Họ tên: Dương Anh Khôi

MSSV: 22520696

Đề: Thiết kế khối FIFO và LIFO, sau đó viết testbench đầy đủ.

Bài làm

1. FIFO

Source Code:

```
module FIFO (
    input wire clk,          // Clock signal
    input wire rst,          // Reset signal
    input wire w_en,         // Write enable
    input wire r_en,         // Read enable
    input wire [31:0] data_in,  // Data input
    output reg [31:0] data_out, // Data output
    output wire full,        // FIFO full indicator
    output wire empty        // FIFO empty indicator
);

// Register file declaration
reg [31:0] reg_file [31:0];  // 32x32 memory array

// Pointer declarations
reg [4:0] wr_ptr;            // Write pointer (5-bit for 0-31)
reg [4:0] r_ptr;             // Read pointer (5-bit for 0-31)

// Status flags assignments
assign full = (wr_ptr == 31); // Full when write pointer reaches 31
assign empty = (wr_ptr == r_ptr); // Empty when pointers are equal
```

```
// FIFO control logic
always @(posedge clk) begin
    if (rst) begin          // Synchronous reset
        wr_ptr <= 5'b0;
        r_ptr <= 5'b0;
        data_out <= 32'bz;   // High-impedance when reset
    end
    else begin
        // Write operation
        if (w_en && !full) begin
            reg_file[wr_ptr] <= data_in;
            wr_ptr <= wr_ptr + 1;
        end

        // Read operation
        if (r_en && !empty) begin
            data_out <= reg_file[r_ptr];  // Output data
            r_ptr <= r_ptr + 1;          // Increment read pointer
        end
    end
end

endmodule
```

Testbench:

```
`timescale 1ns/100ps

module TB;
    // Parameters
    parameter CLK_PERIOD = 10; // 10ns clock period (100MHz)

    // Signals for DUT
    reg clk;
    reg rst;
    reg w_en;
    reg r_en;
    reg [31:0] data_in;
    wire [31:0] data_out;
    wire full;
    wire empty;

    // Instantiate the DUT (Design Under Test)
    FIFO dut (
        .clk(clk),
        .rst(rst),
        .w_en(w_en),
        .r_en(r_en),
        .data_in(data_in),
        .data_out(data_out),
        .full(full),
        .empty(empty)
    );
```

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

// Test vector counter and error tracking
integer test_count = 0;
integer error_count = 0;

// Task for driving signals and checking results
task check_fifo;
    input [31:0] write_data;
    input write_en;
    input read_en;
    input expected_full;
    input expected_empty;
    input [31:0] expected_data;
    input check_data;
    begin
        test_count = test_count + 1;

        // Drive inputs
        @(posedge clk);
```

```
        #1; // Small delay after clock edge
        w_en = write_en;
        r_en = read_en;
        data_in = write_data;

        // Wait for next clock and check outputs
        @(posedge clk);
        #1; // Small delay for outputs to stabilize

        // Check flags
        if (full !== expected_full) begin
            $display("ERROR at test %0d: full = %b, expected = %b", test_count, full,
expected_full);
            error_count = error_count + 1;
        end

        if (empty !== expected_empty) begin
            $display("ERROR at test %0d: empty = %b, expected = %b", test_count,
empty, expected_empty);
            error_count = error_count + 1;
        end

        // Check data output if needed
        if (check_data == 1 && data_out !== expected_data) begin
            $display("ERROR at test %0d: data_out = %h, expected = %h", test_count,
data_out, expected_data);
            error_count = error_count + 1;
        end
```

```
        // Debug info
        $display("Test %0d: w_en=%b, r_en=%b, data_in=%h, data_out=%h, full=%b, empty=%b",
                 test_count, write_en, read_en, write_data, data_out, full, empty);
    end
  endtask

  // Array to store test data
  reg [31:0] test_data [0:31];
  integer i;

  // Main test sequence
  initial begin
    // Initialize test data array with unique values
    for (i = 0; i < 32; i = i + 1) begin
      test_data[i] = 32'hA0000000 + i;
    end

    // Initialize signals
    rst = 0;
    w_en = 0;
    r_en = 0;
    data_in = 0;

    // Apply reset
    #(CLK_PERIOD*2);
```

```
rst = 1;
#(CLK_PERIOD*2);
rst = 0;
#(CLK_PERIOD);

// Test 1: Check initial conditions after reset
check_fifo(32'h0, 0, 0, 0, 1, 32'hz, 0);

// Test 2: Write single data
check_fifo(test_data[0], 1, 0, 0, 0, 32'hz, 0);

// Test 3: Read single data
check_fifo(32'h0, 0, 1, 0, 1, test_data[0], 1);

// Test 4: Write multiple data
for (i = 0; i < 5; i = i + 1) begin
   check_fifo(test_data[i], 1, 0, 0, 0, 32'hz, 0);
end

