Nang Thiri Wutyi

20113

9/30/2024

**Verilog Lab HW 2**

1. **ReductionOperators**

module ReductionOperators();

initial begin

// Bit Wise AND reduction

$display (" & 4'b1001 = %b", (&4'b1001));

$display (" & 4'bx111 = %b", (&4'bx111));

$display (" & 4'bz111 = %b", (&4'bz111));

// Bit Wise NAND reduction

$display (" ~& 4'b1001 = %b", (~&4'b1001));

$display (" ~& 4'bx001 = %b", (~&4'bx001));

$display (" ~& 4'bz001 = %b", (~&4'bz001));

// Bit Wise OR reduction

$display (" | 4'b1001 = %b", (|4'b1001));

$display (" | 4'bx000 = %b", (|4'bx000));

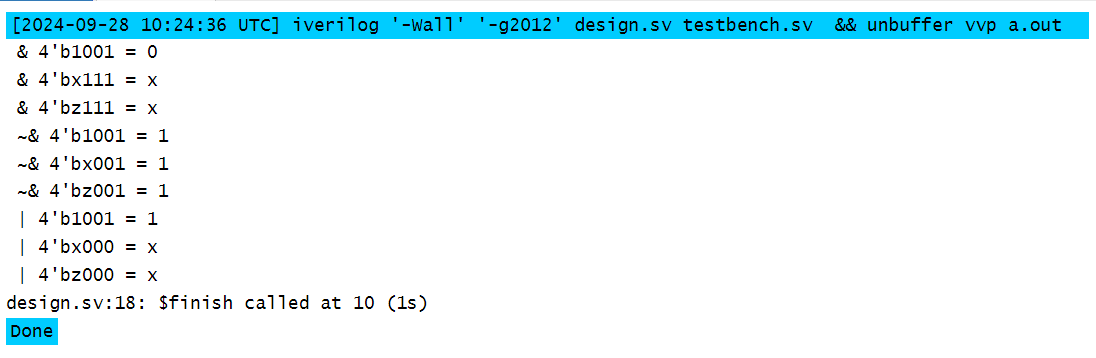
$display (" | 4'bz000 = %b", (|4'bz000));

#10 $finish;

end

endmodule

**Results**

****

**ShiftOperators**

module ShiftOperators();

initial begin

// Left Shift

$display (" 4'b1001 << 1 = %b", (4'b1001 << 1));

$display (" 4'b10x1 << 1 = %b", (4'b10x1 << 1));

$display (" 4'b10z1 << 1 = %b", (4'b10z1 << 1));

// Right Shift

$display (" 4'b1001 >> 1 = %b", (4'b1001 >> 1));

$display (" 4'b10x1 >> 1 = %b", (4'b10x1 >> 1));

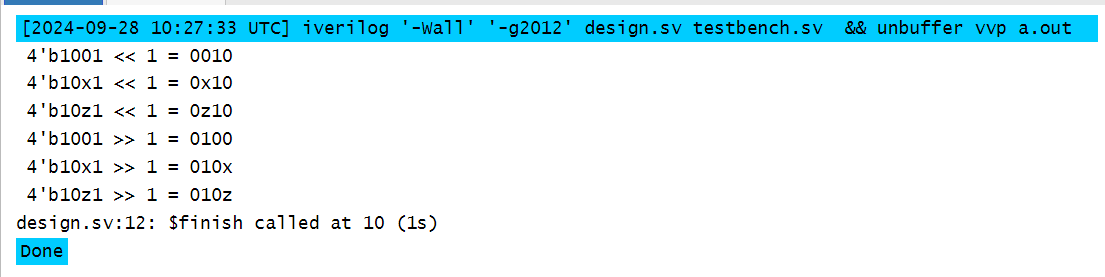
$display (" 4'b10z1 >> 1 = %b", (4'b10z1 >> 1));

#10 $finish;

end

endmodule

**Results**



**ConcatenationOperator**

module ConcatenationOperator();

initial begin

// Concatenation

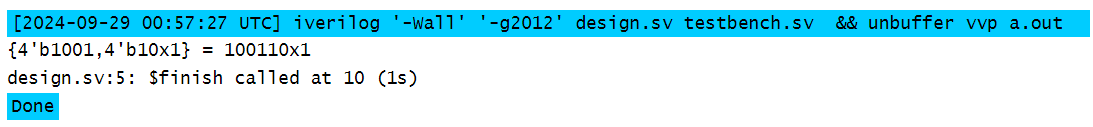
$display ("{4'b1001,4'b10x1} = %b", {4'b1001, 4'b10x1});

