VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Logic Design with HDL (CO1025) - Semester 212

Assignment

FIFO
First In First Out

HO CHI MINH CITY, MAY 2022

Contents

1 Introduction

1.1 Introduction of the topic

FIFO is an acronym for First In First Out, which describes how data is managed relative to time or priority. In this case, the first data that arrives will also be the first data to leave from a group of data. A FIFO Buffer is a read/write memory array that automatically keep track of the order in which data enters into the module and reads the data out in the same order. In hardware FIFO buffer is used for synchronization purposes. It is often implemented as a circular queue, and has two pointers:

- Read Pointer/Read Address Register
- Write Pointer/Write Address Register

Read and write addresses are initially both at the first memory location and the FIFO queue is Empty. When the difference between the read address and write address of the FIFO buffer is equal to the size of the memory array then the FIFO queue is Full.

FIFO can be classified as synchronous or asynchronous depending on whether same clock (synchronous) or different clocks (asynchronous) control the read and write operations.

A synchronous FIFO refers to a FIFO design where data values are written sequentially into a memory array using a clock signal, and the data values are read out sequentially from the memory array using the same clock signal. Figure 1 shows the flow of the operation of a typical FIFO.

1.2 Scope

Circular buffers are popular constructs for creating queues in sequential programming languages, but they can also be implemented in hardware. In this article, we will create a ring buffer in VHDL to implement a FIFO

A ring buffer is a FIFO implementation that uses contiguous memory for storing the buffered data with a minimum of data shuffling. New elements stay at the same memory location from the time of writing until it is read and removed from the FIFO.

Two counters are used to keep track of the location and the number of elements in the FIFO. These counters refer to an offset from the start of the memory space where the data is stored. In VHDL, this will be an index to an array cell. For the rest of this article, we will refer to these counters are pointers.

These two pointers are the head(front) and tail(rear) pointers. The rear always points to the memory slot that will contain the next written data, while the front refers to the next element that will be read from the FIFO. There are other variants, but this is the one we are going to use.

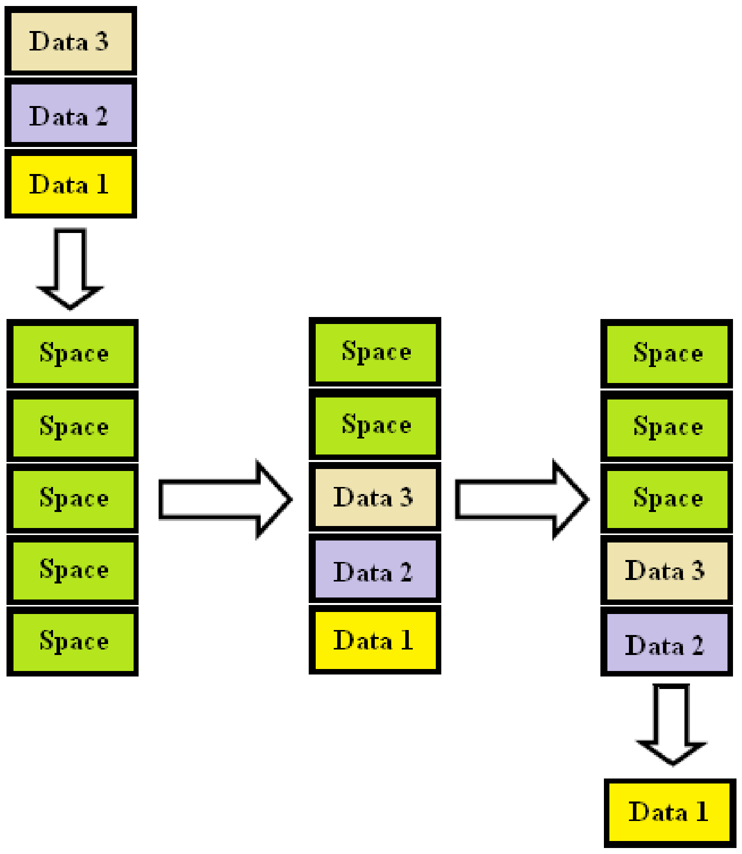


Figure 1: Typical FIFO

1.3 Content Summary

- Understanding theory of FIFO and knowing application of FIFO
- How to know working mechanism of FIFO
- Design diagrams for FIFO
- Implementation of FIFO

2 Backgrounds and applications

2.1 Theory

The FIFO module is a variable-length buffer with scalable register word-width and address space, or depth. There are watermark flags available for “almost_full” conditions. The depth of the “almost_full” flags can be adjusted within the module’s parametrization, or generic block in the case of the verilog version. The FIFO also has flags for empty, full. There is an output port for reading out the data count. As the port name suggest, this tells the world (outside the module) how many words are currently stored between the read and write pointers within the RAM.