// Test 5: Simultaneous read and write
for (i = 0; i < 5; i = i + 1) begin
   check_fifo(test_data[i+10], 1, 1, 0, 0, test_data[i], 1);
end

// Test 6: Fill the FIFO to check full flag
rst = 1; #(CLK_PERIOD*2); rst = 0; #(CLK_PERIOD); // Reset FIFO
```

```
// Fill to almost full
for (i = 0; i < 31; i = i + 1) begin
    check_fifo(test_data[i], 1, 0, 0, 0, 32'hz, 0);
end

// Check full condition
check_fifo(test_data[31], 1, 0, 1, 0, 32'hz, 0);

// Test 7: Empty the FIFO to check empty flag
for (i = 0; i < 32; i = i + 1) begin
    check_fifo(32'h0, 0, 1, 0, (i==31), test_data[i], 1);
end

// Test 8: Try to read from empty FIFO
check_fifo(32'h0, 0, 1, 0, 1, 32'hz, 0);

// Test 9: Try to write to full FIFO
rst = 1; #(CLK_PERIOD*2); rst = 0; #(CLK_PERIOD); // Reset FIFO

// Fill FIFO completely
for (i = 0; i < 32; i = i + 1) begin
    check_fifo(test_data[i], 1, 0, (i==31), 0, 32'hz, 0);
end

// Try to write one more (should not change FIFO state)
check_fifo(32'hFFFFFFFF, 1, 0, 1, 0, 32'hz, 0);
```

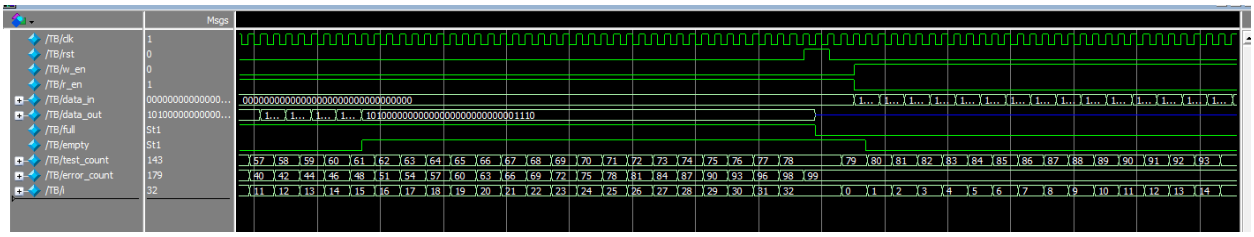
```
    // Test 10: Read and verify all data from full FIFO
    for (i = 0; i < 32; i = i + 1) begin
        check_fifo(32'h0, 0, 1, 0, (i==31), test_data[i], 1);
    end

    // Display test results
    if (error_count == 0)
        $display("All %0d tests PASSED!", test_count);
    else
        $display("%0d out of %0d tests FAILED!", error_count, test_count);

    #(CLK_PERIOD*5);
    $stop ;
  end

endmodule
```

Mô phỏng:



2. LIFO

Source Code:

```
module LIFO (
    input wire clk,          // Clock signal
    input wire rst,          // Reset signal
    input wire w_en,         // Write (push) enable
    input wire r_en,         // Read (pop) enable
    input wire [31:0] data_in,  // Data input
    output reg [31:0] data_out, // Data output
    output wire full,        // Stack full indicator
    output wire empty        // Stack empty indicator
);

// Memory array declaration
reg [31:0] stack_mem [31:0]; // 32x32 memory array (depth=32, width=32)

// Stack pointer declaration (6-bit to handle 0-32 states)
reg [5:0] sp;                // Points to next available space

// Status flags assignments
assign full = (sp == 6'd32); // Full when pointer reaches 32
assign empty = (sp == 6'd0); // Empty when pointer is 0

// Stack control logic
always @(posedge clk) begin
    if (rst) begin           // Synchronous reset
```

```
        sp <= 6'd0;
        data_out <= 32'bz;   // High-impedance when reset
    end
    else begin
        // Push operation (highest priority)
        if (w_en && !full) begin
            stack_mem[sp] <= data_in;  // Store data at current pointer
            sp <= sp + 1;              // Increment stack pointer
        end

        // Pop operation (only if not pushing)
        else if (r_en && !empty) begin
            sp <= sp - 1;              // Decrement pointer first
            data_out <= stack_mem[sp - 1]; // Read previous top element
        end

        // Maintain output when no operations
        else begin
            data_out <= data_out;     // Keep previous value
        end
    end
end

endmodule
```

Testbench:

```
`timescale 1ns/100ps


module TB;
    // Parameters
    parameter CLK_PERIOD = 10; // 10ns clock period (100MHz)

    // Signals for DUT
    reg clk;
    reg rst;
    reg w_en;
    reg r_en;
    reg [31:0] data_in;
    wire [31:0] data_out;
    wire full;
    wire empty;

