Digital System Design using HDL Lab Report

Experiment 8b

Instruction Memory and Data Memory

Submitted by

Madhu Krishnan A P

M.Tech VLSI and Embedded Systems

Department of Electronics

Cochin University of Science and Technology

Contents

1 Instruction Memory

1.1 Description

The instruction memory unit takes in an instruction address and reads 32-bit data from the given address.

Inputs:

- **Instruction Address (A):** A 32-bit address input that specifies the location in memory from which data needs to be read.
- **Width:** 32 bits
- **Description:** The address points to the location where the data is stored in memory.

Outputs:

- **Read Data (Data_out):** A 32-bit data output that carries the data read from the specified memory address.
- **Width:** 32 bits
- **Description:** This output holds the 32-bit data fetched from the memory location specified by the input address.

Operation:

- The module continuously monitors the address input (A) and reads the 32-bit data from the memory location corresponding to that address.
- Upon receiving the address, the module outputs the 32-bit data at Data_out.

1.2 Block diagram

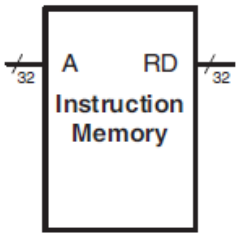

Figure 1: Instruction memory: Block diagram

1.3 Verilog description

```
module instru_file(
    input logic [31:0] A,          // 32-bit instruction address input
    output logic [31:0] read_data, // 32-bit read data output
    input logic clk,               // Clock signal
    input logic reset              // Reset signal
);

    // Memory declaration: 256 32-bit memory locations
    logic [31:0] memory [0:255];
```

```
    // Initialize memory with some values
    initial begin
        integer i;
        for (i = 0; i < 256; i = i + 1) begin
            memory[i] = 32'h0;  // Initialize all memory locations to 0
        end
        memory[0] = 32'hDEADBEEF; // Example initialization
        memory[1] = 32'hCAFEBABE;
        memory[2] = 32'hDEADBEEF;
        memory[3] = 32'h0000BEEF;
        memory[255] = 32'hBEEFBEEF;
    end

    // Read data logic
    always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
            read_data <= 32'h0; // Reset read data to 0
        end else begin
            read_data <= memory[A[31:2]]; // Read data from memory
                (word-aligned address)
        end
    end

endmodule
```

1.4 Testbench

```
module tb_instru_file;

    // Testbench signals
    logic [31:0] A;         // 32-bit instruction address input
    logic [31:0] read_data; // 32-bit read data output
    logic clk;              // Clock signal
    logic reset;            // Reset signal

    // Instantiate the unit under test (UUT)
    instru_file uut (
        .A(A),
        .read_data(read_data),
        .clk(clk),
        .reset(reset)
    );

    // Clock generation
    always begin
        #5 clk = ~clk; // Toggle clock every 5 time units (100 MHz
            clock)
    end

    // Stimulus process
    initial begin
        // Initialize signals
        clk = 0;
        reset = 0;
        A = 32'h0;
```

```
        // Apply reset
        reset = 1;
        #10 reset = 0; // Deassert reset after 10 time units

        // Test 1: Access memory at address 0
        A = 32'h0;
        #10;

        // Test 2: Access memory at address 4 (aligned address, i.e.,
            A[31:2] = 0)
        A = 32'h4;
        #10;

        // Test 3: Access memory at address 8 (another aligned address)
        A = 32'h8;
        #10;

        // Test 4: Access memory at address 255 (last location)
        A = 32'h03FC; // 32-bit address for memory[255]
        #10;

        // Test 5: Access an invalid address (out of bounds)
        A = 32'h1000; // Invalid address (out of range)
        #10;

        // End simulation
        $finish;
    end

endmodule
```

1.5 Simulation results

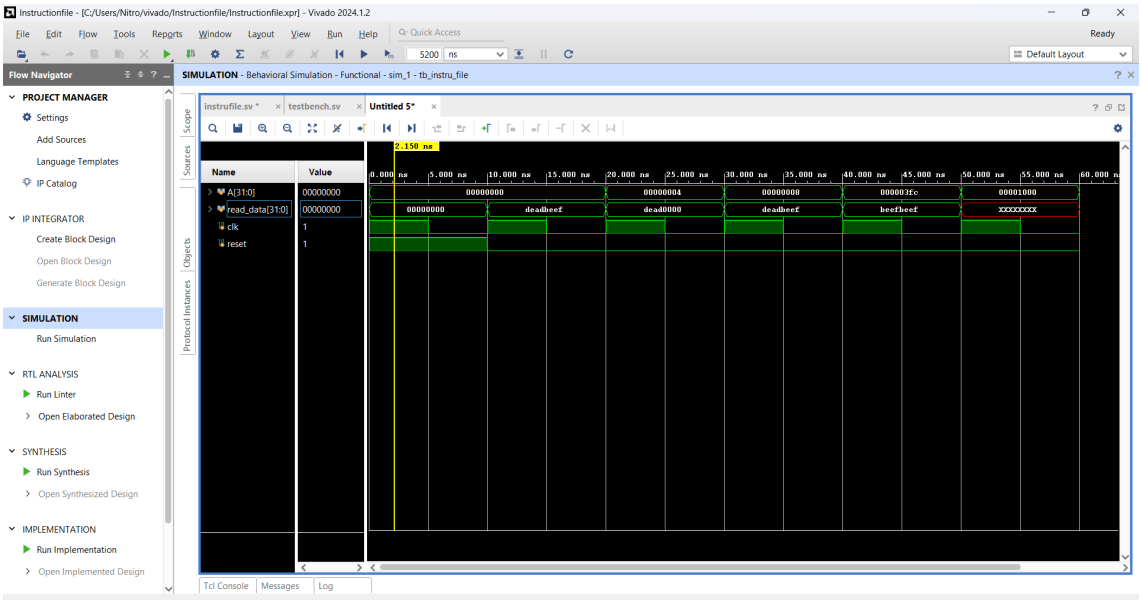

Figure 2: Instruction memory: Testbench results

2 Data memory

The data memory is a synchronous memory module with the following key features:

2.1 Single Read/Write Port

The memory module has a single port that can be used for either reading or writing data, depending on the state of the control signals.

2.2 Write Operation

- When the **Write Enable (WE)** signal is asserted (set to 1), the memory writes the value from the **Write Data (WD)** input into the memory location specified by the address input **A**.
- This write operation occurs on the **rising edge of the clock**, ensuring that the data is stored synchronously.

2.3 Read Operation

- When the **Write Enable (WE)** signal is deasserted (set to 0), the memory performs a read operation.
- The data stored at the address specified by **A** is output onto the **Read Data (RD)** bus.
- This happens without requiring a clock edge, indicating that the read operation is asynchronous.

2.4 Summary of Behavior

- **WE = 1**: Write **WD** to memory at address **A** on the rising edge of the clock.
- **WE = 0**: Output data from memory at address **A** onto the **RD** bus immediately.

It allows efficient use of a single port for both read and write operations.

2.5 Block diagram

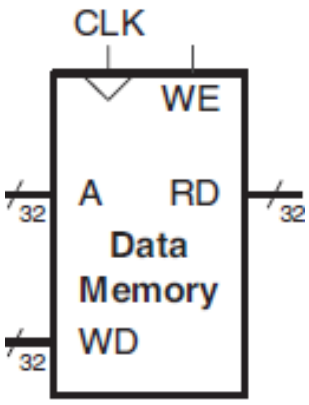

Figure 3: Data memory: Block diagram

2.6 Verilog description

```
module DataMemory #(
    parameter ADDR_WIDTH = 8,  // Address width
    parameter DATA_WIDTH = 16 // Data width
) (
    input  logic                  clk,        // Clock signal
    input  logic                  WE,         // Write Enable
    input  logic [ADDR_WIDTH-1:0] A,          // Address bus
    input  logic [DATA_WIDTH-1:0] WD,         // Write Data bus
    output logic [DATA_WIDTH-1:0] RD          // Read Data bus
);
    // Define the memory array
    logic [DATA_WIDTH-1:0] mem [0:(1 << ADDR_WIDTH)-1]; // Memory array

    // Write operation: Synchronous
    always_ff @(posedge clk) begin
        if (WE) begin
            mem[A] <= WD;  // Write data to address A
        end
    end

    // Read operation: Asynchronous
    always_comb begin
        if (!WE) begin
            RD = mem[A];   // Read data from address A
        end

        else begin
            RD = 'z;       // High impedance when writing
        end
    end
endmodule
```