The Software required/used for this design:

Vivado Design Suite is a software suite produced by Xilinx for synthesis and analysis of hardware description language (HDL) designs

2.2 Applications of the circuit

Building the Circuit The fifo.v can be configured by changing the values within the parameterization and generic parameters respectively within each module. The output word width can be scaled as can the FIFO address space. The watermark flags can be set to trigger at various depths by adjusting the almst_full parameter values.

The design has a data input port, a clock, active low reset, read enable, write enable, data output port, data count port, empty, full, almost full. The VHDL generated black box model with inputs and outputs can be seen below in figure 2.

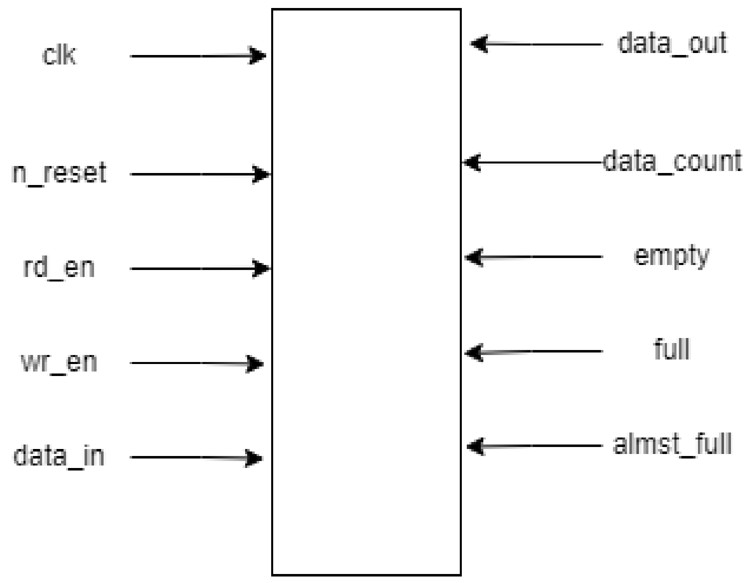


Figure 2: Black box of FIFO

3 Design

3.1 The idea

A first in first out (FIFO) queue is a queue of data such that the data that is written into it first, is read from it first.

The pointer block provides read and write pointers according to read and write operations into the queue.

The count block keeps track of the number of data in the queue and issues empty and full flags.

These flags are generated by combinational blocks using the present count of data as input.

The read block uses the read pointer(frontAddr) to read from the queue memory, and the write block uses the write pointer(rearAddr) to write into this memory.

