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



