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);

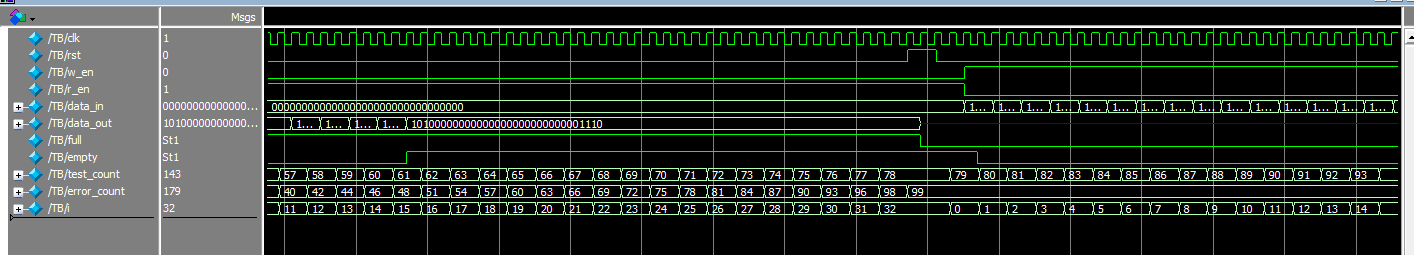
#(CLK\_PERIOD\*5);

$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:***