3.2 Block diagram

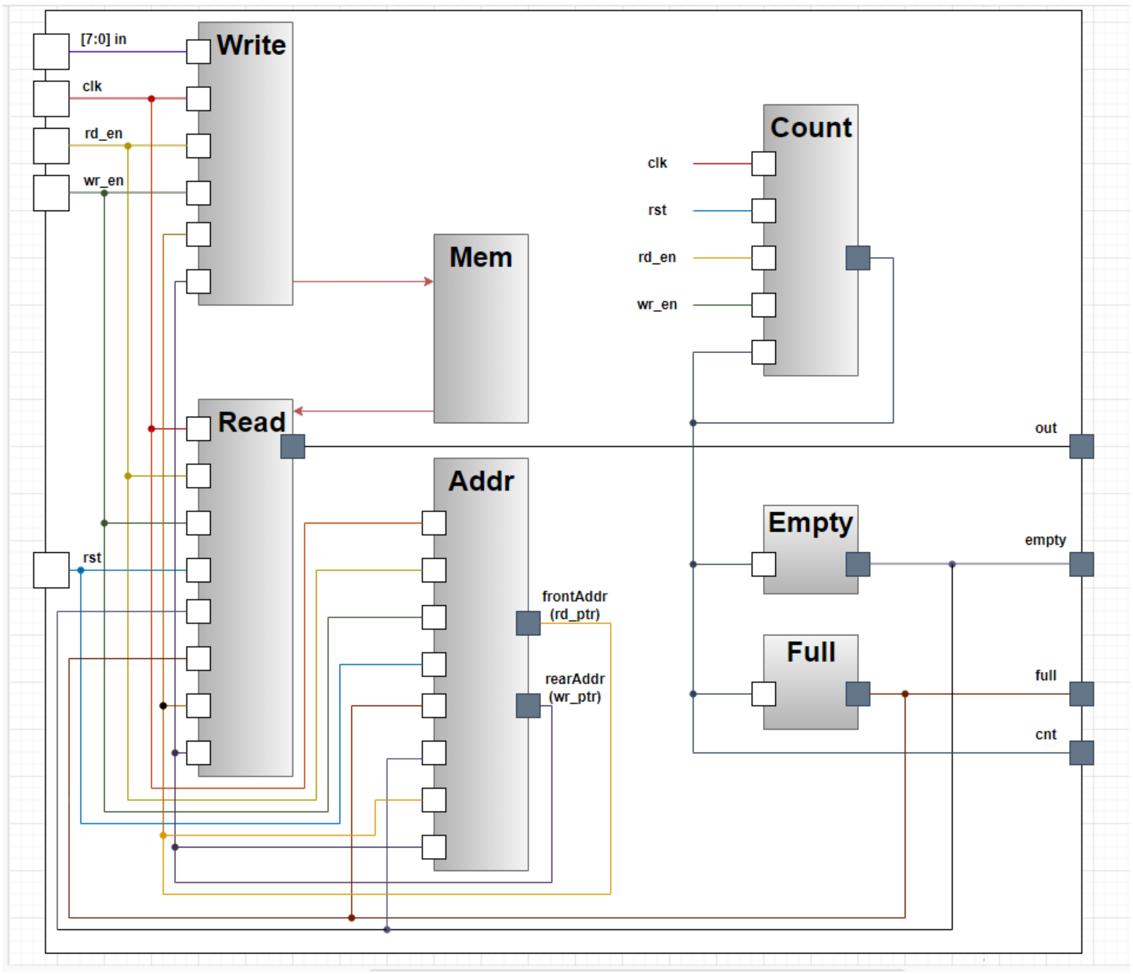


Figure 3: Block diagram of FIFO

3.3 Flowchart

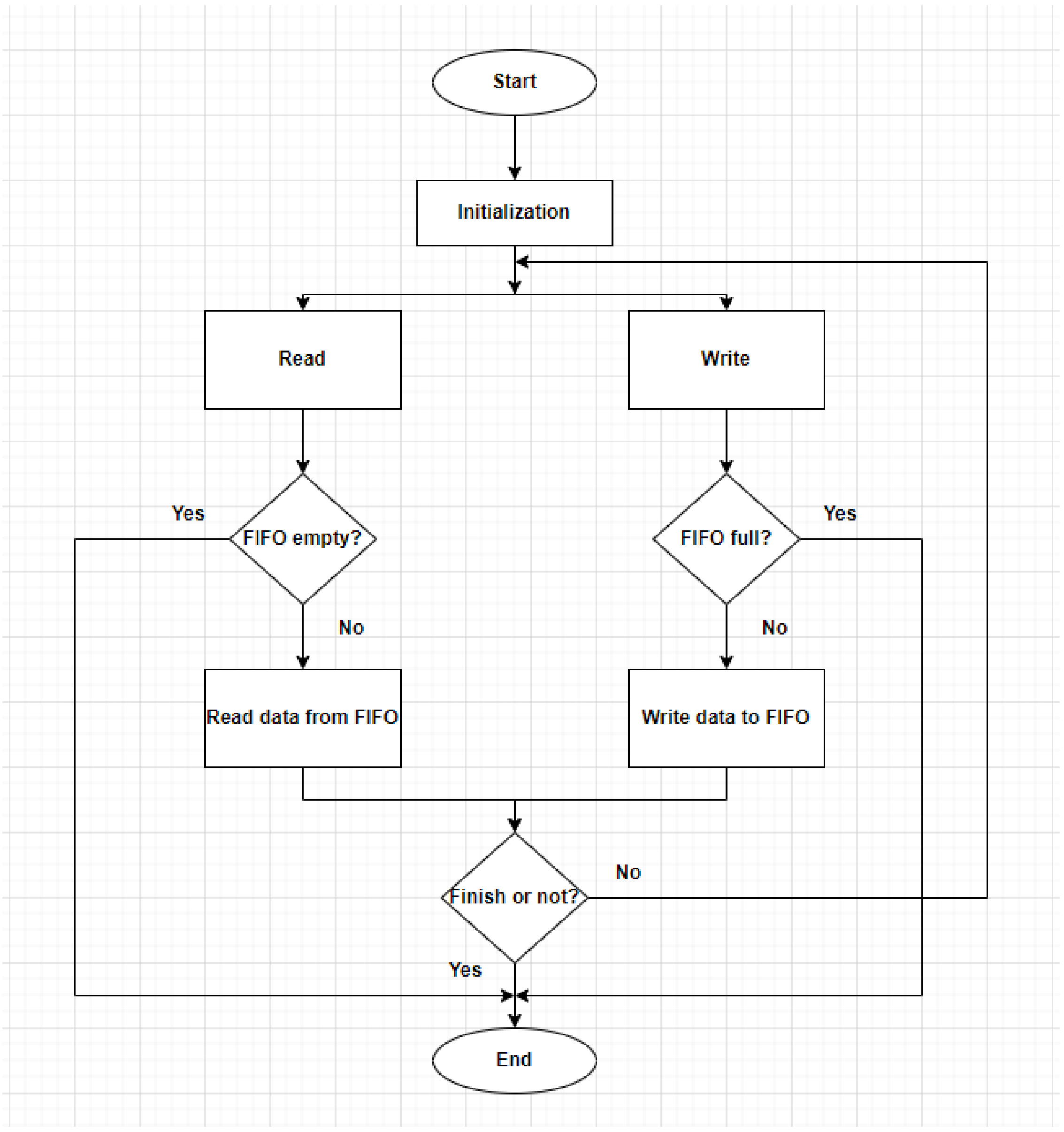
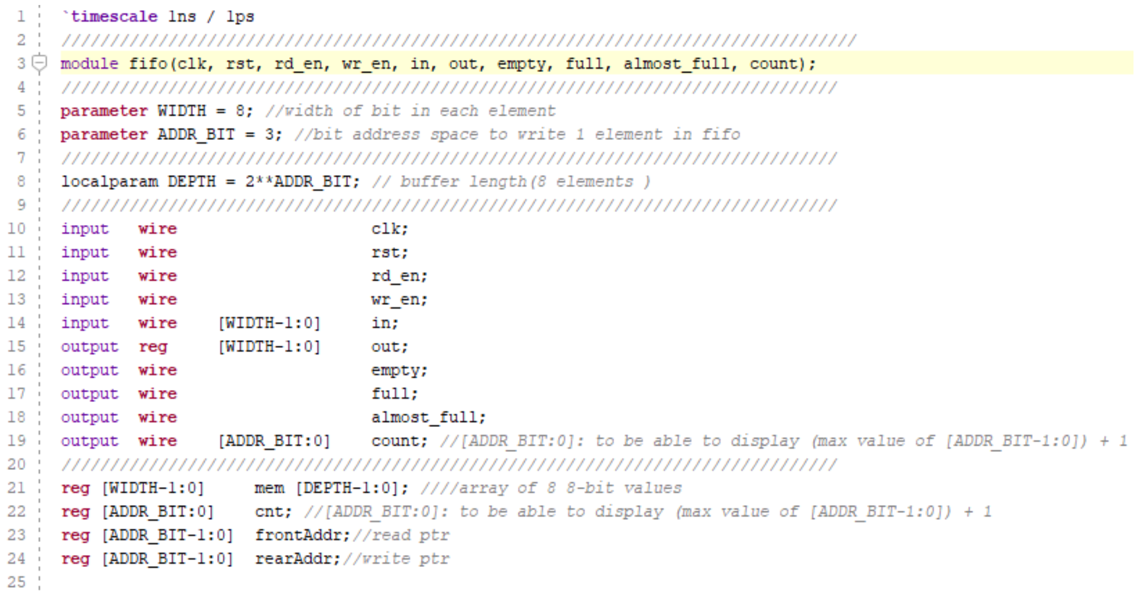


Figure 4: Flowchart of FIFO

4 Implementation

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(rearAddr) points to the start of the circle whereas the read pointer(frontAddr) points to the end of the circle. Both these pointers increment themselves by one after each read or write operation. This buffer will also consist of two flags; empty and full. These will help in indicating when the FIFO is full (cannot be written to) or empty (cannot be read from). This circular implementation can be seen from the following figure:

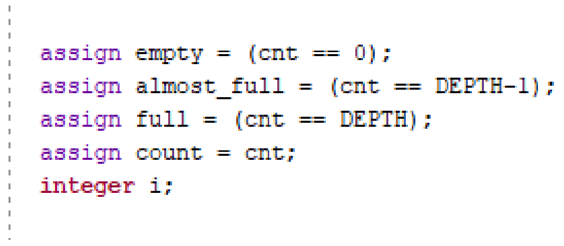


```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
module fifo(clk, rst, rd_en, wr_en, in, out, empty, full, almost_full, count);
////////////////////////////////////////////////////////////////////////////////
parameter WIDTH = 8; //width of bit in each element
parameter ADDR_BIT = 3; //bit address space to write 1 element in fifo
////////////////////////////////////////////////////////////////////////////////
localparam DEPTH = 2**ADDR_BIT; // buffer length(8 elements )
////////////////////////////////////////////////////////////////////////////////
input   wire                   clk;
input   wire                   rst;
input   wire                   rd_en;
input   wire                   wr_en;
input   wire    [WIDTH-1:0]    in;
output  reg     [WIDTH-1:0]    out;
output  wire                   empty;
output  wire                   full;
output  wire                   almost_full;
output  wire    [ADDR_BIT:0]   count; //[ADDR_BIT:0]: to be able to display (max value of [ADDR_BIT-1:0]) + 1
////////////////////////////////////////////////////////////////////////////////
reg [WIDTH-1:0]     mem [DEPTH-1:0]; ////array of 8 8-bit values
reg [ADDR_BIT:0]    cnt; //[ADDR_BIT:0]: to be able to display (max value of [ADDR_BIT-1:0]) + 1
reg [ADDR_BIT-1:0]  frontAddr;//read ptr
reg [ADDR_BIT-1:0]  rearAddr;//write ptr
```

Figure 5: The declarative region

The size of the FIFO buffer greatly depends on the number of address bits. As the number of words in fifo = 2ˆ(number of address bits). The FIFO I will be coding here will consist of 8 memory elements (8 bits in each memory element and 3 address bits). This can be easily changed by changing the parameters in the code, by doing so you can create a buffer of any size.

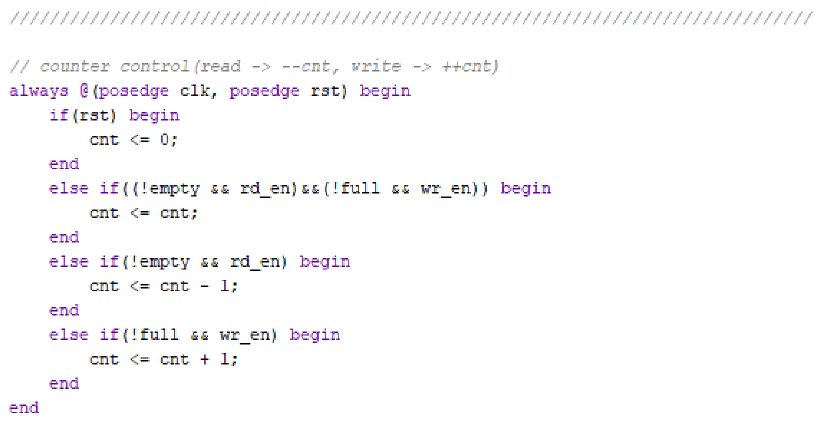
The block of concurrent statements is where we assign the output flags, signaling the full, empty, almost_full state of the ring buffer. We are basing the calculations on the DEPTH generic and on the count signal. The depth is a constant that won't be changing. Therefore, the flags will change only as a result of an updated count.



```
assign empty = (cnt == 0);
assign almost_full = (cnt == DEPTH-1);
assign full = (cnt == DEPTH);
assign count = cnt;
integer i;
```

Figure 6: The concurrent statements

The count signal is used for generating the full and empty signals, which in turn are used for preventing overwrite and over-read of the FIFO. The counter is updated by a combinational process that is sensitive to the front and rear pointer, but those signals are only updated at the rising edge of the clock. Therefore, the fill count will also change immediately after the clock edge.

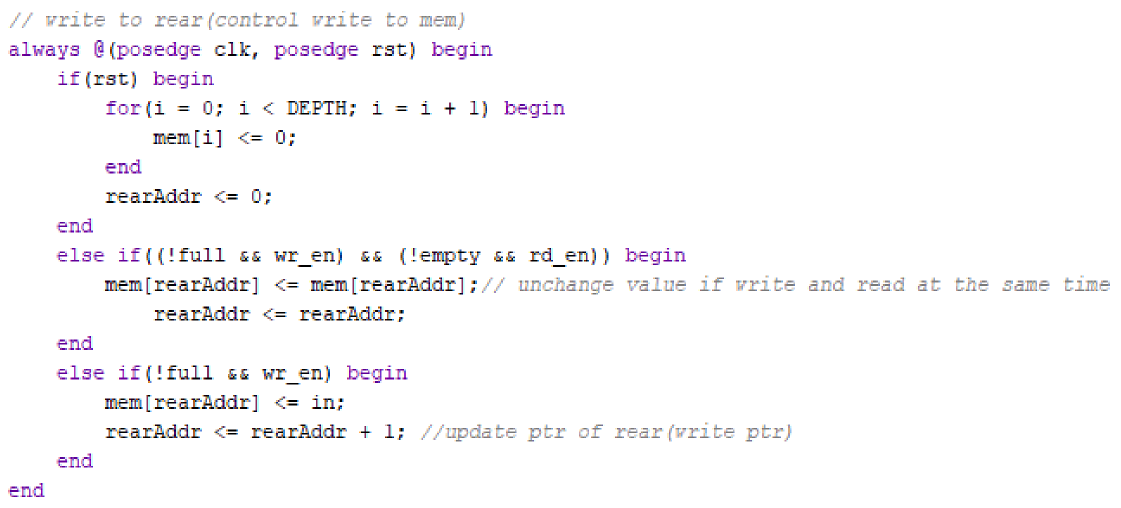


```
//////////////////////////////////////////////////////////////////////////////////

// counter control(read -> --cnt, write -> ++cnt)
always @(posedge clk, posedge rst) begin
    if(rst) begin
        cnt <= 0;
    end
    else if((!empty && rd_en)&&(!full && wr_en)) begin
        cnt <= cnt;
    end
    else if(!empty && rd_en) begin
        cnt <= cnt - 1;
    end
    else if(!full && wr_en) begin
        cnt <= cnt + 1;
    end
end
```

Figure 7: The counter control

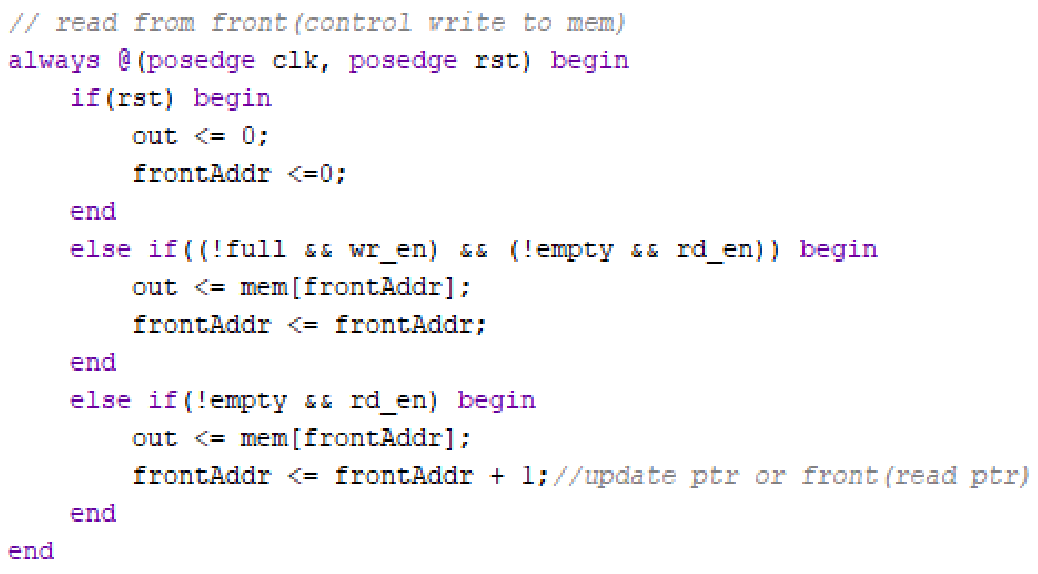
The basic function of the rear pointer is to increment whenever the write enable signal is asserted from the outside of this module. We are doing this by passing the rear signal to the previously mentioned increase procedure.



```
// write to rear(control write to mem)
always @(posedge clk, posedge rst) begin
    if(rst) begin
        for(i = 0; i < DEPTH; i = i + 1) begin
            mem[i] <= 0;
        end
        rearAddr <= 0;
    end
    else if((!full && wr_en) && (!empty && rd_en)) begin
        mem[rearAddr] <= mem[rearAddr];// unchange value if write and read at the same time
            rearAddr <= rearAddr;
    end
    else if(!full && wr_en) begin
        mem[rearAddr] <= in;
        rearAddr <= rearAddr + 1; //update ptr of rear(write ptr)
    end
end
```

Figure 8: The write control

The front pointer is incremented in a similar manner as the rear pointer, but the rd_en input is used as the trigger. Just like with the overwrites, we are protecting against over-reads by including and check empty in the Boolean expression.



```
// read from front(control write to mem)
always @(posedge clk, posedge rst) begin
    if(rst) begin
        out <= 0;
        frontAddr <=0;
    end
    else if((!full && wr_en) && (!empty && rd_en)) begin
        out <= mem[frontAddr];
        frontAddr <= frontAddr;
    end
    else if(!empty && rd_en) begin
        out <= mem[frontAddr];
        frontAddr <= frontAddr + 1;//update ptr or front(read ptr)
    end
end
```

