Verilog assignment 2

P Venkata Chanakya(ee23btech11048)

1 Convolution

convolution Module

```
module convolution (
    // Inputs
    input [3:0] x0, x1, x2, x3, x4, x5, x6, x7,
    input [3:0] h0, h1, h2, h3, h4, h5, h6, h7,
    // Output
    output reg [3:0] y0, y1, y2, y3, y4, y5, y6, y7,
    output reg [3:0] y8, y9, y10, y11, y12, y13, y14, y15,
     input  clk
);

    // Convolution operation
    always @* begin
        y0  = x0  * h7  + x1  * h6  + x2  * h5  + x3  * h4
            +
              x4  * h3  + x5  * h2  + x6  * h1  + x7  * h0;
        y1  = x0  * h6  + x1  * h5  + x2  * h4  + x3  * h3
            +
              x4  * h2  + x5  * h1  + x6  * h0  + x7  * h7;
        y2  = x0  * h5  + x1  * h4  + x2  * h3  + x3  * h2
            +
              x4  * h1  + x5  * h0  + x6  * h7  + x7  * h6;
        y3  = x0  * h4  + x1  * h3  + x2  * h2  + x3  * h1
            +
              x4  * h0  + x5  * h7  + x6  * h6  + x7  * h5;
        y4  = x0  * h3  + x1  * h2  + x2  * h1  + x3  * h0
            +
              x4  * h7  + x5  * h6  + x6  * h5  + x7  * h4;
        y5  = x0  * h2  + x1  * h1  + x2  * h0  + x3  * h7
            +
              x4  * h6  + x5  * h5  + x6  * h4  + x7  * h3;
        y6  = x0  * h1  + x1  * h0  + x2  * h7  + x3  * h6
            +
              x4  * h5  + x5  * h4  + x6  * h3  + x7  * h2;
        y7  = x0  * h0  + x1  * h7  + x2  * h6  + x3  * h5
            +
```

```
                x4  * h4  + x5  * h3  + x6  * h2  + x7  * h1;
        y8  = x1  * h0  + x2  * h7  + x3  * h6  + x4  * h5
            +
                x5  * h4  + x6  * h3  + x7  * h2  + x0  * h1;
        y9  = x2  * h0  + x3  * h7  + x4  * h6  + x5  * h5
            +
                x6  * h4  + x7  * h3  + x0  * h2  + x1  * h1;
        y10 = x3  * h0  + x4  * h7  + x5  * h6  + x6  * h5
            +
                x7  * h4  + x0  * h3  + x1  * h2  + x2  * h1;
        y11 = x4  * h0  + x5  * h7  + x6  * h6  + x7  * h5
            +
                x0  * h4  + x1  * h3  + x2  * h2  + x3  * h1;
        y12 = x5  * h0  + x6  * h7  + x7  * h6  + x0  * h5
            +
                x1  * h4  + x2  * h3  + x3  * h2  + x4  * h1;
        y13 = x6  * h0  + x7  * h7  + x0  * h6  + x1  * h5
            +
                x2  * h4  + x3  * h3  + x4  * h2  + x5  * h1;
        y14 = x7  * h0  + x0  * h7  + x1  * h6  + x2  * h5
            +
                x3  * h4  + x4  * h3  + x5  * h2  + x6  * h1;
        y15 = x0  * h0  + x1  * h7  + x2  * h6  + x3  * h5
            +
                x4  * h4  + x5  * h3  + x6  * h2  + x7  * h1;
    end

endmodule
```