#10 $finish;

end

endmodule

**Results**

****

**ReplicationOperator**

module ReplicationOperator();

initial begin

// replication

$display ("{4{4'b1001}} = %b", {4{4'b1001}});

// replication and concatenation

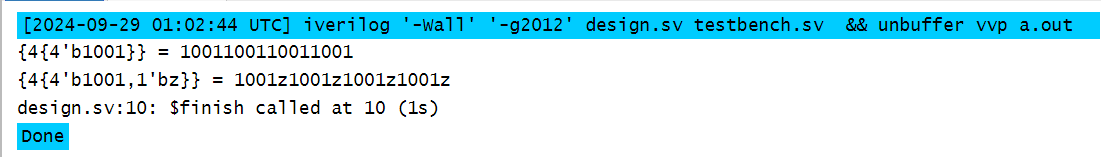
$display ("{4{4'b1001,1'bz}} = %b", {4{4'b1001,1'bz}});

#10 $finish;

end

endmodule

Results



1. **Addbit module**

// Addbit Module

module addbit (

input a, // First bit

input b, // Second bit

input cin, // Carry in

output sum, // Sum output

output cout // Carry out

);

// Full adder logic

assign sum = a ^ b ^ cin; // Sum is the XOR of the inputs and carry in

assign cout = (a & b) | (cin & (a ^ b)); // Carry out is generated based on inputs and carry in

endmodule

**Adder Hier Module**

// Adder Hier Module

module adder\_hier (

output [3:0] result\_o, // 4-bit sum output

output carry\_o, // Carry out of the final addition

input [3:0] r1\_i, // First 4-bit input

input [3:0] r2\_i, // Second 4-bit input

input ci\_i // Carry in input

);

// Internal wires for carry between stages

wire c1\_w, c2\_w, c3\_w;

// Instantiate addbit modules for each bit

addbit u0 (

.a(r1\_i[0]), .b(r2\_i[0]), .cin(ci\_i),

.sum(result\_o[0]), .cout(c1\_w)

);

addbit u1 (

.a(r1\_i[1]), .b(r2\_i[1]), .cin(c1\_w),

.sum(result\_o[1]), .cout(c2\_w)

);

addbit u2 (

.a(r1\_i[2]), .b(r2\_i[2]), .cin(c2\_w),

.sum(result\_o[2]), .cout(c3\_w)

);

addbit u3 (

.a(r1\_i[3]), .b(r2\_i[3]), .cin(c3\_w),

.sum(result\_o[3]), .cout(carry\_o)

);

endmodule

**Testbench Module**

// Testbench Module

module tb\_adder\_hier();

// Declare registers for inputs and wires for outputs

reg [3:0] r1\_r, r2\_r;

reg ci\_r;

wire [3:0] result\_w;

wire carry\_w;

// Instantiate the adder\_hier module

adder\_hier UUT (

.result\_o(result\_w),

.carry\_o(carry\_w),

.r1\_i(r1\_r),

.r2\_i(r2\_r),

.ci\_i(ci\_r)

);

// Drive the inputs

initial begin

// Test case 1: Add 0 + 0 + 0

r1\_r = 4'b0000;

r2\_r = 4'b0000;

ci\_r = 1'b0;

#10; // Wait for 10 time units

// Test case 2: Add 10 (0b1010) + 0 + 0

r1\_r = 4'b1010;

r2\_r = 4'b0000;

ci\_r = 1'b0;

#10;

// Test case 3: Add 10 (0b1010) + 2 (0b0010) + 0

r1\_r = 4'b1010;

r2\_r = 4'b0010;

ci\_r = 1'b0;

#10;

// Test case 4: Add 10 (0b1010) + 2 (0b0010) + 1 (carry-in)

r1\_r = 4'b1010;

r2\_r = 4'b0010;

ci\_r = 1'b1;

#10;

$finish; // End simulation

end

// Monitor the inputs and outputs

initial begin

$display("+-----------------------------------------------+");

$display("| r1 | r2 | ci | result | carry |");

$display("+-----------------------------------------------+");