Figure 9: The read control

5 Results

5.1 Waveform

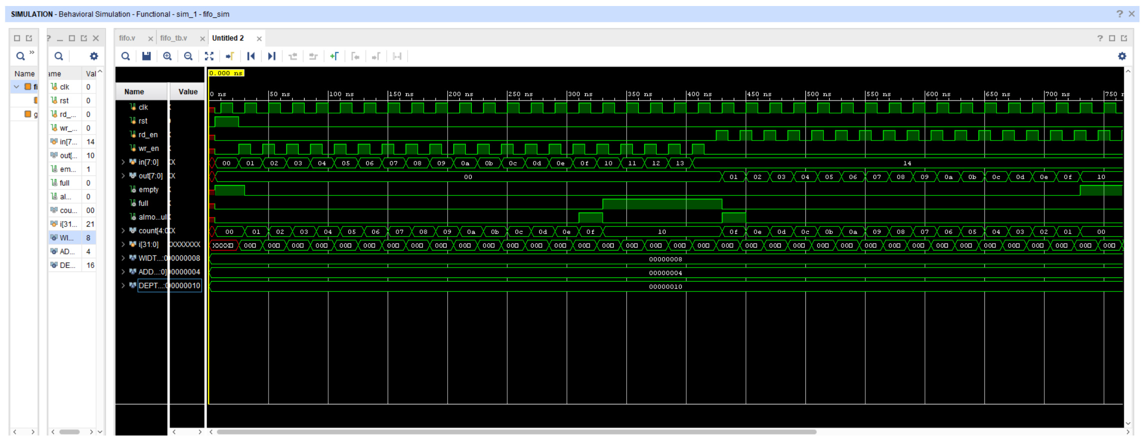


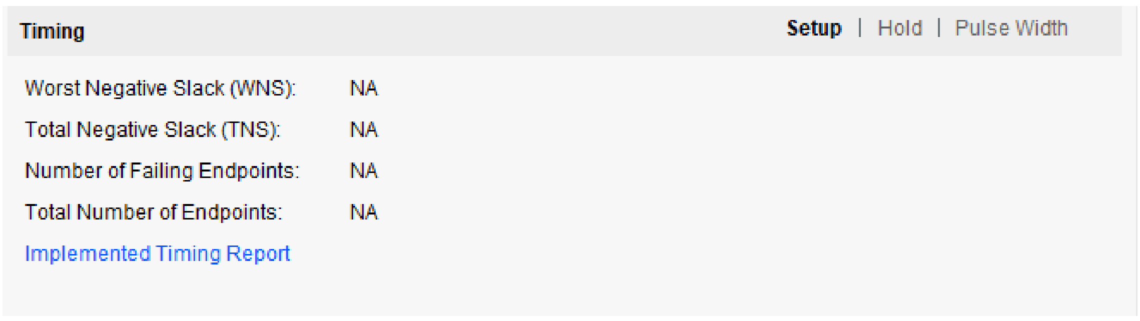
Figure 10: The waveform of FIFO

5.2 Explanations

The FIFO is instantiated in a simple testbench to demonstrate how it works.Even though in fifo.v i initialize ADDR_BIT is 3(DEPTH will be 8) but when we test in uut i set ADDR_BIT is 4 so DEPTH will be 16. I do it because i want to check my function will work well when we change parameter. It is ok for all because this is testbench.

You see that initial state, when rst is high then everyting will be 0 and empty in FIFO will be 1. Then wr_en(write control) is 1 , empty will be low and count will be increased until count is 0f(15 in decimal) so almost_full will be high, it informs you FIFO has almost full. When we write last element, full will be high. After that rd_en(read control) is 1, FIFO read first element and then almost_full is high because count will be decreased by 1 unit(count == DEPTH-1). At the same time full will be 0 and we read elements output will be display to out.

