VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY

HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTER SCIENCE AND ENGINEERING

School year 2021-2022 – Semester 2

REPORT ASSIGNMENT

TOPIC: FIFO - First In First Out

CO1025_Logic Design with HDL

Class: CC02 - Group 3

Group members:

Nguyễn Khánh Nam - Student ID: 2153599

Lê Văn Phúc - Student ID: 2152241

Nguyễn Quang Thiện - Student ID: 2152994

Completion Day: May 4th, 2022

CONTENTS:

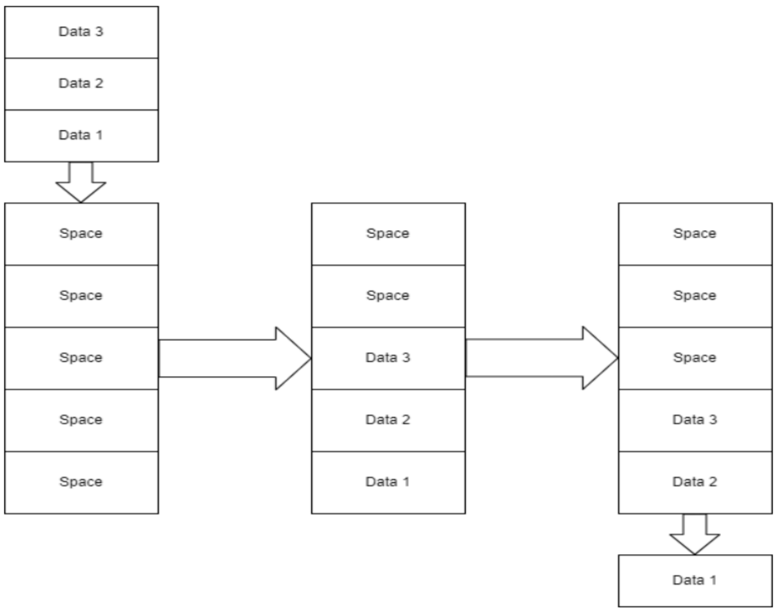
1. INTRODUCTION:

To buffer the transmission of data between processing blocks, First-In-First-Out (FIFO) memory structures are commonly utilized. Digital systems with high performance and complexity are increasingly required to transport data across modules with different, even unrelated clock frequencies. The reasons and design considerations for a resilient and scalable FIFO architecture are described in detail in this project. The suggested architecture makes use of a memory array structure that may be tweaked to work in situations where the data producer, FIFO, and data consumer are separated by many clock cycles. The goal of this project is to design, validate, and synthesize a synchronous FIFO that addresses the memory array using binary-code read and write pointers. The FIFO's RTL description is written in Verilog HDL, and the design is simulated and synthesized using VIVADO (Xilinx).

2. BACKGROUND AND APPLICATIONS:

WHAT IS A FIFO (First In First Out)?

FIFO (First-In-First-Out) is a method of dealing with program work requests from queues or stacks in which the oldest request is processed first. It is either an array of flops or Read/Write memory in hardware that stores data from one clock domain and feeds the same data to another clock domain on demand, following the first in first out logic. Write or Input logic refers to the clock domain that sends data to the FIFO, whereas Read or Output logic refers to the clock domain that reads data from the FIFO. FIFOs are used in designs to reliably transport multi-bit data words from one clock domain to another or to manage data flow between source and destination clock domains. The FIFO is considered to be synchronous if the read and write clock domains are regulated by the same clock signal, and asynchronous if the read and write clock domains are driven by distinct (asynchronous) clock signals.



Data Flow: Operation FIFO of 3 data values

The FIFO full and FIFO empty flags are of major significance because no data should be written in full condition and no data should be retrieved in empty condition since this might result in data loss or the production of irrelevant data. Binary or gray pointers are used to regulate the full and empty states of the FIFO. Because we are developing SYNCHRONOUS FIFO, we only deal with binary pointers in this report. For ASYNCHRONOUS FIFO, the gray pointers are employed to generate full and empty situations. This paper does not go into detail on why they are utilized.

FIFO SYNCHRONOUS

A synchronous FIFO is a FIFO architecture in which data values are sequentially put into a memory array using a clock signal and sequentially read out of the memory array using the same clock signal. Because there is no clock domain crossover in synchronous FIFO, the production of empty and full flags is simple. Because of this, users may create programmed partial empty and partial full flags, which are useful in a variety of applications.

APPLICATIONS

The FIFO logic provided here may be modified further to create more complex applications. Components in system-on-chip architectures, for example, frequently run on multiple clocks. So we'll need an asynchronous FIFO to transmit data from one component to the next.

FIFO is sometimes required, even though the source and request sides are controlled by the same clock signal. This is done to match the source's throughput to the request. For example, the source could be delivering data at a pace that the request can't manage, or the request might be submitting data requests at a rate that the source can't meet. Asynchronous FIFO is employed as an elastic buffer in this case.

Computer science:

Depending on the application, a FIFO could be implemented as a hardware shift register, or using different memory structures, typically a circular buffer or a kind of list. For information on the abstract data structure, see Queue (data structure). Most software implementations of a FIFO queue are not thread safe and require a locking mechanism to verify the data structure chain is being manipulated by only one thread at a time.

In computing environments that support the pipes-and-filters model for interprocess communication, a FIFO is another name for a named pipe.

Disk controllers can use the FIFO as a disk scheduling algorithm to determine the order in which to service disk I/O (Input/Output) requests, where it is also known by the same FCFS (First Come, First Served) initialism as for CPU (Central Processing Unit) scheduling mentioned before.

Communication network bridges, switches and routers used in computer networks use FIFOs to hold data packets en route to their next destination. Typically at least one FIFO structure is used per network connection. Some devices feature multiple FIFOs for simultaneously and independently queuing different types of information.

Electronics:

FIFOs are commonly used in electronic circuits for buffering and flow control between hardware and software. In its hardware form, a FIFO primarily consists of a set of read and write pointers, storage and control logic. Storage may be Static Random Access Memory (SRAM), flip-flops, latches or any other suitable form of storage. For FIFOs of non-trivial size, a dual-port SRAM is usually used, where one port is dedicated to writing and the other to reading.

The first known FIFO implemented in electronics was by Peter Alfke in 1969 at Fairchild Semiconductor. Alfke was later a director at Xilinx.

Synchronicity:

A synchronous FIFO is one in which the reading and writing clocks are the same. Asynchronous FIFOs utilize distinct clocks for reading and writing, which can cause concerns with metastability. To ensure reliable flag generation, a popular implementation of an asynchronous FIFO utilizes a Gray code (or any unit distance code) for the read and write pointers. Another thing to keep in mind about flag generation is that for asynchronous FIFO implementations, pointer arithmetic is required. In synchronous FIFO implementations, however, flags can be generated using either a leaky bucket technique or pointer arithmetic.

A hardware FIFO is used for synchronization purposes. It is often implemented as a circular queue, and thus has two pointers:

- Read pointer / read address register
- Write pointer / write address register

<u>Status flags:</u>

Full, empty, almost full, and almost empty are examples of FIFO status flags. When the read address register reaches the write address register, the FIFO is empty. When the write address register attains the read address register, the FIFO is full. The read and write addresses are both at the first memory location at the start, and the FIFO queue is empty.

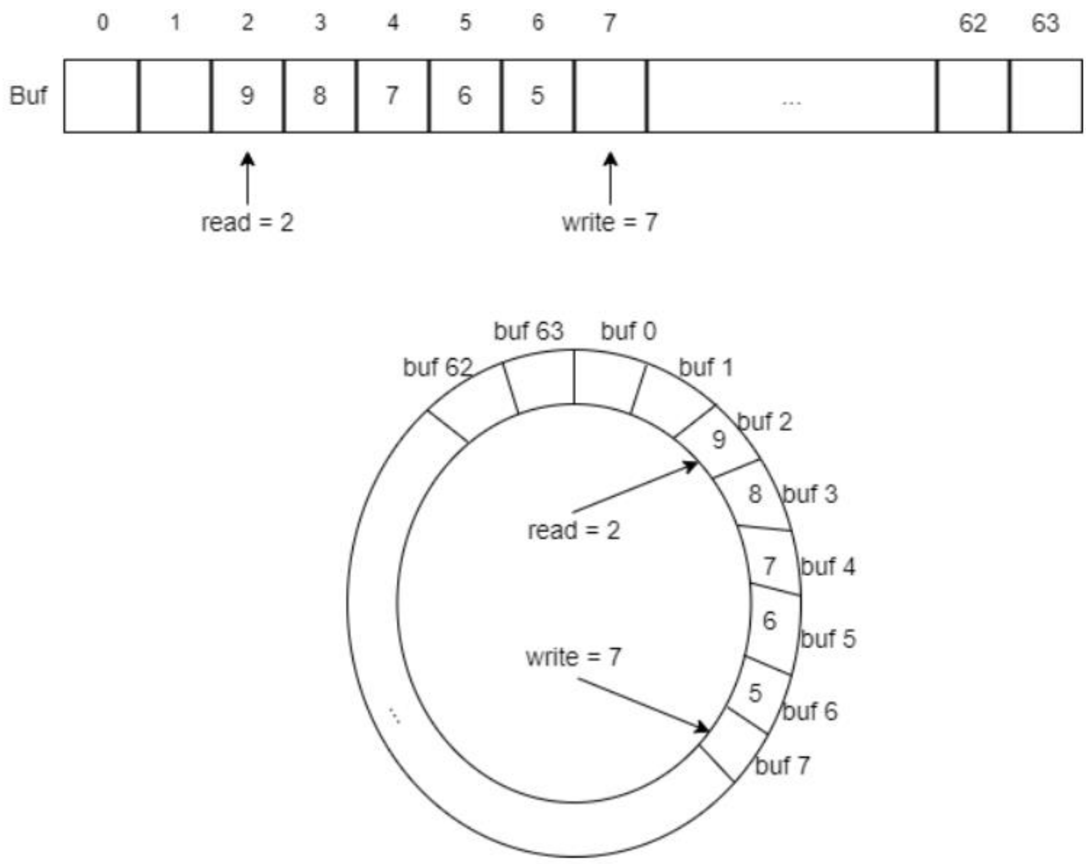
The read and write addresses are the same in both circumstances. A simple and reliable way to distinguish between the two scenarios is to add one additional bit to each read and write address, which is reversed each time the address wraps. With this set up, the disambiguation conditions are:

- When the read address register equals the write address register, the FIFO is empty.
- When the read address least significant bit (LSB) equals the write address LSBs and the extra most significant bit are different, the FIFO is full.

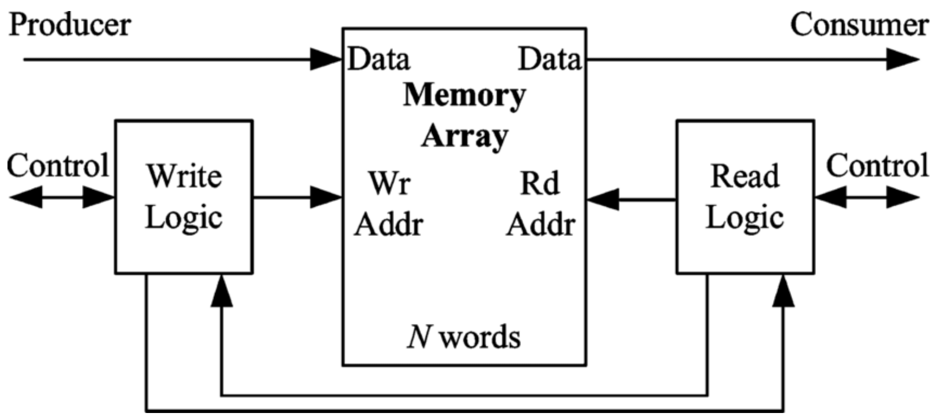
3. DESIGN:

3.1 Ideal:

The goal of this project is to design, validate, and synthesize a synchronous FIFO that addresses the memory array using binary coded read and write pointers. To prevent data overflow or underflow, FIFO full and empty flags are produced and sent on to source and destination logics, respectively. In this way data between source and destination is maintained. Creating a circular buffer using an array of memory components built such that data may be written to and read from any positions in the memory array is a more efficient technique of constructing a FIFO.

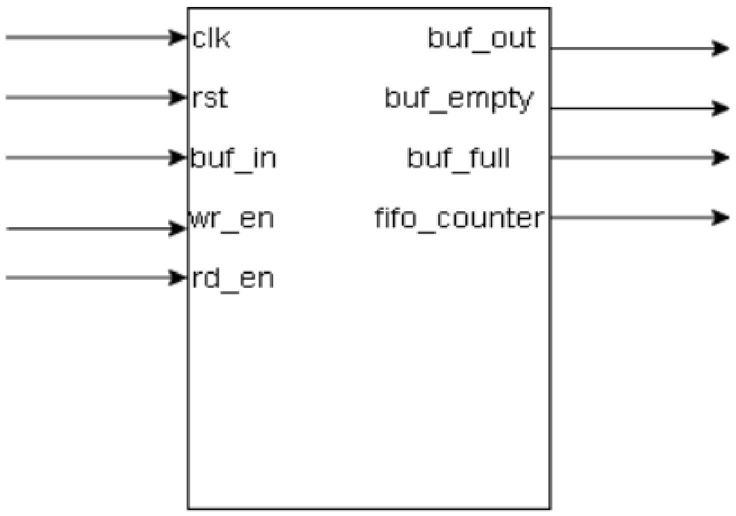


Circular buffer



Write and read control logic in a FIFO block diagram

From the block diagram, a synchronous FIFO has 3 basic building blocks, which are: memory array, write control logic and read control logic. The memory array can be implemented either with an array of flip-flops or with a dual-port read/write memory. Both of these implementations allow simultaneous read and write accesses. The FIFO's intrinsic synchronization feature is due to this simultaneous access. There are no constraints on the order in which the two ports can be accessed. This means that while one port is writing to the memory at one pace, the other port can read at a different rate, completely independent of the first. The only restriction placed is that the simultaneous read and write access should not be from/to the same memory location (flowthrough condition).



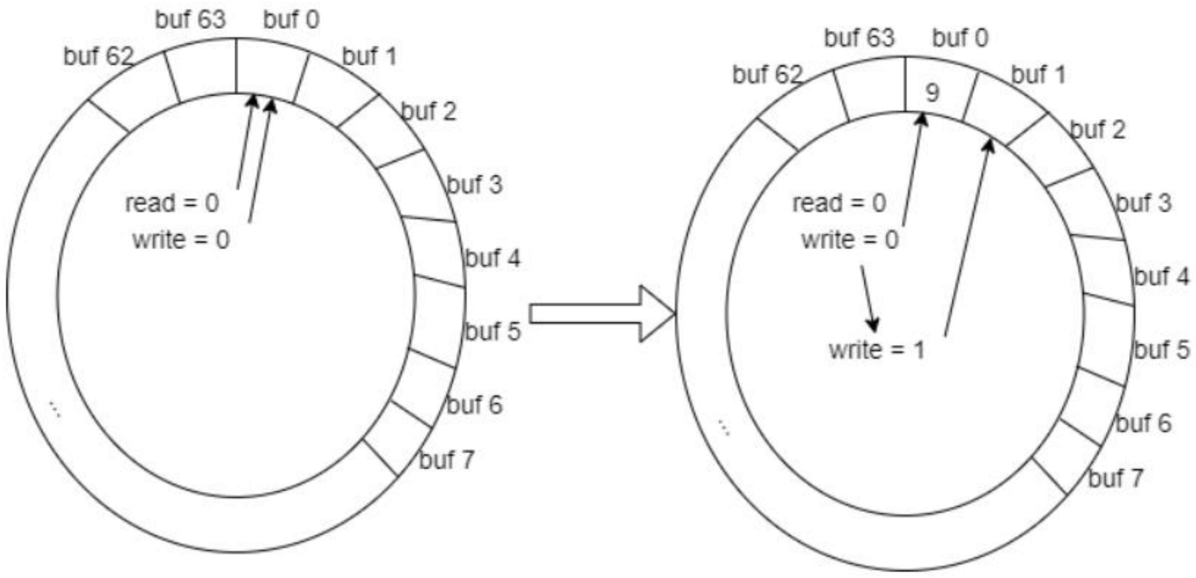
Blackbox diagram of FIFO

The Synchronous FIFO has a single clock port *clk* for both data-read and data-write operations. Data presented at the module's data-input port *buf_in* is written into the next available empty memory location on a rising clock edge when the write-enable input *wr_en* is high. The output *buf_ full* indicates that no more empty locations remain in the module's internal memory. Data can be retrieved from *buf_out* port when read-enable signal *rd_en* is high and a rising clock edge. The output *buf_empty* indicates that no more data resides in the module's internal memory.

Write and Read Control Pointer:

Write Control Pointer is used to control the write operation of the FIFO's internal memory. It generates a binary-coded write pointer which points to the memory location where the incoming data is to be written. Write pointer is incremented by one after every successful write operation. If a write request comes when FIFO is full (*buff_ full* is high), the data can not be written to the FIFO anymore till the time *buff_ full* is low.

Read Control Pointer is used to control the read operation of the FIFO's internal memory. It generates a binary-coded read pointer which points to the memory location from where the data is to be read. Read pointer is incremented by one after every successful read operation. Additionally it generates FIFO empty and only one data value in FIFO, flags which in turn are used to prevent any spurious false data read. For example if a read request comes when FIFO is empty(*buff_empty* is high) then Read Control Pointer stalls the read from the memory till the time *buff_empty* is low.



Simulate write and read pointer

Memory Array:

A memory array is a collection of flip-flops used to store data. The DEPTH of the FIFO is the number of data words that the memory array can store. The WIDTH of the FIFO refers to the length of the data word.

Memory Array's functioning is rather straightforward, as stated below:

1.It can handle simultaneous read and write enables as long as their addresses do not match

2.If *wr_en* signal is high DATA present on write data is written into the row pointed

by Write Control Pointer on the next rising edge of the clock signal clk. Note that *wr_en* is asserted only when FIFO is not full to avoid any data corruption.

3.If *rd_en* signal is high the DATA present in the row pointed by Read Control Pointer is sent onto the read data bus on the next rising edge of the clock signal clk. Note that *rd_en* is asserted only when FIFO is not empty to avoid any false data being sent to the requestor.

3.2 Flowchart:

PORT LIST

- **clk** (active high input): clock signal is provided to synchronize all operations in the FIFO.

- **rst** (active high input): when this signal is 1, the counters are set to 0.

- **buf_in** (8-bit data input): it is the data input into the FIFO.

- **wr_en** (active high input): write into FIFO.

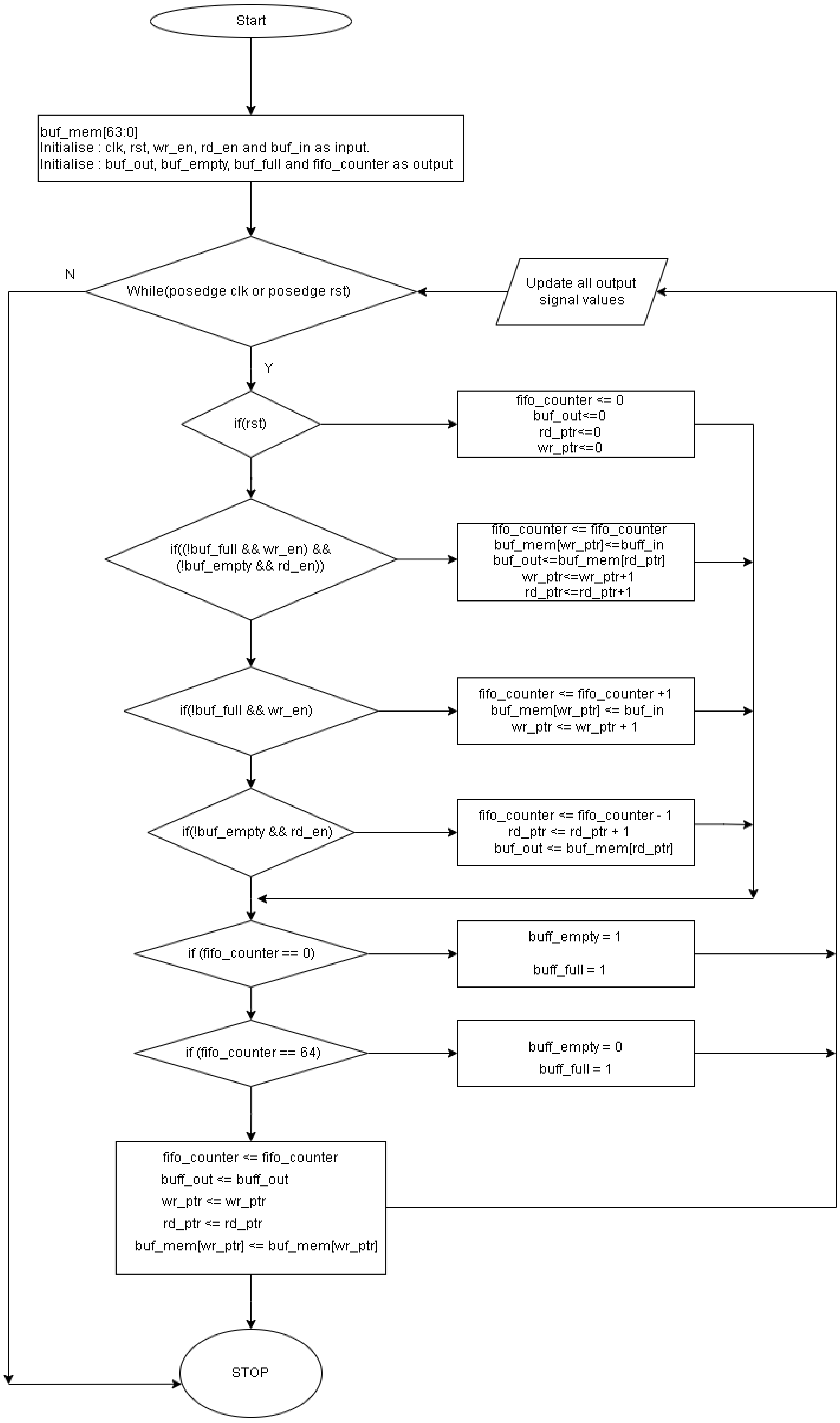
- **rd_en** (active high input):read from FIFO.

- **buf_out** (8-bit output): it is the data output from the FIFO.

- **buf_empty (**active high output):indicating that FIFO is empty.

- **buf_full** (active high output): indicating that FIFO is full.

- **fifo_counter** (7-bit output)**:** indicating the number of data values in FIFO.


Start
buf_mem[63:0]
Initialise : clk, rst, wr_en, rd_en and buf_in as input.
Initialise : buf_out, buf_empty, buf_full and fifo_counter as output
While(posedge clk or posedge rst)
N
Y
Update all output signal values
if(rst)
fifo_counter <= 0
buf_out<=0
rd_ptr<=0
wr_ptr<=0
if((!buf_full && wr_en) && (!buf_empty && rd_en))
fifo_counter <= fifo_counter
buf_mem[wr_ptr]<=buff_in
buf_out<=buf_mem[rd_ptr]
wr_ptr<=wr_ptr+1
rd_ptr<=rd_ptr+1
if(!buf_full && wr_en)
fifo_counter <= fifo_counter +1
buf_mem[wr_ptr] <= buf_in
wr_ptr <= wr_ptr + 1
if(!buf_empty && rd_en)
fifo_counter <= fifo_counter - 1
rd_ptr <= rd_ptr + 1
buf_out <= buf_mem[rd_ptr]
if (fifo_counter == 0)
buff_empty = 1
buff_full = 1
if (fifo_counter == 64)
buff_empty = 0
buff_full = 1
fifo_counter <= fifo_counter
buff_out <= buff_out
wr_ptr <= wr_ptr
rd_ptr <= rd_ptr
buf_mem[wr_ptr] <= buf_mem[wr_ptr]
STOP

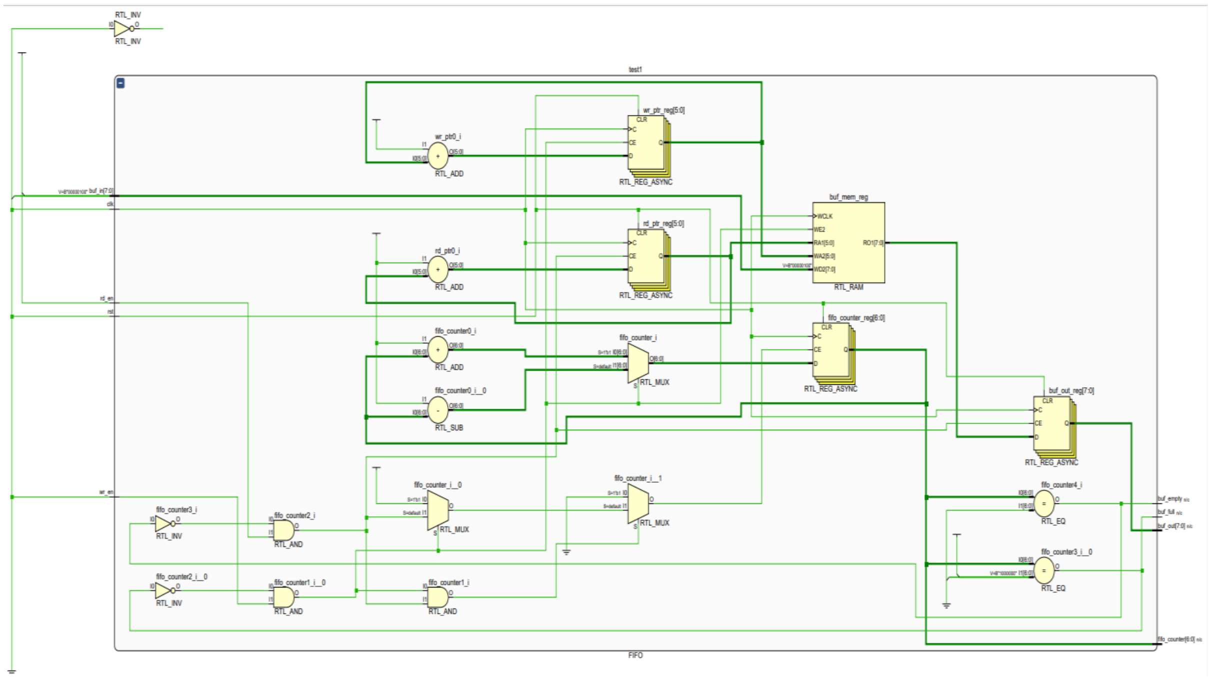
4. IMPLEMENTATION:

File name: FIFO.v, FIFO_tb.v

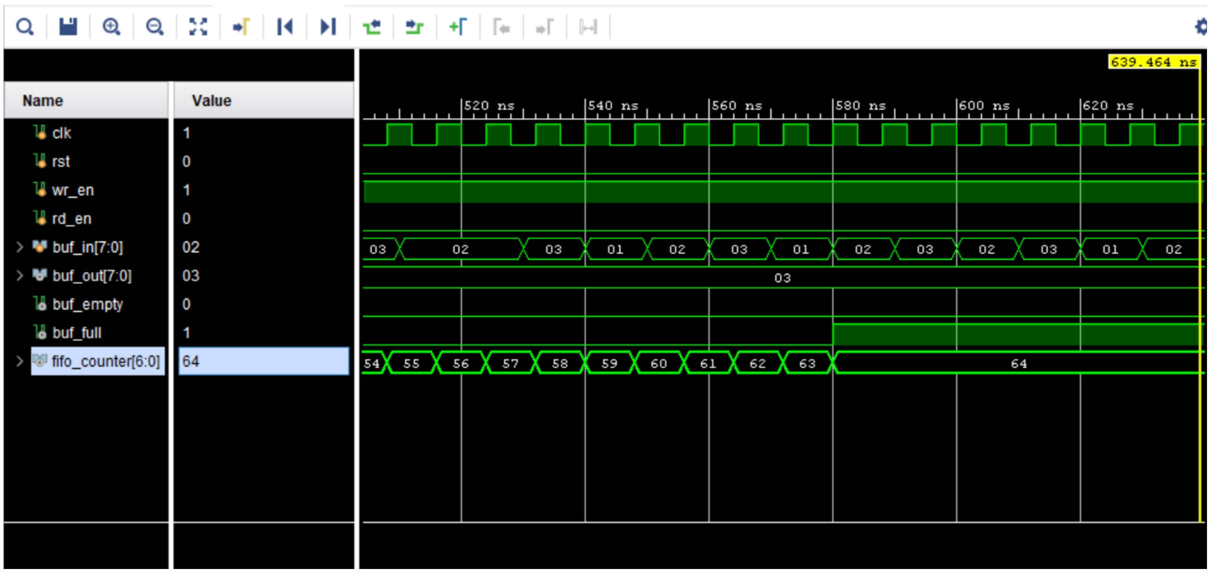
5. RESULTS:

Experimental setups: Arty-Z7-20, VIVADO (Xilinx), Laptop or Desktop.

RTL Schematic:

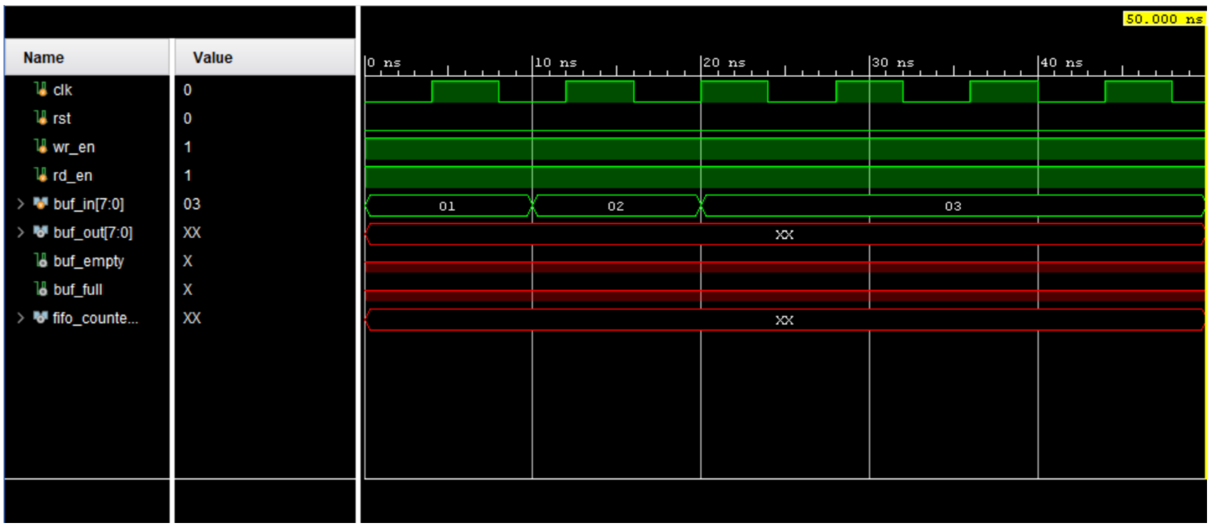


Waveform:



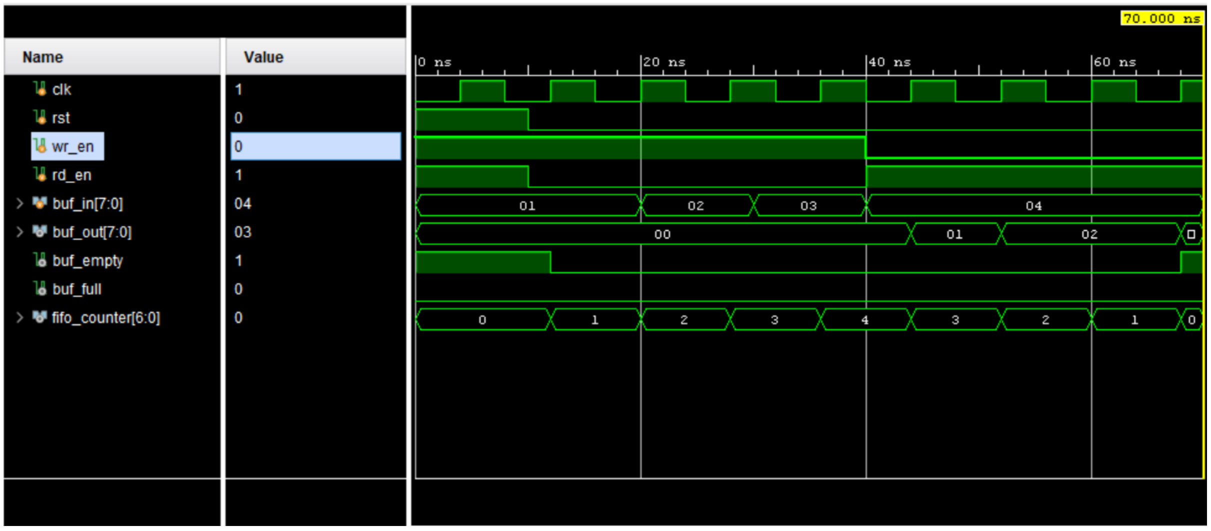
Name	Value
clk	1
rst	0
wr_en	1
rd_en	0
buf_in[7:0]	02
buf_out[7:0]	03
buf_empty	0
buf_full	1
fifo_counter[6:0]	64

buf_full = 1 => memory array full



Name	Value
clk	0
rst	0
wr_en	1
rd_en	1
buf_in[7:0]	03
buf_out[7:0]	XX
buf_empty	X
buf_full	X
fifo_counte...	XX

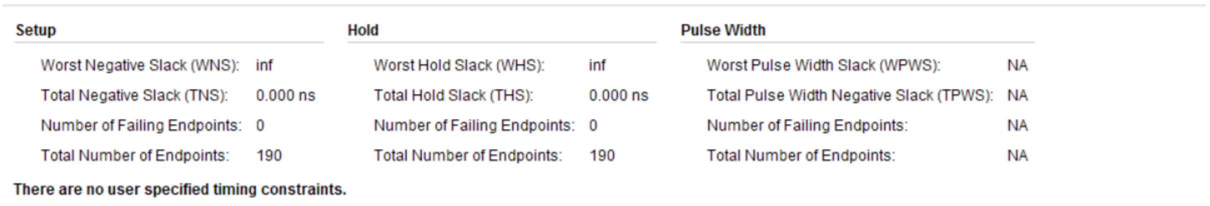
Read and Write access in the same memory location (flowthrough condition)



Name	Value
clk	1
rst	0
wr_en	0
rd_en	1
buf_in[7:0]	04
buf_out[7:0]	03
buf_empty	1
buf_full	0
fifo_counter[6:0]	0