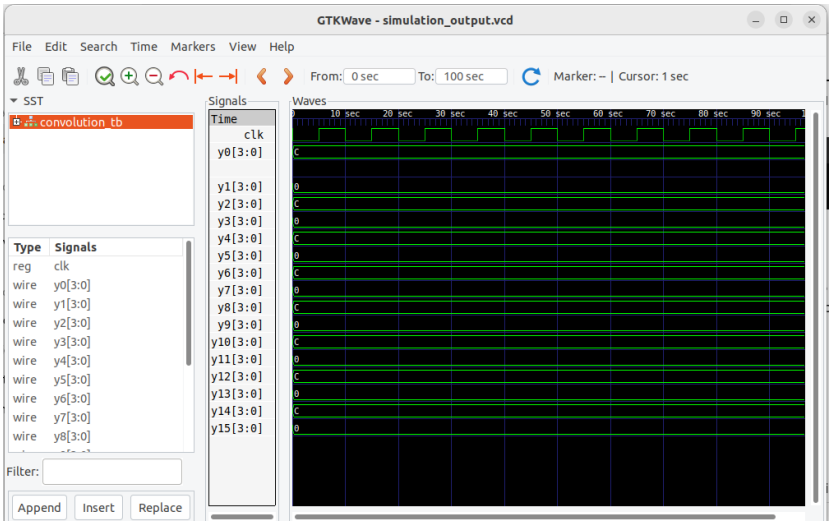
1.1 testbench code

```
module convolution_tb;

    // Inputs
    reg [3:0] x [0:7];
    reg [3:0] h [0:7];

    // Outputs
    wire [3:0] y0, y1, y2, y3, y4, y5, y6, y7;
    wire [3:0] y8, y9, y10, y11, y12, y13, y14, y15;

    // Clock
    reg clk = 0;

    // Instantiate the convolution module
    convolution dut (
        // Inputs
        .x0(x[0]), .x1(x[1]), .x2(x[2]), .x3(x[3]),
        .x4(x[4]), .x5(x[5]), .x6(x[6]), .x7(x[7]),
        .h0(h[0]), .h1(h[1]), .h2(h[2]), .h3(h[3]),
        .h4(h[4]), .h5(h[5]), .h6(h[6]), .h7(h[7]),
        // Outputs
        .y0(y0), .y1(y1), .y2(y2), .y3(y3),
        .y4(y4), .y5(y5), .y6(y6), .y7(y7),
        .y8(y8), .y9(y9), .y10(y10), .y11(y11),
        .y12(y12), .y13(y13), .y14(y14), .y15(y15),
        // Clock (assuming it's named clk in convolution
            module)
        .clk(clk)
    );

    // Clock generation
    always #5 clk = ~clk;

    // Stimulus
    initial begin
        // Initialize inputs
        x[0] = 4'b0001; x[1] = 4'b0010; x[2] = 4'b0011; x[3]
            = 4'b0100;
        x[4] = 4'b0101; x[5] = 4'b0110; x[6] = 4'b0111; x[7]
            = 4'b1000;

        h[0] = 4'b1000; h[1] = 4'b0111; h[2] = 4'b0110; h[3]
            = 4'b0101;
        h[4] = 4'b0100; h[5] = 4'b0011; h[6] = 4'b0010; h[7]
            = 4'b0001;

        // Dump VCD file
        $dumpfile("simulation_output.vcd");
```

```
        $dumpvars(0, convolution_tb);

        // Simulate for 100 time units
        #100;

        // Display output
        $display("Output vector y:");
        $display("y0 = %b", y0);
        $display("y1 = %b", y1);
        $display("y2 = %b", y2);
        $display("y3 = %b", y3);
        $display("y4 = %b", y4);
        $display("y5 = %b", y5);
        $display("y6 = %b", y6);
        $display("y7 = %b", y7);
        $display("y8 = %b", y8);
        $display("y9 = %b", y9);
        $display("y10 = %b", y10);
        $display("y11 = %b", y11);
        $display("y12 = %b", y12);
        $display("y13 = %b", y13);
        $display("y14 = %b", y14);
        $display("y15 = %b", y15);

        // End simulation
        $finish;
    end

endmodule
```

Results

```
Output vector y:
y0 = 1100
y1 = 0000
y2 = 1100
y3 = 0000
y4 = 1100
y5 = 0000
y6 = 1100
y7 = 0000
y8 = 1100
y9 = 0000
y10 = 1100
y11 = 0000
y12 = 1100
y13 = 0000
y14 = 1100
y15 = 0000
```