    // Instantiate the DUT (Design Under Test)
    LIFO dut (
        .clk(clk),
        .rst(rst),
        .w_en(w_en),
        .r_en(r_en),
        .data_in(data_in),
        .data_out(data_out),
        .full(full),
        .empty(empty)
```

```
);

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

// Test vector counter and error tracking
integer test_count = 0;
integer error_count = 0;

// Task for driving signals and checking results
task check_lifo;
    input [31:0] write_data;
    input write_en;
    input read_en;
    input expected_full;
    input expected_empty;
    input [31:0] expected_data;
    input check_data;
    begin
        test_count = test_count + 1;

        // Drive inputs
```

```
        @(posedge clk);
        #1; // Small delay after clock edge
        w_en = write_en;
        r_en = read_en;
        data_in = write_data;

        // Wait for next clock and check outputs
        @(posedge clk);
        #1; // Small delay for outputs to stabilize

        // Check flags
        if (full !== expected_full) begin
            $display("ERROR at test %0d: full = %b, expected = %b", test_count, full,
expected_full);
            error_count = error_count + 1;
        end

        if (empty !== expected_empty) begin
            $display("ERROR at test %0d: empty = %b, expected = %b", test_count,
empty, expected_empty);
            error_count = error_count + 1;
        end

        // Check data output if needed
        if (check_data == 1 && data_out !== expected_data) begin
            $display("ERROR at test %0d: data_out = %h, expected = %h", test_count,
data_out, expected_data);
            error_count = error_count + 1;
```

```
        end

        // Debug info
        $display("Test %0d: w_en=%b, r_en=%b, data_in=%h, data_out=%h, full=%b, empty=%b",
                 test_count, write_en, read_en, write_data, data_out, full, empty);
    end
  endtask

  // Array to store test data
  reg [31:0] test_data [0:31];
  integer i;

  // Main test sequence
  initial begin
    // Initialize test data array with unique values
    for (i = 0; i < 32; i = i + 1) begin
      test_data[i] = 32'hB0000000 + i;
    end

    // Initialize signals
    rst = 0;
    w_en = 0;
    r_en = 0;
    data_in = 0;

    // Apply reset
```

```
    #(CLK_PERIOD*2);
    rst = 1;
    #(CLK_PERIOD*2);
    rst = 0;
    #(CLK_PERIOD);

    // Test 1: Check initial conditions after reset
    check_lifo(32'h0, 0, 0, 0, 1, 32'hz, 0);

    // Test 2: Push single data
    check_lifo(test_data[0], 1, 0, 0, 0, 32'hz, 0);

    // Test 3: Pop single data (should get the same value back)
    check_lifo(32'h0, 0, 1, 0, 1, test_data[0], 1);

    // Test 4: Push multiple data in sequence
    for (i = 0; i < 5; i = i + 1) begin
        check_lifo(test_data[i], 1, 0, 0, 0, 32'hz, 0);
    end

    // Test 5: Pop multiple data in reverse order (LIFO behavior)
    for (i = 4; i >= 0; i = i - 1) begin
        check_lifo(32'h0, 0, 1, 0, (i==0), test_data[i], 1);
    end

    // Test 6: Fill the stack to check full flag
    for (i = 0; i < 32; i = i + 1) begin
```

```
    check_lifo(test_data[i], 1, 0, (i==31), 0, 32'hz, 0);
end

// Test 7: Try to push when full (should not change stack state)
check_lifo(32'hFFFFFFFF, 1, 0, 1, 0, 32'hz, 0);

// Test 8: Pop all data in reverse order and verify
for (i = 31; i >= 0; i = i - 1) begin
    check_lifo(32'h0, 0, 1, 0, (i==0), test_data[i], 1);
end

// Test 9: Try to pop from empty stack
check_lifo(32'h0, 0, 1, 0, 1, 32'hz, 0);

// Test 10: Test push priority over pop (push and pop simultaneously)
// Push some data first
for (i = 0; i < 3; i = i + 1) begin
    check_lifo(test_data[i], 1, 0, 0, 0, 32'hz, 0);
end

// Now try simultaneous push and pop (push should win)
check_lifo(test_data[10], 1, 1, 0, 0, 32'hz, 0);

// Verify the push happened by popping values
check_lifo(32'h0, 0, 1, 0, 0, test_data[10], 1); // First pop gets recently pushed value
for (i = 2; i >= 0; i = i - 1) begin
    check_lifo(32'h0, 0, 1, 0, (i==0), test_data[i], 1);
```

```
    end

    // Display test results
    if (error_count == 0)
        $display("All %0d tests PASSED!", test_count);
    else
        $display("%0d out of %0d tests FAILED!", error_count, test_count);

    #(CLK_PERIOD*5);
    $finish;
  end

  // Optional waveform dump for viewing in a waveform viewer
  initial begin
    $dumpfile("lifo_tb.vcd");
    $dumpvars(0, TB);
  end

endmodule
```

Mô phỏng: