STACK PROCESSER

Project report Front End Simulation and FPGA Synthesis Lab

M. SHANMUKHA SRI DATTA (CB.EN.P2VLD24007)

Abstract:

This project focuses on designing and implementing a stack processor using Verilog, tested through simulation. A stack processor is a specialized architecture that utilizes a Last-In, First-Out (LIFO) approach to manage data. The objective is to develop a stack processor capable of performing basic arithmetic operations—such as addition, subtraction, multiplication, and division as well as logical operations like AND, OR, XOR, and NOT.The design consists of two main components: stack memory and control logic. The stack memory holds data in a 16-element stack, where data can be added (PUSH) or retrieved (POP) according to the LIFO principle. Arithmetic and logical operations are performed on the stack's top elements, with results returned to the stack or sent out as output. The control unit manages these operations, instructing the stack on when to perform specific actions and ensuring that operations are executed in the correct sequence.Verilog code is used to define this control and data handling logic, which is then simulated to verify functionality. The design was tested in a Verilog testbench, confirming that the processor performs each operation accurately, making it a promising design for applications needing a compact, stack-based processing unit.

Introduction:

Stack-based processors are essential in computing systems for efficiently handling nested subroutine calls and managing temporary data storage. These processors use a stack, a specialized data structure where data is added (pushed) or removed (popped) according to the LIFO principle. In this project, we design a simple 8-bit stack processor using Verilog that performs essential operations using an internal stack memory. The processor supports a variety of instructions including arithmetic (addition, subtraction, multiplication, and division) and logical (AND, OR, XOR, NOT) operations, which are common in computational tasks. By implementing this stack processor, we aim to explore the foundational principles of stack-based processing and evaluate its functionality through simulation.

Methodology:

The project was implemented using Verilog hardware description language and simulated using a Verilog testbench to verify correct functionality. The stack processor module, StackProcessor, includes a stack of 16 elements, each 8 bits wide, and a stack pointer (sp) that tracks the top of the stack. The stack operates synchronously on a positive clock edge with a reset option for initialization.

Push (0001): Pushes an 8-bit data input onto the stack and increments the stack pointer.

Pop (0010): Pops the top value from the stack, decrements the stack pointer, and outputs the value.

Arithmetic Operations: The processor supports add (0011), subtract (0100), multiply (0101), and divide (0110) operations, each of which operates on the top two stack elements, stores the result back into the stack, and decrements the stack pointer.

Logical Operations: The processor performs AND (0111), OR (1000), XOR (1001), and NOT (1010) operations, manipulating the topmost stack elements and updating the result back on the stack.