The provided Verilog code consists of two modules: `convolution` and `convolution_tb`. The `convolution` module implements a convolution operation, which takes eight input signals (`x0` to `x7`) and eight filter coefficients (`h0` to `h7`). It performs convolution on these inputs and produces sixteen output signals (`y0` to `y15`). The `convolution_tb` module serves as a testbench for the `convolution` module. It provides stimulus to the `convolution` module by initializing the input signals and filter coefficients. The testbench generates a clock signal (`clk`) and drives it to the `convolution` module. After a certain simulation time, it displays the output signals (`y0` to `y15`) and finishes the simulation. The Verilog

code is compiled and executed using Icarus Verilog, and the simulation results are visualized using GTKWave.

2 Universal Shift Register Verilog Code Explanation

Universal Shift Register Module

```
module universal_shift_reg(
    input clk, rst_n,
    input [8:0] select,
    input [15:0] parallel_in,
    input serial_left_data_in,
    input serial_right_data_in,
    output reg [15:0] p_dout,
    output reg s_left_dout,
    output reg s_right_dout
);
    // Internal signals and registers
    reg [15:0] shift_reg;

    // Parallel-in Parallel-out (PIPO) mode
    always @(posedge clk or negedge rst_n) begin
        if (~rst_n) begin
            shift_reg <= 16'b0;
        end else begin
            if (select[8]) begin
                shift_reg <= parallel_in;
            end
        end
    end

    // Serial-in Serial-out (SISO) mode
    always @(posedge clk or negedge rst_n) begin
        if (~rst_n) begin
            // Reset shift register
            shift_reg <= 16'b0;
        end else begin
            if (select[0]) begin
                // Shift left
                shift_reg <= {shift_reg[14:0],
                    serial_left_data_in};
            end else if (select[1]) begin
                // Shift right
                shift_reg <= {serial_right_data_in,
                    shift_reg[15:1]};
            end
        end
    end

    // Outputs
    assign p_dout = shift_reg;
```

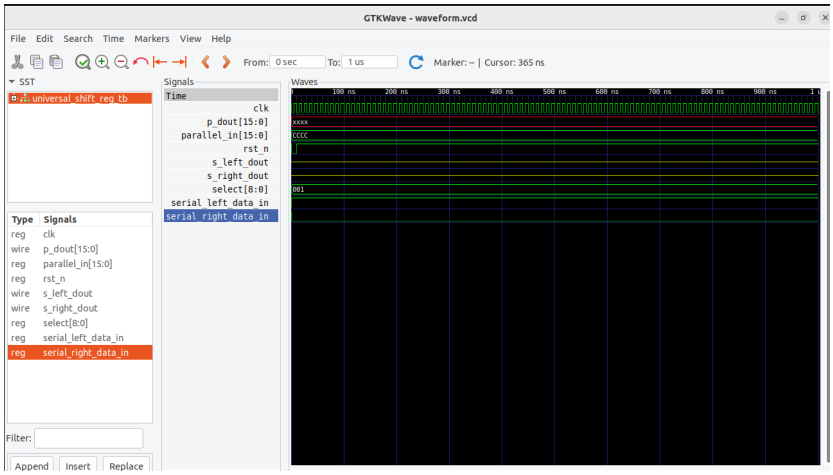
```
43      assign s_left_dout = shift_reg[15];
44      assign s_right_dout = shift_reg[0];
45
46  endmodule
```