Standard

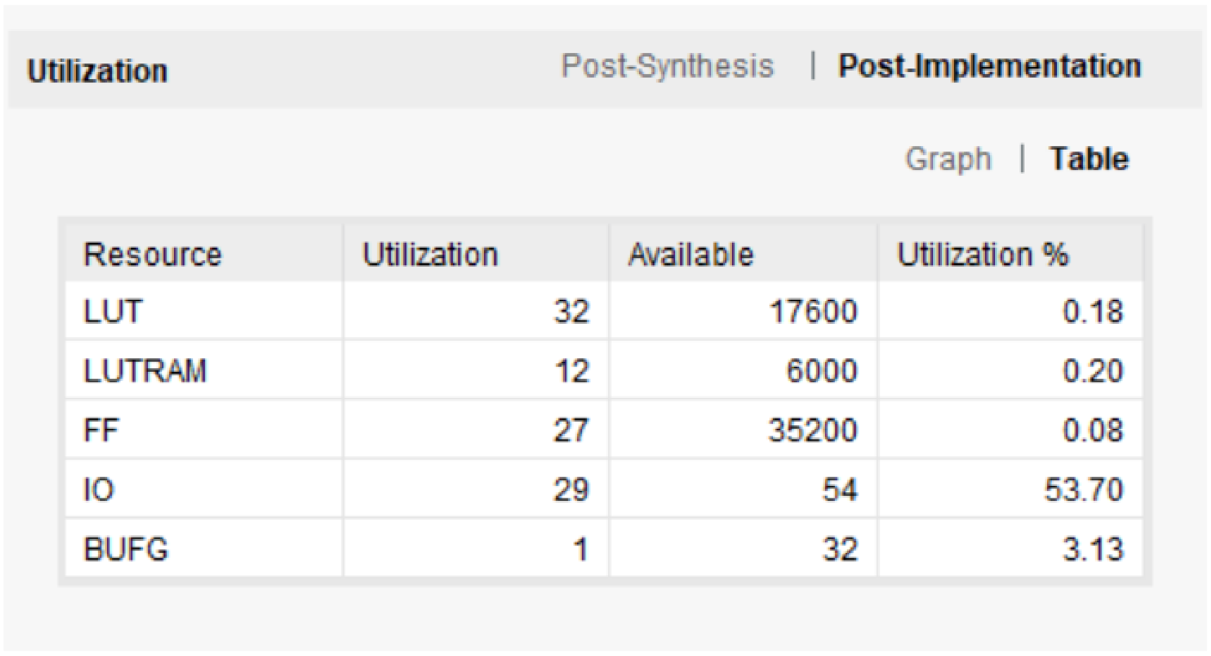
Timelines of the report:



Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	inf	Worst Hold Slack (WHS):	inf	Worst Pulse Width Slack (WPWS):	NA
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	NA
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	NA
Total Number of Endpoints:	190	Total Number of Endpoints:	190	Total Number of Endpoints:	NA

There are no user specified timing constraints.

Resources :



Resource	Utilization	Available	Utilization %
LUT	32	17600	0.18
LUTRAM	12	6000	0.20
FF	27	35200	0.08
IO	29	54	53.70
BUFG	1	32	3.13

6. CONCLUSIONS:

<u>Summary working achievement:</u>

The project's major goal was to create a functional FIFO memory block that performs appropriately according to its specifications and intent. This was accomplished by creating and simulating the FIFO design using VIVADO (Xilinx). This design's right functionality was confirmed by developing a testbench and comparing the real response to the predicted one from simulated waveforms.

<u>Pros:</u>

FIFO has several advantages as an accounting system. Among them:

- It's easy to understand and use-in fact, it's one of the most widely applied accounting methods out there, both in the U.S. and abroad.
- It makes it difficult to manipulate figures and income -the cost attached to the unit sold is always the oldest cost.
- It aligns the expected cost flow with the logical, physical flow of goods (in our example, we sold our older muffins first, remember), offering businesses a truer picture of inventory costs.
- It's a better indicator of the worth of the ending inventory-the balance sheet amount is likely to approximate the current market value.

<u>Cons:</u>

FIFO, as strong as it is, has drawbacks, particularly during periods of high inflation or persistent inflation.

Companies utilize the FIFO technique to report Cost Of Goods Sold (COGS) for a business that does not represent what production and materials actually cost at the time the financial statements are produced and presented in a rising-price environment. Instead, lesser costs are ascribed to the products sold, leaving the balance sheet with the newer, more costly inventory. As a result, FIFO can boost net income and profits by using inventory that is several years old and was purchased or manufactured at a cheaper cost to value your costs.

To put it bluntly, FIFO makes it appear that corporations are generating more money than they are, at least on paper. This higher-than-life profit, of course, entails a higher tax burden: record greater profits on your tax return, and the IRS will want a larger portion.

During instances of hyperinflation, FIFO is particularly vulnerable: When material prices rise fast and/or excessively, it frequently fails to provide a true picture of costs. In this instance, matching the oldest inventory with the most recent sales would be inappropriate and might skew the image by inflating earnings. The same thing might happen during moments of high market volatility.

Future working:

The implementation of asynchronous FIFO and verification of FIFO under boundary is an crucial role for an industry whenever they need to instantiation the ASYNC_FIFO as to store the frame or any sort of data, need to check/verify all scenario like one of method/test case i.e. boundary presently. In future, verification of FIFO as per DO-254 designing, it carry the test case to be in robustness method using one of transcript techniques like Self-Checking test bench to be added into stimulus and further one more techniques to verify the FIFO for better instance which is constrained random verification, which improves the performance and safety analysis of the FIFO which is more secure in the sense, no data loss, encryption and decryption or read/write to be done fastly including time factor.

THE END

---------- THANK YOU FOR READING ----------