Source Code:

```
module StackProcessor(
    input clk,
    input rst,
    input [3:0] opcode,
    input [7:0] data_in,
    output reg [7:0] data_out
);

    reg [7:0] stack [0:15];   // 16-element stack, each 8 bits wide
    reg [3:0] sp;             // Stack pointer

    // Synchronous reset to avoid issues in timing simulation
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            sp <= 4'b0;
            data_out <= 8'b0;
        end else begin
            case (opcode)
                4'b0001: begin // PUSH
                    if (sp < 15) begin
                        stack[sp] <= data_in;
                        sp <= sp + 1;
                    end
                end
                4'b0010: begin // POP
                    if (sp > 0) begin
                        sp <= sp - 1;
                        data_out <= stack[sp - 1]; // Update data_out with the value after decrement
                    end else begin
                        data_out <= 8'b0; // If stack is empty, output 0
                    end
                end
                4'b0011: begin // ADD
                    if (sp > 1) begin
                        stack[sp - 2] = stack[sp - 2] + stack[sp - 1];
                        sp <= sp - 1; // Reduce stack size after addition
                        data_out = stack[sp - 2]; // Update data_out with result of addition
                    end
                end
                4'b0100: begin // SUB
                    if (sp > 1) begin
                        stack[sp - 2] = stack[sp - 2] - stack[sp - 1];
                        sp <= sp - 1; // Reduce stack size after subtraction
                        data_out = stack[sp - 2]; // Update data_out with result of subtraction
                    end
                end
                4'b0101: begin // MULT
                    if (sp > 1) begin
                        stack[sp - 2] = stack[sp - 2] * stack[sp - 1];
                        sp <= sp - 1; // Reduce stack size after multiplication
                        data_out = stack[sp - 2]; // Update data_out with result of multiplication
                    end
                end
                4'b0110: begin // DIV
                    if (sp > 1 && stack[sp - 1] != 0) begin
                        stack[sp - 2] = stack[sp - 2] / stack[sp - 1];
                        sp <= sp - 1; // Reduce stack size after division
                        data_out = stack[sp - 2]; // Update data_out with result of division
                    end
                end
                4'b0111: begin // AND
                    if (sp > 1) begin
                        stack[sp - 2] = stack[sp - 2] & stack[sp - 1];
                        sp <= sp - 1; // Reduce stack size after AND
                        data_out = stack[sp - 2]; // Update data_out with result of AND
                    end
                end
                4'b1000: begin // OR
                    if (sp > 1) begin
                        stack[sp - 2] = stack[sp - 2] | stack[sp - 1];
                        sp <= sp - 1; // Reduce stack size after OR
                        data_out = stack[sp - 2]; // Update data_out with result of OR
                    end
                end
```

```
            4'b1001: begin // XOR
                if (sp > 1) begin
                    stack[sp - 2] = stack[sp - 2] ^ stack[sp - 1];
                    sp <= sp - 1; // Reduce stack size after XOR
                    data_out = stack[sp - 2]; // Update data_out with result of XOR
                end
            end
            4'b1010: begin // NOT
                if (sp > 0) begin
                    stack[sp - 1] = ~stack[sp - 1]; // Perform NOT on the top element
                    data_out = stack[sp - 1]; // Update data_out with result of NOT
                end
            end
        endcase
    end
end
endmodule
```

A testbench, StackProcessor_tb, was designed to simulate the behavior of the stack processor. The testbench initializes the processor with a clock and reset signal and then sequences through various operations, checking the results after each step to ensure correctness. Simulation steps include pushing values to the stack, performing operations, and popping results to verify the stack's state. The testbench begins by initializing the clock signal and applying a reset to clear the stack pointer and output. The testbench follows a sequence of operations, pushing values onto the stack and performing various arithmetic and logical instructions. The results are observed through the data_out output signal.

Testbench:

