*Digital System Design Lab*

# CEL-442



*Class: BCE-6(A).*

### Enrollment no: 01-132182-024

**Lab 15**

**Title:Implementation of lifo and fifo in one single module**

**Background:**

                               There are two ways to work with a queue. You can use it as a general array (and use all the operaions that you can perform on an array). The other way is to use the built-in methods that a queue provide.

**Introduction:**

* A Queue is a variable size ordered collection of similar objects. There are two main aspects of a queue that makes it attractive for verification purposes.
* To implement FIFO in Verilog **imagine the memory components to be arranged in a circular queue fashion with two pointers; write and read**. The write pointer points to the start of the circle whereas the read pointer points to the end of the circle. Both these pointers increment themselves by one after each read or write operation.
* Stack or LIFO Verilog Code The Last In First Out (LIFO) or Stack is a data arrangement structure in which the data that enters the last is the one that is removed first. Let us see how to implement the concept of Stack using Verilog. For a normal memory, we provide the address for reads and writes.
* First, a queue can have variable length, including a length of zero. This makes a queue an ideal candidate as a storage element that can shrink or grow as elements are deleted or added to it without fixing an artificial upper limit on its size as a regular fixed size array.
* The other advantage of having a queue is that it provides a way to emulate both Last In First Out (LIFO) and First In First Out (FIFO) behavior that are required in so many ordered transactions. At the same time, as we will see in the next section, a queue still allows you to access any element randomly within the queue without any overhead just as a regular array.

**Procedure:**

1. In low level programming, we mostly manipulate data that is present in registers.
2. Registers are units of memory which are closely coupled with the processor core (ALU) and in most architectures the ALU operations are carried out on these registers and the results obtained are also stored in them.
3. Registers are a limited resource and efficient management of registers is a key issue in low level programming.
4. Processors must carry out a large number of functions utilizing the limited number of registers available.
5. In order to do so, the registers are used to store the data values that are required by the current function being executed on the processor and not all the data values of all the functions which are currently in inactive states.
6. Instead the data values of inactive functions are stored in a special area of memory known as the stack

**Code:**

/////////check=1 then LIFO & check=0 then FIFO///////////////

timescale 1ns / 1ps

module LIFO\_FIFO(clk,DATA\_IN,delete,error,lifo\_in,lifo\_out,mem0,mem1,mem2,mem3,check);

input clk,DATA\_IN,delete,lifo\_in,check;

output lifo\_out,error;

reg lifo\_out,error;

output mem0,mem1,mem2,mem3;

reg mem0,mem1,mem2,mem3;

reg [2:0]count;

reg r0,r1,r2,r3;

initial

begin

count = 3'b000;

end

always @(posedge clk)

begin

if(DATA\_IN == 1'b1 && delete == 1'b1)

begin error = 1'b1; end

if(DATA\_IN == 1'b1 && check==1'b0)

begin

if(count == 3'b000 || count == 3'b001 || count == 3'b010 || count == 3'b011)

begin

r3 = r2;

r2 = r1;

r1 = r0;

r0 = lifo\_in;

lifo\_out = r0;

count = count + 1'b1;

end

else

error = 1'b1;

end

if(delete == 1'b1)

begin

if(count == 3'b100)

begin

lifo\_out = r3;

r3 = 1'b0;

count = count - 1;

end

else if( count == 3'b011)

begin

lifo\_out = r2;

r2 = 1'b0;

count = count - 1;

end

else if( count == 3'b010)

begin

lifo\_out = r1;

r1 = 1'b0;

count = count - 1;

end

else if( count == 3'b001)

begin

lifo\_out = r0;

r0 = 1'b0;

count = count - 1;

end

else

error = 1'b1;

end

mem0 = r0;

mem1 = r1;

mem2 = r2;

mem3 = r3;

//////////////////////////////////////////////

if(DATA\_IN == 1'b1 && check==1'b1)

begin

if(count == 3'b000 || count == 3'b001 || count == 3'b010 || count == 3'b011)

begin

r3 = r2;

r2 = r1;

r1 = r0;

r0 = lifo\_in;

lifo\_out = r3;

count = count + 1'b1;

end

else

error = 1'b1;

end

if(delete == 1'b1)

begin

if(count == 3'b100)

begin

lifo\_out = r3;

r3 = 1'b0;

count = count - 1;

end

else if( count == 3'b011)

begin

lifo\_out = r2;

r2 = 1'b0;

count = count - 1;

end

else if( count == 3'b010)

begin

lifo\_out = r1;

r1 = 1'b0;

count = count - 1;

end

else if( count == 3'b001)

begin

lifo\_out = r0;

r0 = 1'b0;

count = count - 1;

end

else

error = 1'b1;

end

mem0 = r0;

mem1 = r1;

mem2 = r2;

mem3 = r3;

end

endmodule

**Test Bench:**

`timescale 1ns / 1ps

module LIFO\_FIFO\_test;

// Inputs

reg clk;

reg DATA\_IN;

reg delete;

reg lifo\_in;

reg check;

// Outputs

wire error;

wire lifo\_out;

wire mem0;

wire mem1;

wire mem2;

wire mem3;

// Instantiate the Unit Under Test (UUT)

LIFO\_FIFO uut (

.clk(clk),

.DATA\_IN(DATA\_IN),

.delete(delete),

.error(error),

.lifo\_in(lifo\_in),

.lifo\_out(lifo\_out),

.mem0(mem0),

.mem1(mem1),

.mem2(mem2),

.mem3(mem3),

.check(check)

