***Non-FSM:***

**Module**

`timescale 1ns / 1ps

module FIFO(

clk,

reset,

data\_in,

write\_to\_stack,

read\_from\_stack,

data\_out,

stack\_full,

stack\_empty,

ptr

);

input [7:0] data\_in;

input clk,

reset,

write\_to\_stack,

read\_from\_stack;

output reg [7:0] data\_out;

output reg stack\_full = 0,

stack\_empty = 1;

output reg [4:0] ptr = 0;

reg [7:0] queue [31:0];

always@(\*) begin //generates stack empty and full signals

if (ptr == 0) begin

stack\_empty = 1;

end

else if (ptr == 31) begin

stack\_full = 1;

end

else begin

stack\_empty = 0;

stack\_full = 0;

end

end

always@(\*) begin

if (reset == 1) begin

ptr = 0;

end

end

always@(posedge clk) begin

if (write\_to\_stack == 1 & stack\_full == 0) begin

queue[ptr] = data\_in;

ptr = ptr + 1;

end

else if (read\_from\_stack == 1 & stack\_empty == 0) begin

data\_out = queue[0];

for (integer i = 0; i < ptr; i = i + 1) begin

queue[i] = queue[i+1];

end

ptr = ptr - 1;

end

end

endmodule

***Test bench 1:***

Below we can see that all operations are done perfectly and pointer value (ptr) is also changing as expected for the queue. A clock cycle is of 10 ns. First we write and extract from the queue hence the pointer value changed from 0 -> 1 (enque) and then 1 -> 0 (deque.) We are then enqueing 4 times and dequeing 3 times. Hence last ptr value is 1.



***Test bench 2:***

Below we can see the working functionality of Stack\_empty and Stack\_full signal. Initially the stack\_empty signal is high. But as soon as we enque all 32 values the stack\_full signal is high and it accepts no more values even though we attempt to write 2 more values into the queue.

`timescale 1ns / 1ps

module FIFO\_tb;

// Signals

reg clk;

reg reset;

reg [7:0] data\_in;

reg write\_to\_stack;

reg read\_from\_stack;

wire [7:0] data\_out;

wire stack\_full;

wire stack\_empty;

wire [4:0] ptr;

// Instantiate the FIFO module

FIFO fifo\_inst (

.clk(clk),

.reset(reset),

.data\_in(data\_in),

.write\_to\_stack(write\_to\_stack),

.read\_from\_stack(read\_from\_stack),

.data\_out(data\_out),

.stack\_full(stack\_full),

.stack\_empty(stack\_empty),

.ptr(ptr)

);

// Clock generation

always #5 clk = ~clk;

// Initial values

initial begin

clk = 1;

reset = 1;

write\_to\_stack = 0;

read\_from\_stack = 0;

data\_in = 8'h00;

// Reset

#10;

reset = 0;

#10;

for (integer i = 1; i <= 34; i = i + 1) begin

// Write data\_in to FIFO

write\_to\_stack = 1;

data\_in = i;

#10;

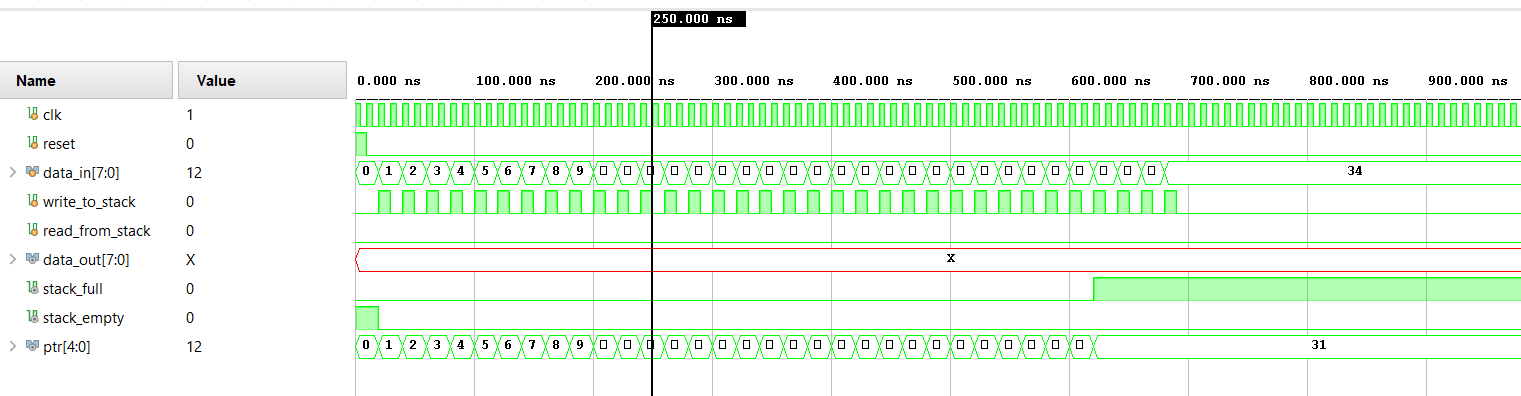
write\_to\_stack = 0;