Universal Shift Register Testbench

```
module universal_shift_reg_tb;

    // Inputs
    reg clk;
    reg rst_n;
    reg [8:0] select;
    reg [15:0] parallel_in;
    reg serial_left_data_in;
    reg serial_right_data_in;

    // Outputs
    wire [15:0] p_dout;
    wire s_left_dout;
    wire s_right_dout;

    // Instantiate the Universal Shift Register module
    universal_shift_reg dut (
        .clk(clk),
        .rst_n(rst_n),
        .select(select),
        .parallel_in(parallel_in),
        .serial_left_data_in(serial_left_data_in),
        .serial_right_data_in(serial_right_data_in),
        .p_dout(p_dout),
        .s_left_dout(s_left_dout),
        .s_right_dout(s_right_dout)
    );

    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    // Reset generation
    initial begin
        rst_n = 1'b0;
        #10;
        rst_n = 1'b1;
        #10;
    end

    // Stimulus
    initial begin
        // Initialize inputs
        select = 9'b000000001; // Example: SISO left mode
        parallel_in = 16'b1100110011001100;
        serial_left_data_in = 1'b1;
```

```
49          serial_right_data_in = 1'b0;
50
51          // End simulation after some time
52          #1000;
53          $finish;
54      end
55
56      // Dump VCD file
57      initial begin
58          $dumpfile("waveform.vcd");
59          $dumpvars(0, universal_shift_reg_tb);
60      end
61
62  endmodule
```

The provided Verilog code implements a Universal Shift Register, a digital circuit capable of shifting data in multiple modes. The module `universal_shift_reg` defines the behavior of the shift register based on the input signals `clk`, `rst_n`, `select`, `parallel_in`, `serial_left_data_in`, and `serial_right_data_in`. The shift register operates in three modes: Parallel-in Parallel-out (PIPO), Serial-in Serial-out (SISO), and Serial-out Serial-in (SOSI). In the PIPO mode, the register loads data in parallel from the `parallel_in` input. In the SISO mode, the register shifts data left or right based on the `select` input, with the leftmost or rightmost bit being replaced by `serial_left_data_in` or `serial_right_data_in`, respectively. The output `p_dout` provides the parallel output of the shift register, while `s_left_dout` and `s_right_dout` provide the leftmost and rightmost bits of the serial output, respectively. The testbench `universal_shift_reg_tb` verifies the functionality of the shift register by providing stimulus to its inputs and generating waveform data for analysis using a VCD file. The testbench initializes the inputs, toggles the clock, and terminates the simulation after a specified time. Overall, the code enables the simulation and verification of a versatile Universal Shift Register design.


GTKWave - waveform.vcd
File Edit Search Time Markers View Help
From: 0 sec
To: 1 us
Marker: – | Cursor: 365 ns
SST
universal_shift_reg_tb
Type Signals
reg clk
wire p_dout[15:0]
reg parallel_in[15:0]
reg rst_n
wire s_left_dout
wire s_right_dout
reg select[8:0]
reg serial_left_data_in
reg serial_right_data_in
Filter:
Append Insert Replace
Signals
Time
clk
p_dout[15:0]
parallel_in[15:0]
rst_n
s_left_dout
s_right_dout
select[8:0]
serial_left_data_in
serial_right_data_in
Waves
100 ns 200 ns 300 ns 400 ns 500 ns 600 ns 700 ns 800 ns 900 ns
xxxx
cccc
001