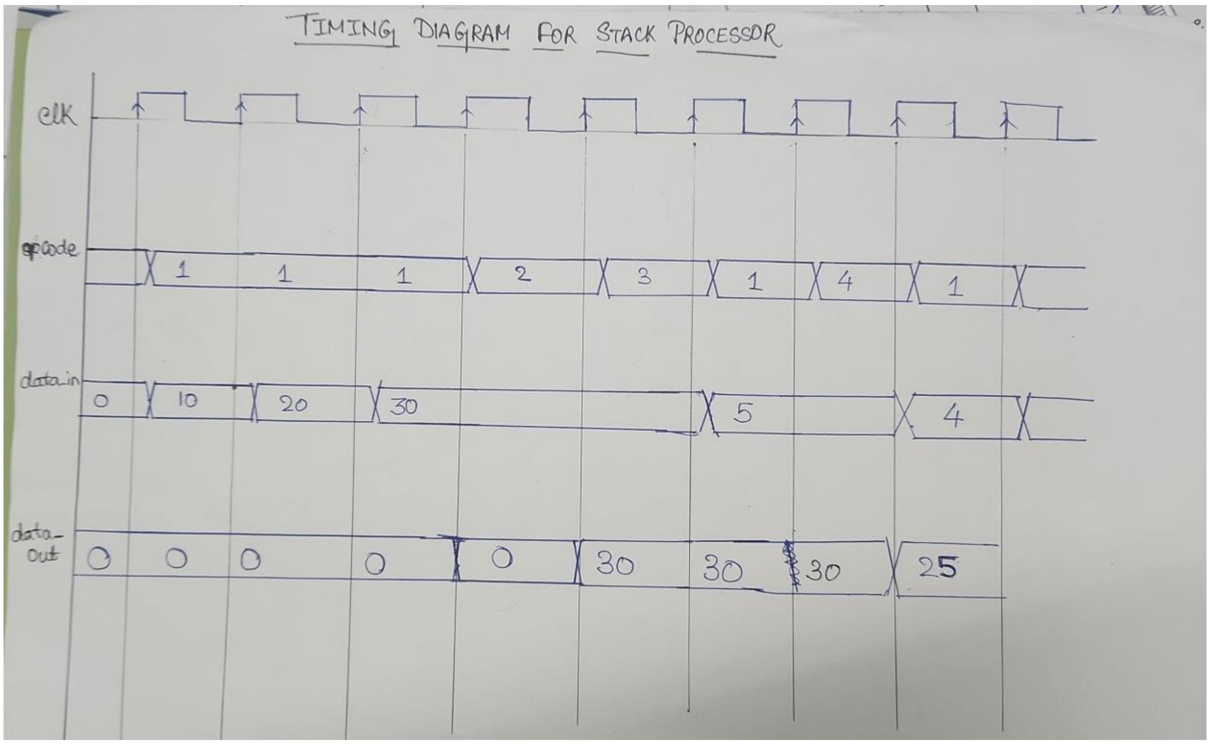
```
timescale 1ns / 1ns

module StackProcessor_tb;
    reg clk;
    reg rst;
    reg [3:0] opcode;
    reg [7:0] data_in;
    wire [7:0] data_out;

    // Instantiate the StackProcessor module
    StackProcessor uut (
        .clk(clk),
        .rst(rst),
        .opcode(opcode),
        .data_in(data_in),
    .data_out(data_out)
);
// Generate clock signal
initial begin
    clk <= 0;
    forever #200 clk = ~clk; // 20ns period (50MHz clock)
end
initial begin
    // Initialize inputs
    rst <= 1;
    opcode <= 4'b0000;
    data_in <= 8'b00000000;
    // Apply reset
    #20;
    rst <= 0;

  // Test PUSH operations
    @(posedge clk);
    #10 opcode <= 4'b0001;
    data_in <= 8'd10; // Push 10
    @(posedge clk);
    #10 opcode <= 4'b0001;
    data_in <= 8'd20; // Push 20
    @(posedge clk);
    #10 opcode <= 4'b0001;
    data_in <= 8'd30; // Push 30

    // Test POP operation
    @(posedge clk);
    #10 opcode <= 4'b0010; // Pop top value (30) into data_out

   // Test ADD operation
    @(posedge clk);
    #10 opcode <= 4'b0011; // Add top two values (10 + 20)

   // Test PUSH and SUB operation
    @(posedge clk);
    #10 opcode <= 4'b0001;
    data_in <= 8'd5; // Push 5
    @(posedge clk);
    #10 opcode <= 4'b0100; // Subtract top two values (30 - 5)

   // Test MULT operation
    @(posedge clk);
```

```
        #10 opcode <= 4'b0001;
        data_in <= 8'd4; // Push 4
        @(posedge clk);
        #10 opcode <= 4'b0101; // Multiply top two values (25 * 4)

        // Test DIV operation
        @(posedge clk);
        #10 opcode <= 4'b0001;
        data_in <= 8'd2; // Push 2
        @(posedge clk);
        #10 opcode <= 4'b0110; // Divide top two values (100 / 2)

         // Test AND operation
        @(posedge clk);
        #10 opcode <= 4'b0001;
        data_in <= 8'b00001111; // Push 15 (00001111)
        @(posedge clk);
        #10 opcode <= 4'b0111; // AND top two values (50 & 15) result is 2

        // Test OR operation
        @(posedge clk);
        #10 opcode <= 4'b0001;
        data_in <= 8'b11110000; // Push 240 (11110000)
        @(posedge clk);
        #10 opcode <= 4'b1000; // OR top two values (2 | 240) result is 242

        // Test XOR operation
        @(posedge clk);
        #10 opcode <= 4'b0001;
        data_in <= 8'b00001111; // Push 15 (00001111)
        @(posedge clk);
        #10 opcode <= 4'b1001; // XOR top two values (242 ^ 15) result is 253

        // Test NOT operation
        @(posedge clk);
        #10 opcode <= 4'b1010; // NOT top value result is 2

       // Final POP to check remaining stack
        @(posedge clk);
        #10 opcode <= 4'b0010; // Pop top value

end
endmodule
```