5.3 Timing and resources


Timing — Setup | Hold | Pulse Width

Worst Negative Slack (WNS):	NA
Total Negative Slack (TNS):	NA
Number of Failing Endpoints:	NA
Total Number of Endpoints:	NA

Implemented Timing Report

Figure 11: The timing of FIFO

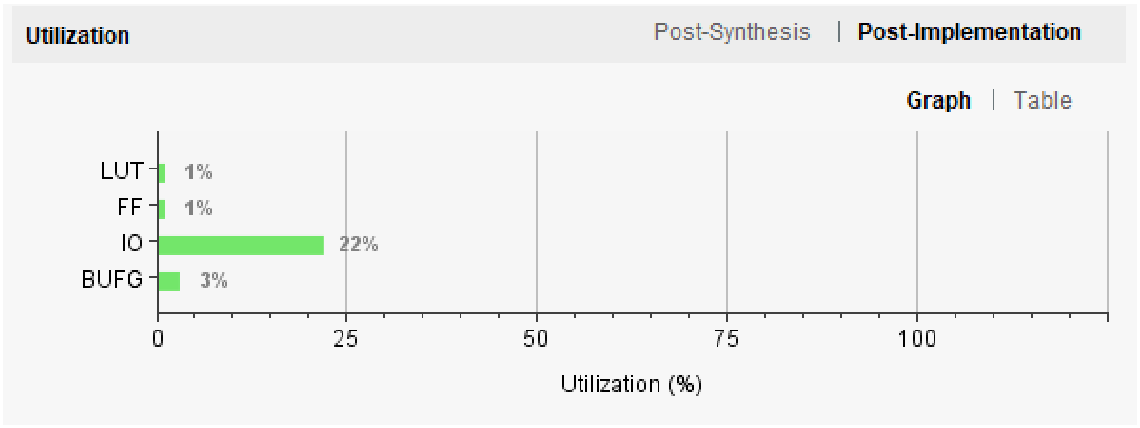
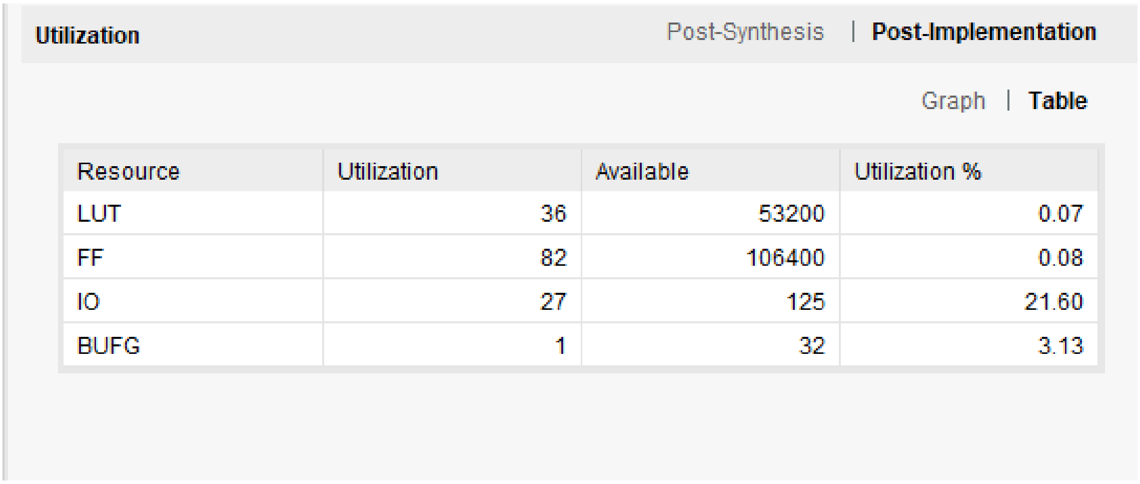


Figure 12: The utilization resource of FIFO


Utilization — Post-Synthesis | Post-Implementation

Graph | Table

Resource	Utilization	Available	Utilization %
LUT	36	53200	0.07
FF	82	106400	0.08
IO	27	125	21.60
BUFG	1	32	3.13

Figure 13: The utilization resource of FIFO

6 Conclusion

6.1 Summary working achievements

This paper deals with the creation of a FIFO module and its verification using UUT test bench environment. The FIFO design were designed using Verilog HDL and integrated by instantiating all the three always blocks.

6.2 Advantages

The test bench environment for this design was easily built by integrating all the verification components which communicate with each other through ports. It tries to verify the correct functionality of data write and data read with proper flag triggers at the expected time. For achieving this, a reference model was written the behavior of the Unit under Test (UUT). Comparison with the UUT outputs and the reference model outputs tells if the UUT functions correctly or not.A covergroup was written in the sequencer part of the test bench to check if all the possible cases of inputs are achieved and reports coverage judging the efficiency of the test

6.3 Disadvantages

We cannot read and write data with different clock sign in this design. Because this design is a synchronous FIFO module.

6.4 Future works

FIFO logic described here can be further refined to make more advanced-level projects. For example, in system on chip designs, there are components that often run on different clocks. So to pass data from one component to another, we need an asynchronous FIFO