#10;

end

end

endmodule



***FSM:***

**Module**

`timescale 1ns / 1ps

module FIFO\_FSM(

clk,

reset,

data\_in,

write\_to\_stack,

read\_from\_stack,

data\_out,

stack\_full,

stack\_empty,

ptr

);

input [7:0] data\_in;

input clk,

reset,

write\_to\_stack,

read\_from\_stack;

output reg [7:0] data\_out;

output reg stack\_full = 0,

stack\_empty = 1;

output reg [4:0] ptr = 0;

reg [7:0] queue [31:0];

parameter IDLE = 2'b00;

parameter ENQUE = 2'b10;

parameter DEQUE = 2'b11;

reg [1:0] state;

integer i;

always @(posedge clk or posedge reset) begin

if (reset) begin

ptr = 0;

stack\_empty = 1;

stack\_full = 0;

state <= IDLE;

end

else begin

case(state)

IDLE: begin

if (write\_to\_stack == 1 & stack\_full == 0) begin

state <= ENQUE;

end

else if (read\_from\_stack == 1 & stack\_empty == 0) begin

state <= DEQUE;

end

else begin

state <= IDLE;

end

if (ptr == 0) begin

stack\_empty = 1;

end

else if (ptr == 31) begin

stack\_full = 1;

end

else begin

stack\_empty = 0;

stack\_full = 0;

end

end

ENQUE: begin

queue[ptr] = data\_in;

ptr = ptr + 1;

state <= IDLE;

end

DEQUE: begin

data\_out = queue[0];

for ( i = 0; i < ptr; i = i + 1) begin

queue[i] = queue[i+1];

end

ptr = ptr - 1;

state <= IDLE;

end

endcase

end

end

endmodule

**TestBench**

`timescale 1ns / 1ps

module FIFO\_tb;

// Signals

reg clk;

reg reset;

reg [7:0] data\_in;

reg write\_to\_stack;

reg read\_from\_stack;

wire [7:0] data\_out;

wire stack\_full;

wire stack\_empty;

wire [4:0] ptr;

FIFO\_FSM fifo\_inst1 (

.clk(clk),

.reset(reset),

.data\_in(data\_in),

.write\_to\_stack(write\_to\_stack),

.read\_from\_stack(read\_from\_stack),

.data\_out(data\_out),

.stack\_full(stack\_full),

.stack\_empty(stack\_empty),

.ptr(ptr)

);

// Clock generation

always #5 clk = ~clk;

initial begin

clk = 1;

// Reset sequence

reset = 1;

#10;

reset = 0;

#10;

// Enqueue data

data\_in = 8'hAA; // Sample data

write\_to\_stack = 1;

#10;

write\_to\_stack = 0;

#10;

// Dequeue data

read\_from\_stack = 1;

#10;

read\_from\_stack = 0;

#10;

// Another Enqueue

data\_in = 8'hBB; // Another sample data

write\_to\_stack = 1;

#10;

write\_to\_stack = 0;

#10;

// Dequeue data again

read\_from\_stack = 1;

#10;

read\_from\_stack = 0;

#10;

// Enqueue more data to fill the FIFO

repeat (30) begin

data\_in = $random;

write\_to\_stack = 1;

#10;

write\_to\_stack = 0;

#10;

end

// Dequeue all data

repeat (32) begin

read\_from\_stack = 1;

#10;

read\_from\_stack = 0;

#10;

end

end

endmodule

**Results**

We can see that the functionality of our queue is working as expected. But there is a clock cycle delay because our module is working on clock’s positive edge and reset only. Therefore any change to the state will be implemented on next positive edge of the clock and hence the output will be delayed.