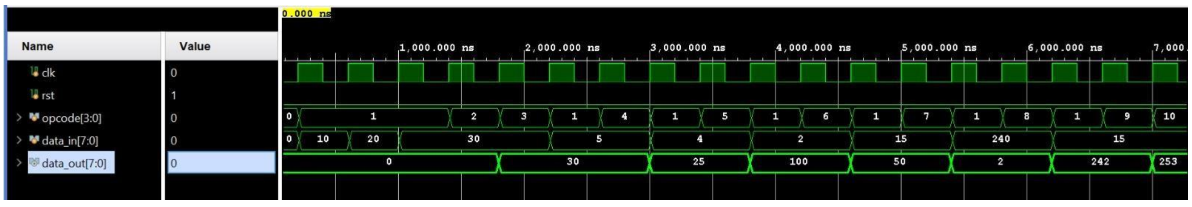
Results and Discussion:

The simulation results confirm that the stack processor correctly handles push, pop, arithmetic, and logical operations according to the specified instruction set. Values such as 10, 20, and 30 are successfully pushed onto the stack, with the stack pointer incrementing after each push. The pop instruction retrieves the last-pushed value, and the stack pointer decrements, correctly following the LIFO principle. The top two values (10 and 20) were added to yield 30, stored back at the top of the stack. Subsequent values (30 and 5) were subtracted, yielding 25. A multiply operation between 25 and 4 yielded a result of 100.The result of 100 / 2 yielded 50, confirming accurate arithmetic processing. The AND operation on 50 and 15 yielded 2.The OR operation on 2 and 240 yielded 242. The XOR operation on 242 and 15 resulted in 253.The NOT operation successfully inverted the topmost stack element.

Timing Diagram:



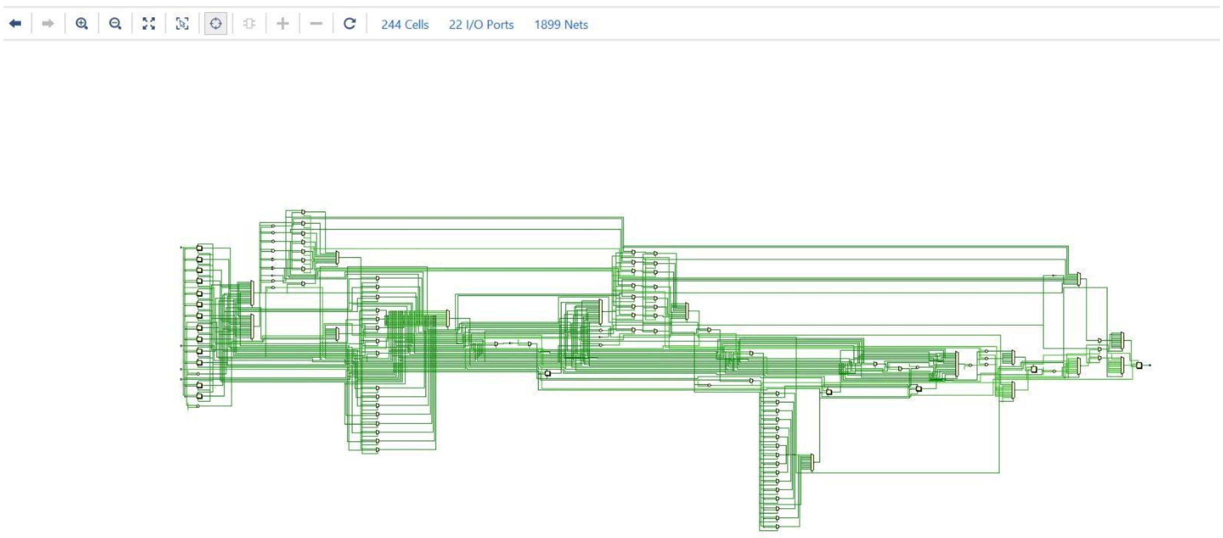
Behavioural Simulation Output:



