FPGA Based Stack Controller

Verilog Codes for FPGA -base Stack Controller Using Finite State Machine

Prepared BY:

Aabis Nasir (20-ES-3)

Ahtisham Hussain (20-ES-11)

Fahad Ahmed (20-ES-29)

Muhammad Shaheer (20-ES-63)

Submitted to:

Sir Waqar Alam

Objective:

The main objective of this CEP is to design an FPGA-based Stack Controller.

Introduction

How we organize our data is very important in systems. One common way to do this is using something called a "stack." It's like a pile of plates, where you can only add or remove from the top.

This report is all about creating a special circuit that watches over our stack to make sure everything goes smoothly. Our stack can only hold up to five items. We're focusing on three main actions: PUSH (adding something to the stack), POP (removing something from the stack), and EXCHNG (swapping items in the stack).

Since our stack has a limit, we need to make sure no one tries to add too much stuff (overflow) or take out something that isn't there (underflow). That's where our circuit comes in – it keeps an eye on everything to make sure it's all done correctly.

Literature Review

Paper 1:

This article suggests a way to design SFQ circuits (a type of electronic circuit) using Cadence Innovus, a computer program. It focuses on making the process automatic. They created special SFQ cells for this purpose. Scripts were made to change regular circuits into SFQ ones. The article explains how they placed and routed the circuits using Innovus, making sure they meet certain requirements. They also checked if the design works correctly by simulating it. They demonstrated this process with a 4-bit arithmetic logic unit design [1]

Paper 2:

This paper introduces a method for designing large SFQ circuits. It starts by placing cells in rows on the chip. Then, it groups cells in each row based on their logic level. For clock routing, it suggests a new method called HL-tree, which uses passive transmission lines to distribute the clock. Within each group, it uses splitters and Josephson transmission lines to provide the clock to cells. The paper evaluates this method by designing a 32-bit adder and comparing it with traditional methods. Using this new method reduces the chip area by 27% compared to traditional approaches [2]

Paper 3:

This paper explains how to design and build an 8-bit shift register circuit using Cadence tools. First, the circuit's behavior is described using Verilog code and tested with the Cadence NCLaunch tool. Then, the code is converted into instructions for the computer hardware using the Genus tool. The power usage, physical size, and timing of these instructions are measured. Next, the instructions are used to plan where each part of the circuit will go and how they will connect together. This planning is done using the Cadence Innovus tool with a specific library file [3]

Paper 4:

This paper talks about a special computer program that helps make electronic circuits faster. It's particularly useful for something called an FPGA. This program helps decide where each part of the circuit should go and how they should connect. By using this program, the delays caused by different

parts of the circuit can be minimized. The program takes in a list of parts and how they connect and figures out the best places for them to be. This helps save money and power and makes the circuit work better [4]

Paper 5:

This paper talks about a clever method called the primal-dual method. It's really useful for solving tough problems quickly, especially in computer science. Originally used for exact solutions in areas like matching and network flow, it's now used for approximations too. The paper explains how this method can be adapted for online algorithms, which deal with problems happening in real-time. It shows how this method can be applied to various [5]

Design Methodology:

Procedure to Implement FSM for Stack:

Implementing a stack in Verilog provides an efficient way to manage data in a Last in First Out manner, useful for various applications in digital design and computer architecture.

Memory Creation:

- Create a normal memory in Verilog.

Push Operation:

- Upon receiving data and a push signal, write to the memory starting from the first address.

Pop Operation:

- When a pop signal is received, read from the memory from the last written address.

Empty and Full Detection:

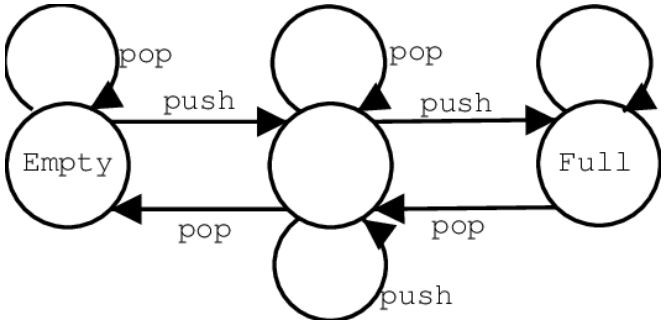
- Assert an empty signal when the stack becomes empty.
- Assert a full signal when the stack becomes full.