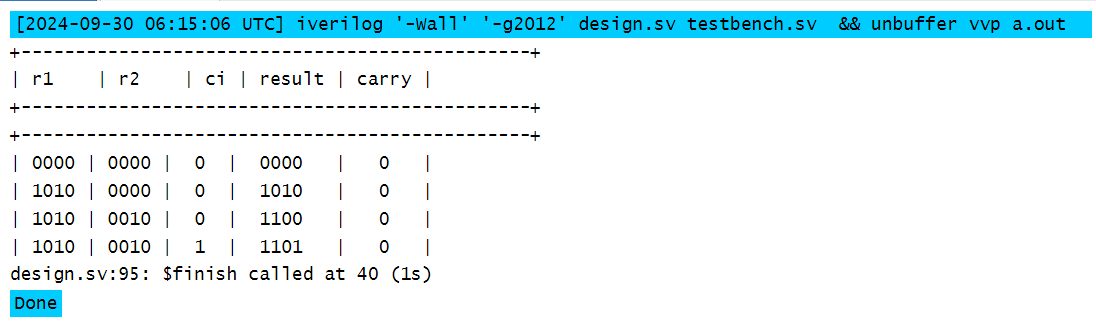
$monitor("| %b | %b | %b | %b | %b |", r1\_r, r2\_r, ci\_r, result\_w, carry\_w);

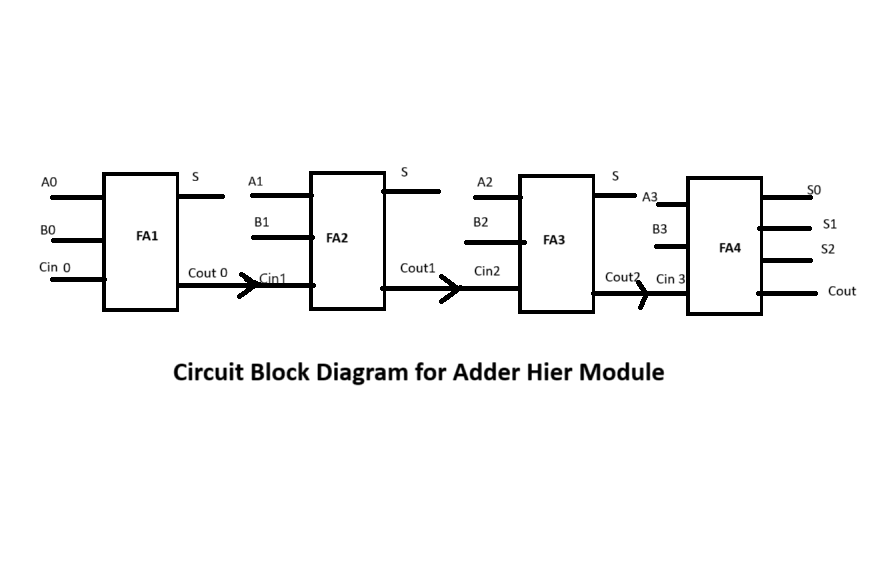
$display("+-----------------------------------------------+");

end

endmodule

**Results**

****

****

**Synchronous DFF**

// D Flip-Flop Module

`timescale 1ns / 1ps // Set the time unit to 1ns and precision to 1ps

module DFFSynch(

input d\_i, // Data input

input rst\_i, // Reset input

input clk\_i, // Clock input

output reg q\_o // Output

);

always @(posedge clk\_i) begin

if (rst\_i)

q\_o <= 0; // Reset output to 0

else

q\_o <= d\_i; // Update output with data input on clock edge

end

endmodule

**Testbench**

// Testbench for D Flip-Flop

`timescale 1ns / 1ps // Set the time unit and precision

module tb\_DFFSynch;

// Declare registers for inputs and wire for output

reg d\_i; // Input data

reg rst\_i; // Reset input

reg clk\_i; // Clock input

wire q\_o; // Output

// Instantiate the DFF module

DFFSynch dut (

.d\_i(d\_i),

.rst\_i(rst\_i),

.clk\_i(clk\_i),

.q\_o(q\_o)

);

// Clock generation

initial begin

clk\_i = 0; // Initialize clock

forever #5 clk\_i = ~clk\_i; // Toggle clock every 5 time units

end

// Test sequence

initial begin

// Set up the VCD file for waveform viewing

$dumpfile("dump.vcd"); // Specify the name of the VCD file

$dumpvars(0, tb\_DFFSynch); // Dump all variables in the testbench

// Monitor the output

$monitor("Time: %0dns | d\_i: %b | rst\_i: %b | clk\_i: %b | q\_o: %b", $time, d\_i, rst\_i, clk\_i, q\_o);