);

initial begin

clk = 0;

forever #5 clk = ~clk;

end

initial begin

// Initialize Inputs

//clk = 1;

DATA\_IN = 1;

delete = 0;

lifo\_in = 1;

check = 1;

// Wait 100 ns for global reset to finish

#50;

//Again

//clk = 1;

DATA\_IN = 1;

delete = 0;

lifo\_in = 0;

check = 1;

// Wait 100 ns for global reset to finish

#50;

// Initialize Inputs

//clk = 1;

DATA\_IN = 1;

delete = 0;

lifo\_in = 1;

check = 1;

// Wait 100 ns for global reset to finish

#50;

//Again

//clk = 1;

DATA\_IN = 1;

delete = 0;

lifo\_in = 0;

check = 1;

// Wait 100 ns for global reset to finish

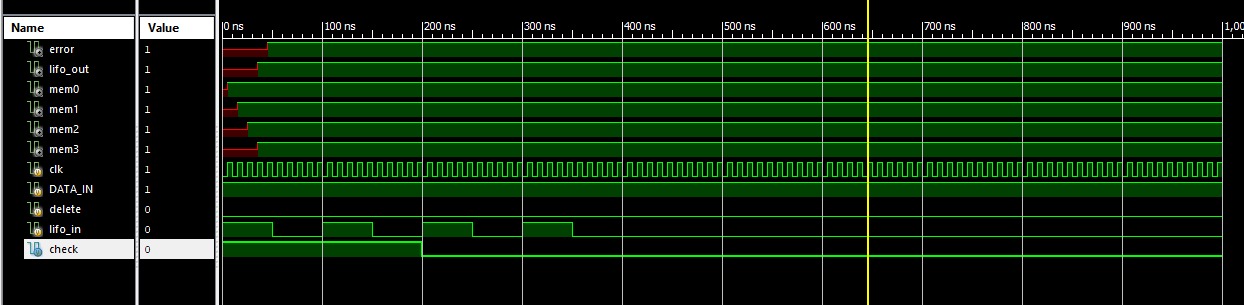
#50;

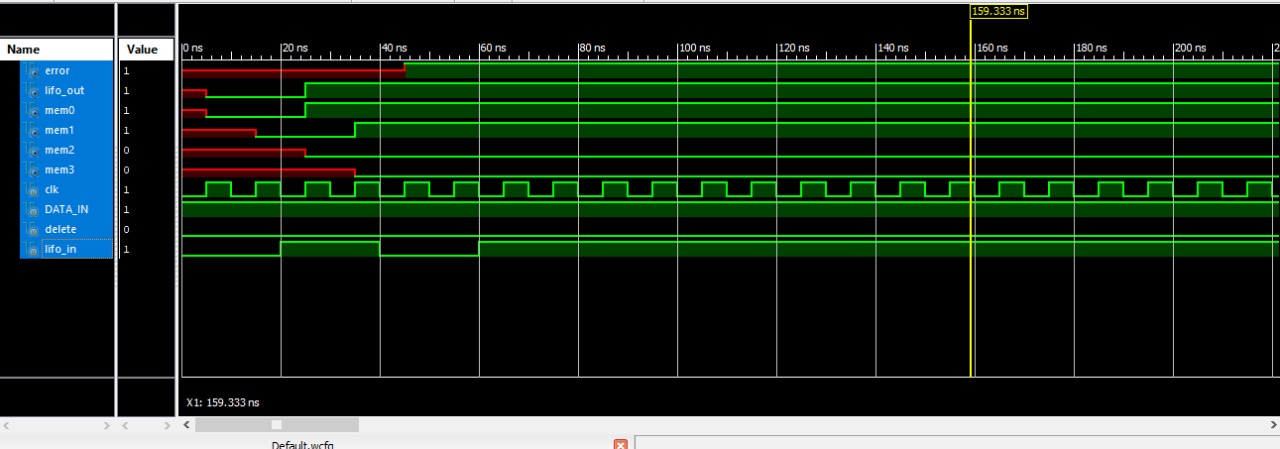
// Add stimulus here

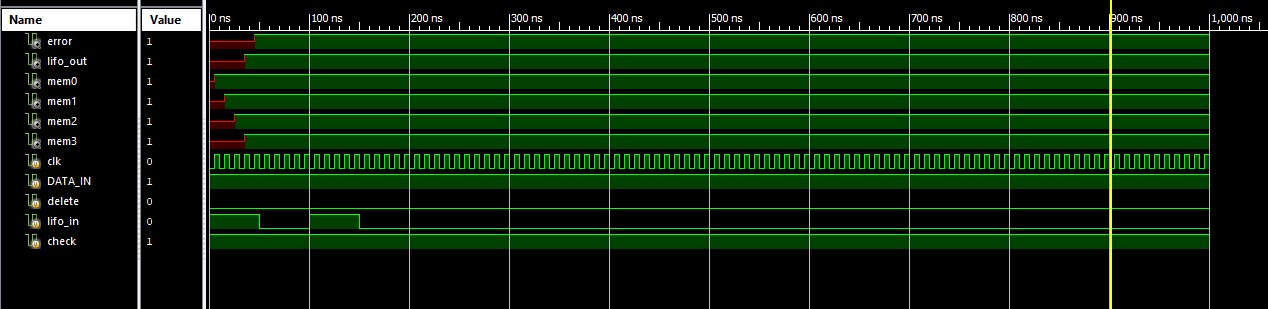
end

endmodule

OUTPUT:







Conclusion:

Today I implemented example#10.5 in book as my lab task in which my primarily objective was to design a module which works both as a LIFO AND FIFO depending upon the user input given at a specific time