States:

Idle State (S0): Initial state where no operation is being performed.

Push State (S1): State for pushing data onto the stack.

Pop State (S2): State for popping data from the stack



Implementation Details:

- Use a single variable `index` to control the stack, as we are writing to and reading from the same address.

- During a push operation, write to the memory and increment the `index`.

- During a pop operation, read from the memory at the last written memory address and decrease the `index`.

- Empty condition occurs when `index` equals 0.

- Full condition occurs when `index` equals the depth of the stack.

Verilog Code:

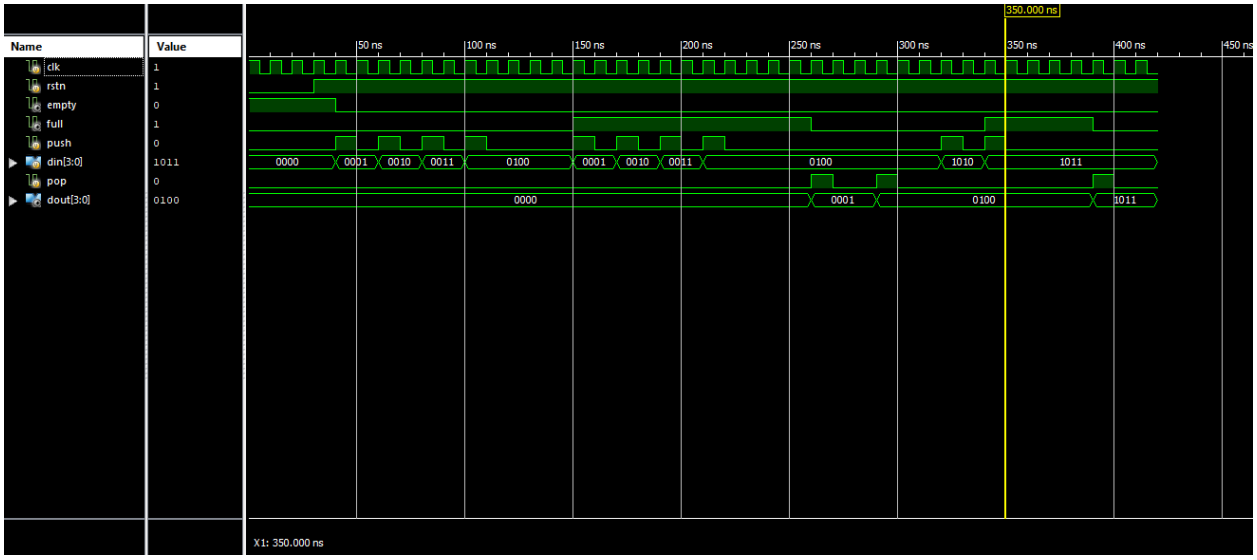
```
`timescale 1ns / 1ps
module Stack(
    input clk,
    input rstn,
    input pop,
    input push,
    input [WIDTH-1:0] din,
    output reg [WIDTH-1:0] dout,
    output empty,
    output full
);
parameter WIDTH = 4;
parameter DEPTH = 5;
reg [WIDTH-1:0] stack [DEPTH-1:0]; // Memory
reg [WIDTH-1:0] index, next_index;
reg [WIDTH-1:0] next_dout;
wire empty, full;
always @ (posedge clk) begin
    if (!rstn) begin
        dout <= 4'd0;
        index <= 1'b0;
    end else begin
        dout <= next_dout;
        index <= next_index;
    end
end

assign empty = (index == 0);
assign full = (index == DEPTH);
always @ (*) begin
    if (push && !full) begin // Write
        stack[index] <= din;
        next_index = index + 1'b1;
    end else if (pop && !empty) begin // Read
        next_dout = stack[index-1'b1];
        next_index = index - 1'b1;
    end else begin
        next_dout = dout;
        next_index = index;
    end
end
endmodule
```