2.7 Testbench

```
module tb_DataMemory;

    // Parameters
    parameter ADDR_WIDTH = 8;   // Address width
    parameter DATA_WIDTH = 16;  // Data width

    // Signals
    logic clk;                              // Clock
    logic WE;                               // Write Enable
    logic [ADDR_WIDTH-1:0] A;               // Address bus
    logic [DATA_WIDTH-1:0] WD;              // Write Data bus
    logic [DATA_WIDTH-1:0] RD;              // Read Data bus

    // DUT Instance
    DataMemory #(.ADDR_WIDTH(ADDR_WIDTH), .DATA_WIDTH(DATA_WIDTH)) dut
        (
        .clk(clk),
        .WE(WE),
        .A(A),
```

```
    .WD(WD),
    .RD(RD)
);

// Clock Generation
initial clk = 0;
always #5 clk = ~clk; // 10 ns clock period

// Test Sequence
initial begin
    // Initialize inputs
    WE = 0;
    A  = 0;
    WD = 0;

    // Apply reset values
    #10;

    // Test 1: Write and read from address 0
    write_memory(8'd0, 16'hAAAA);
    read_memory(8'd0);
    assert(RD == 16'hAAAA) else $fatal("Error: Address 0 read
        mismatch!");

    // Test 2: Write and read from address 1
    write_memory(8'd1, 16'hBBBB);
    read_memory(8'd1);
    assert(RD == 16'hBBBB) else $fatal("Error: Address 1 read
        mismatch!");

    // Test 3: Write and read from a different address
    write_memory(8'd2, 16'hCCCC);
    read_memory(8'd2);
    assert(RD == 16'hCCCC) else $fatal("Error: Address 2 read
        mismatch!");

    // Test completed
    #10;
    $finish; // End simulation
end

// Task: Write data to memory
task write_memory(input logic [ADDR_WIDTH-1:0] address, input
    logic [DATA_WIDTH-1:0] data);
    @(posedge clk);
    WE = 1;
    A  = address;
    WD = data;
    @(posedge clk);
    WE = 0;
endtask

// Task: Read data from memory
task read_memory(input logic [ADDR_WIDTH-1:0] address);
    @(posedge clk);
    WE = 0;
```

```
        A   = address;
        @(posedge clk);
    endtask

endmodule
```

The tb_DataMemory testbench is designed to verify the functionality of the DataMemory module, which supports synchronous writes and asynchronous reads. It uses a clock signal (clk) for synchronization and drives the control signals (WE, A, WD) to simulate memory operations. The testbench initializes all inputs, performs multiple writes and read operations at different memory addresses, and checks the output (RD) against expected values using assertions to ensure correct behaviour. If any assertion fails, the simulation terminates with an error message.

Reusable tasks (write_memory and read_memory) simplify the test logic by encapsulating the operations for writing data to a specific address and reading data from the memory. The clock toggles every 10 time units to synchronize write operations, while read operations remain asynchronous. The testbench concludes by verifying the integrity of the memory module through sequential tests, ensuring its reliability for real-world applications.

2.8 Simulation results

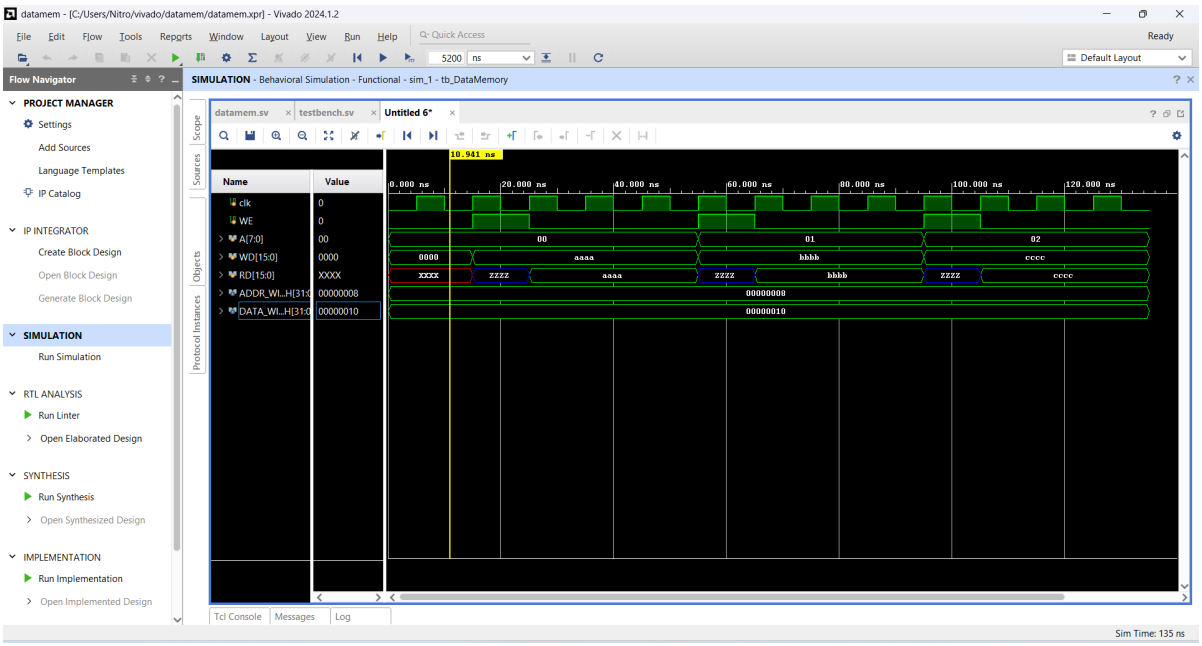

Figure 4: Data memory: Simulation result