negedge rst_n) begin // Corrected 'cllk' to 'clk' here
    if (!rst_n) begin
        data_out <= {DATA_WIDTH{1'b0}}; // Reset data_out to 0
    end else begin
        data_out <= mem[addr]; // Read data from memory at the current address
    end
end

// The 'initial' block and '$display' statements are for simulation only.
// EDA synthesis tools will ignore these constructs.
initial begin
    $display("----------------------------------------");
    $display("Synchronous RAM Module Initialized");
    $display("Data Width: %0d bits", DATA_WIDTH);
    $display("Address Width: %0d bits", ADDR_WIDTH);
    $display("Memory Depth: %0d locations", (1<<ADDR_WIDTH));
```

```
        $display("----------------------------------------");
    end

endmodule
```

simulation photos




data_out[7:0]
[7]
[6]
[5]
[4]
[3]
[2]
[1]
[0]
rst_n
we
10101010
1010101
11111111
10101010
10010