Testbench:

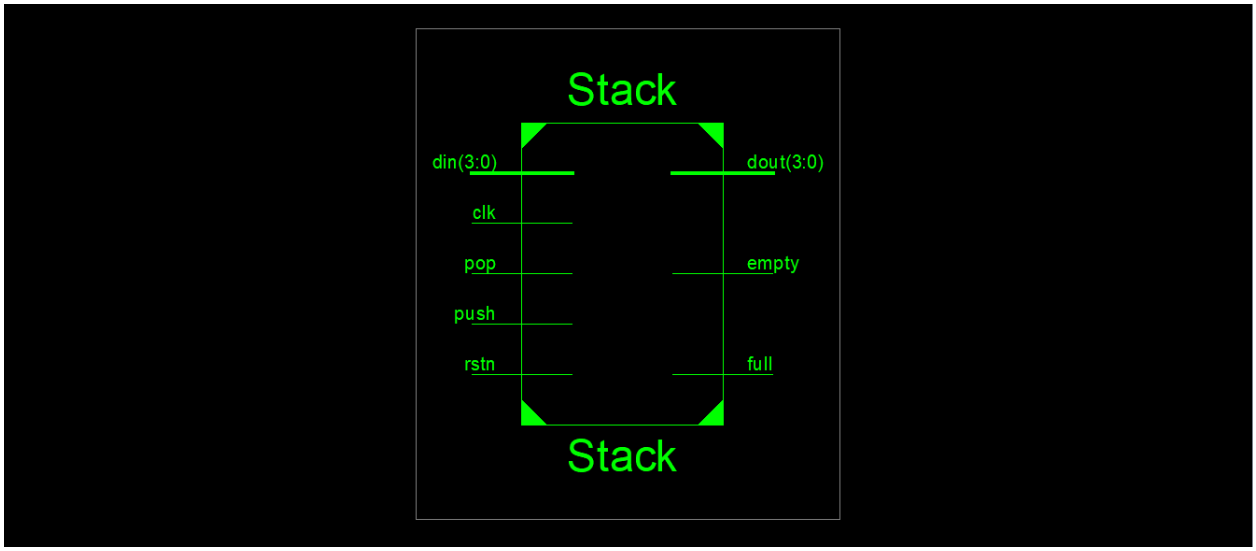
```
module Stack_tb;
 // Inputs
 reg clk;
 reg rstn;
 reg pop;
 reg push;
 reg [3:0] din;
 // Outputs
 wire empty;
 wire full;
 wire [3:0] dout;
 // Instantiate the Unit Under Test (UUT)
 Stack uut (
  .clk(clk),
  .rstn(rstn),
  .pop(pop),
  .push(push),
  .empty(empty),
  .full(full),
  .din(din),
  .dout(dout)
 );
 always #5 clk = ~clk;
 task reset();  //reset task
 begin
 clk = 1'b1;
 rstn = 1'b0;
 pop = 1'b0;
 push = 1'b0;
 din = 4'd0;
 #30 rstn = 1'b1;
 end
 endtask
 task read_stack();  //read task
 begin
 pop = 1'b1;
 #10
 pop = 1'b0;
 end
 endtask
 task write_stack([3:0]din_tb); //write task
 begin
 push = 1'b1;
 din = din_tb;
 #10 push = 1'b0;
 end
 endtask
  // Main code
 initial
 begin
 reset();
 #10;
 repeat(2)
 begin
 write_stack(4'h1);
 #10;
 write_stack(4'h2);
 #10;
 write_stack(4'h3);
 #10;
```

```
write_stack(4'h4);
#40;
end
read_stack();
#20;
read_stack();
#20;
write_stack(4'hA);
#10;
write_stack(4'hB);
#40;
read_stack();
#20;
$finish;
end
endmodule
```