3 wallace$_{t}ree_{m}ultiplier$

3.1 wallace$_{t}ree_{m}ultipliermodule$

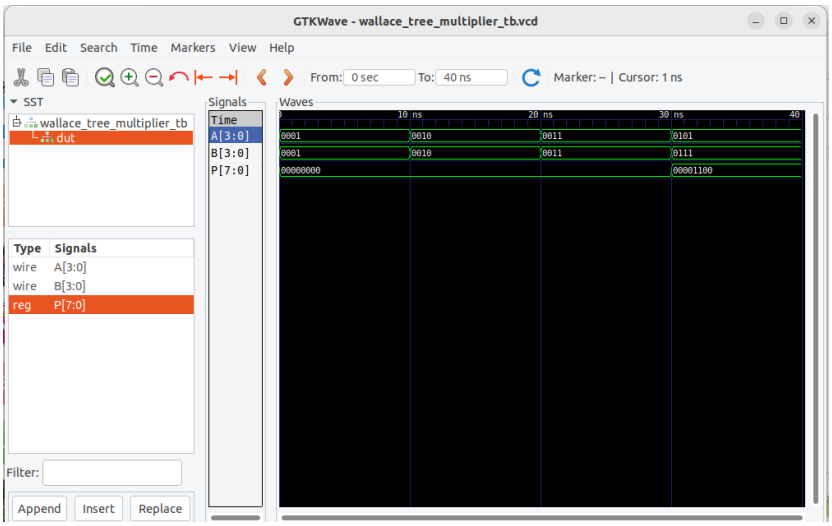
```
module wallace_tree_multiplier (
    input [3:0] A, // 4-bit input A
    input [3:0] B, // 4-bit input B
    output reg [7:0] P // 8-bit output P
);

    reg [5:0] S [0:2]; // Sum register array
    reg [5:0] C [0:1]; // Carry register array
    reg [3:0] P_part [0:3]; // Partial product register
        array

    // Generate partial products
    always @* begin
        P_part[0] = {A[0], B[0], 2'b00};
        P_part[1] = {A[1], B[1], 2'b00};
        P_part[2] = {A[2], B[2], 2'b00};
        P_part[3] = {A[3], B[3], 2'b00};
    end

    // Wallace Tree Reduction
    always @* begin
        // First Reduction Stage
        S[0] = P_part[0] + P_part[1];
        C[0] = &{1'b0, P_part[0], P_part[1]};

        // Second Reduction Stage
        S[1] = P_part[2] + P_part[3] + C[0];
        C[1] = &{P_part[2], P_part[3], C[0]};

        // Final Reduction Stage
        P = S[1] + C[1];
    end

endmodule
```

3.2 testbench code

```
`timescale 1ns / 1ps

module wallace_tree_multiplier_tb;

    // Inputs
    reg [3:0] A;
    reg [3:0] B;

    // Output
    wire [7:0] P;

    // Instantiate the Wallace Tree Multiplier module
    wallace_tree_multiplier dut (
        .A(A),
        .B(B),
        .P(P)
    );

    // Clock
    reg clk = 1'b0;

    // Toggle clock
    always #5 clk = ~clk;

    // Stimulus
    initial begin
        $dumpfile("wallace_tree_multiplier_tb.vcd");
        $dumpvars(0, wallace_tree_multiplier_tb);

        // Test cases
        A = 4'b0001; B = 4'b0001; #10; // A * B = 1 * 1 = 1
        A = 4'b0010; B = 4'b0010; #10; // A * B = 2 * 2 = 4
        A = 4'b0011; B = 4'b0011; #10; // A * B = 3 * 3 = 9
        A = 4'b0101; B = 4'b0111; #10; // A * B = 5 * 7 = 35

        // Add more test cases here if needed

        // End simulation
        $finish;
    end

    // Display results
    always @* begin
        $display("A = %b, B = %b, P = %b", A, B, P);
    end

endmodule
```

Result

```
A = 0001, B = 0001, P = xxxxxxxx
A = 0001, B = 0001, P = 00000000
A = 0010, B = 0010, P = 00000000
A = 0011, B = 0011, P = 00000000
A = 0101, B = 0111, P = 00000000
A = 0101, B = 0111, P = 00001100
```



The provided Verilog code consists of two modules: `wallace_tree_multiplier` and `wallace_tree_multiplier_tb`.

The `wallace_tree_multiplier` module implements a Wallace Tree Multiplier, which is a high-speed multiplier architecture. It takes two 4-bit inputs, `A` and `B`, and produces an 8-bit output `P`.

Inside the module, it generates partial products (`P_part`) by multiplying corresponding bits of `A` and `B` and appending two zeros. Then, it performs Wallace Tree Reduction to sum up these partial products efficiently.

The `wallace_tree_multiplier_tb` module serves as a testbench for the `wallace_tree_multiplier`. It provides stimulus to the multiplier by assigning test cases to inputs `A` and `B`. The clock signal `clk` is toggled periodically to drive the simulation. The testbench captures the simulation results and displays them using `$display`.

Test cases include simple multiplication scenarios like 1*1, 2*2, 3*3, and 5*7, and you can add more test cases as needed.

The simulation results are dumped into a VCD file named `wallace_tree_multiplier_tb.vcd`. You can visualize the simulation results using waveform viewers like GTKWave.