// Test case 1: Reset the DFF

rst\_i = 1; d\_i = 0; #10; // Apply reset

rst\_i = 0; #10; // Release reset

// Test case 2: Set d\_i to 1

d\_i = 1; #10; // Set d\_i to 1, should update q\_o on next clock

// Test case 3: Set d\_i to 0

d\_i = 0; #10; // Set d\_i to 0, should update q\_o on next clock

// Test case 4: Check reset again

rst\_i = 1; #10; // Apply reset again

rst\_i = 0; #10; // Release reset

// Test case 5: Set d\_i to 1 again

d\_i = 1; #10; // Set d\_i to 1, should update q\_o on next clock

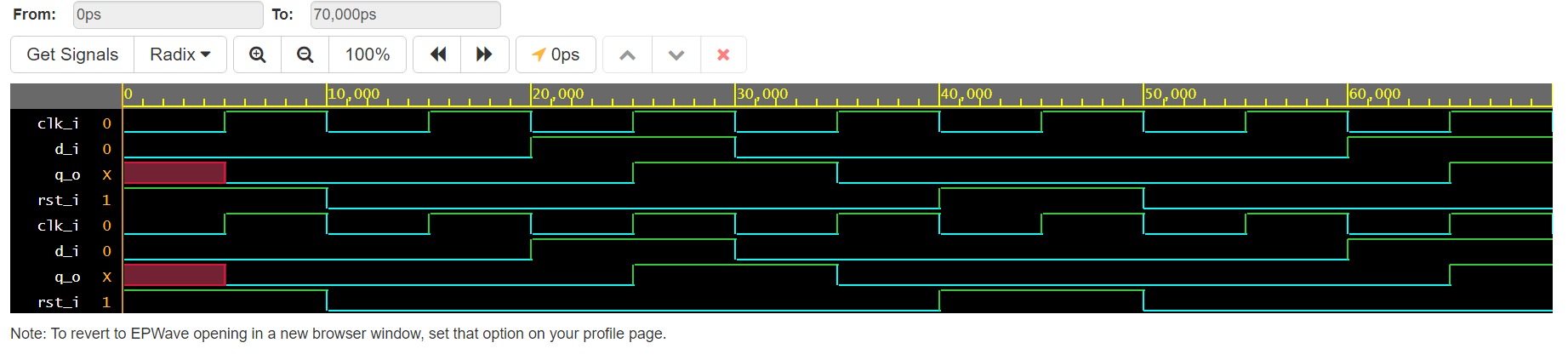
// Finish the simulation

$finish;

end

endmodule

**Results**



**Asynchronous DFF**

// Asynchronous D Flip-Flop Module

`timescale 1ns / 1ps // Set the time unit and precision

module DFFAsynch(

input d\_i, // Data input

input rst\_i, // Reset input

input clk\_i, // Clock input

output reg q\_o // Output

);

always @(posedge clk\_i or posedge rst\_i) begin

if (rst\_i)

q\_o <= 0; // Reset output to 0

else

q\_o <= d\_i; // Update output with data input on clock edge

end

endmodule

**Testbench**

// Testbench for DFFAsynch

`timescale 1ns / 1ps // Set the time unit and precision

module tb\_DFFAsynch;

// Declare registers for inputs and wire for output

reg d\_i; // Input data

reg rst\_i; // Reset input

reg clk\_i; // Clock input

wire q\_o; // Output

// Instantiate the DFFAsynch module

DFFAsynch dut (

.d\_i(d\_i),

.rst\_i(rst\_i),

.clk\_i(clk\_i),

.q\_o(q\_o)

);

// Clock generation

initial begin

clk\_i = 0; // Initialize clock

forever #5 clk\_i = ~clk\_i; // Toggle clock every 5 time units

end

// Test sequence

initial begin

// Set up the VCD file for waveform viewing

$dumpfile("dump.vcd"); // Specify the name of the VCD file

$dumpvars(0, tb\_DFFAsynch); // Dump all variables in the testbench

// Monitor the output

$monitor("Time: %0dns | d\_i: %b | rst\_i: %b | clk\_i: %b | q\_o: %b", $time, d\_i, rst\_i, clk\_i, q\_o);

// Test case 1: Reset the DFF

rst\_i = 1; d\_i = 0; #10; // Apply reset

rst\_i = 0; #10; // Release reset

// Test case 2: Set d\_i to 1

d\_i = 1; #10; // Set d\_i to 1, should update q\_o on next clock

// Test case 3: Set d\_i to 0

d\_i = 0; #10; // Set d\_i to 0, should update q\_o on next clock

// Test case 4: Check reset functionality

rst\_i = 1; #10; // Apply reset again

rst\_i = 0; #10; // Release reset

// Test case 5: Set d\_i to 1 again

d\_i = 1; #10; // Set d\_i to 1, should update q\_o on next clock

// Finish the simulation

$finish;

end

endmodule

**Results**