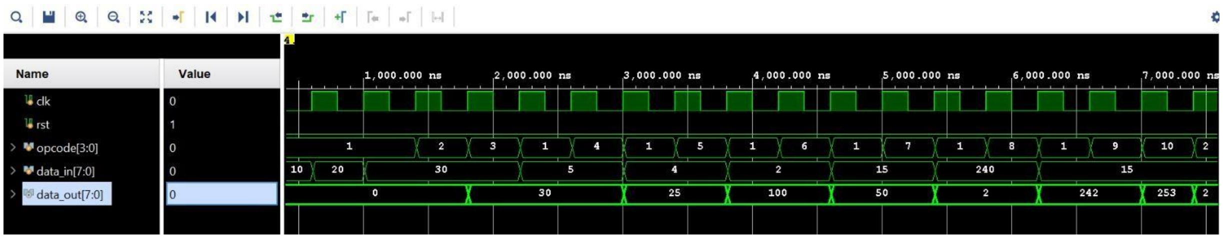
Name	Value
clk	0
rst	1
opcode[3:0]	0
data_in[7:0]	0
data_out[7:0]	0

0.000 ns
1,000.000 ns
2,000.000 ns
3,000.000 ns
4,000.000 ns
5,000.000 ns
6,000.000 ns
7,000.
opcode: 0 1 2 3 1 4 1 5 1 6 1 7 1 8 1 9 10
data_in: 0 10 20 30 5 4 2 15 240 15
data_out: 0 30 25 100 50 2 242 253

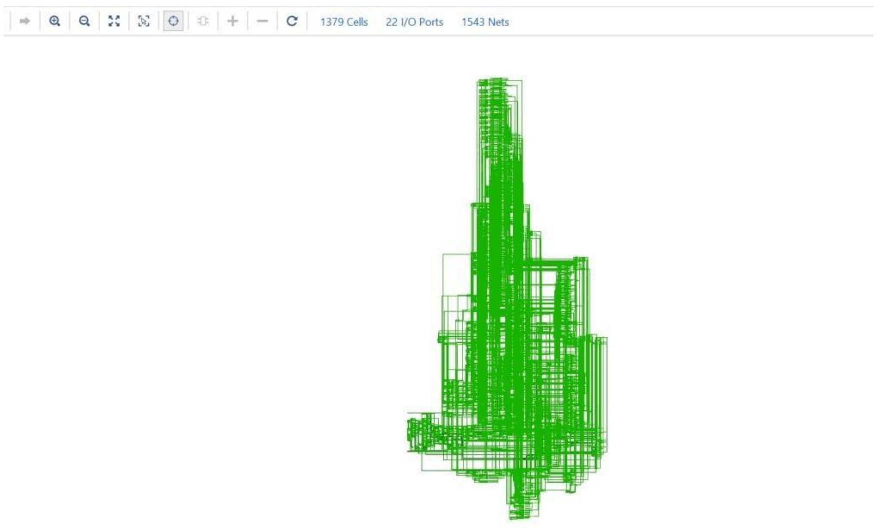
RTL Schemetic:



Post Implementation Output:



Synthesized Schematic:


1379 Cells
22 I/O Ports
1543 Nets