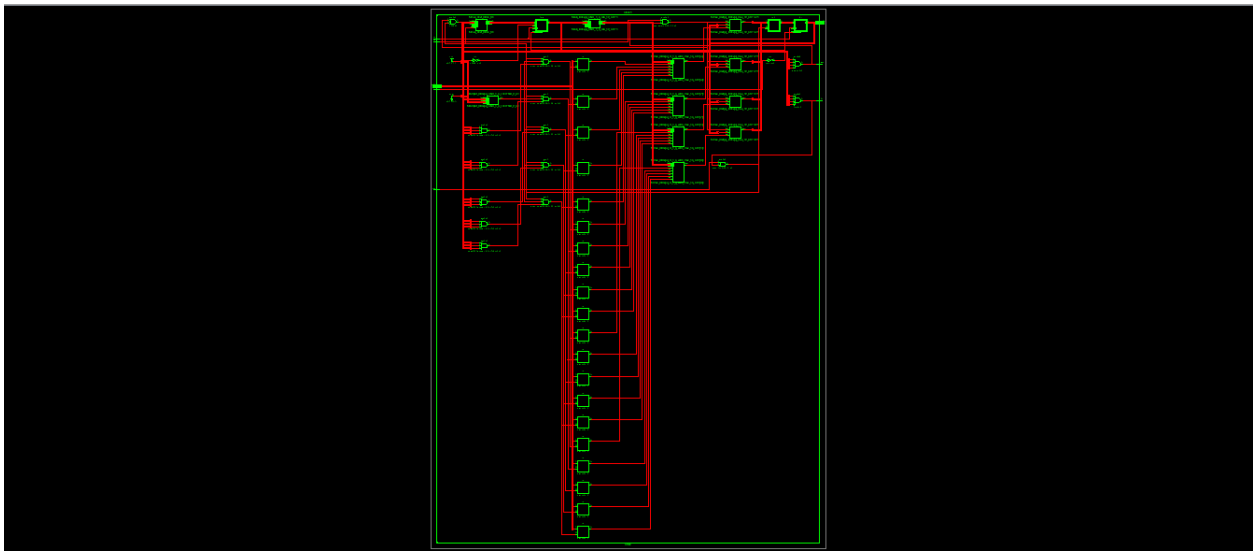
Simulation Result:



Stack (RTL Outside Block)



Stack (RTL Inside Block)



Conclusion:

The conclusion summarizes the development of an FPGA-based stack controller using a Finite State Machine (FSM) design. It outlines the design overview, Verilog implementation, benefits of FPGA-based implementation, and potential future enhancements. The conclusion emphasizes the efficiency and versatility of the stack controller in managing stack operations within digital systems, serving as a foundation for more complex FPGA-based systems requiring stack management functionalities.

References:

[1]Sagnik Nath, Alexander Derrickson “An Automatic Placement and Routing Methodology for Asynchronous SFQ Circuit Design” 1051-8223 © 2019 IEEE

[2] S. N. Shahsavani, T . R. Lin, A. Shafaei, C. J. Fourie, and M. Pedram, “An integrated row-based cell placement and interconnect synthesis tool for large SFQ logic circuits,” IEEE Trans. Appl. Supercond., vol. 27, no. 4, Jun. 2017, Art. No. 1302008.

[3]“Innovus implementation system.” Cadence. [Online]. Available: cadence.com/content/cadence-www/global/en_US/home/training/allcourses/86142.html, Accessed: May 20, 2019.

[4]aishali Udar and Sanjeev Sharma “Analysis of Place and Route Algorithm for Field Programmable Gate Array (FPGA)” 978 -1-4673-5758- 6/13© 2013 IEEE

[5]problems like caching, routing, and balancing loads. It also demonstrates that it can solve classic online problems efficiently.