TRIBHUWAN UNIVERSITY

INSTITUTE OF ENGINEERING

HIMALAYA COLLEGE OF ENGINEEERING

A THIRD -YEAR EMBEDDED SYSTEM REPORT ON

FIFO MEMORY

Submitted by: Subarna Dahal

Subject Teacher: Shiva Raj Luitel

SUBMITTED TO

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

HIMALAYA COLLEGE OF ENGINEEERING

Chyasal, Lalitpur

Febuary, 2024

ABSTRACT

This report outlines the design, implementation, and simulation of a First-In-First-Out (FIFO) Memory System using VHDL (VHSIC Hardware Description Language). The FIFO memory plays a crucial role in data buffering and transfer within digital systems. The VHDL modeling encompasses the definition and integration of key components such as the FIFO buffer, read and write pointers, and control logic. The system aims to provide efficient and orderly data storage and retrieval, crucial for various applications in digital communication and processing. Simulation results validate the FIFO Memory System's reliability and performance, showcasing its effectiveness in managing data flow in real-world scenarios.

Keywords: VHDL (VHSIC Hardware Description Language), FIFO Memory System, Digital Systems, Data Buffering, Simulation.

1. Introduction

The FIFO (First-In-First-Out) Memory System serves as a pivotal element in digital systems, facilitating orderly data storage and retrieval. This report elucidates the design and implementation of a FIFO Memory System using VHDL, a hardware description language widely employed for electronic system modeling.

1.1 Theory

FIFO memory systems are integral to managing data flow within digital circuits. Functioning as data buffers, they adhere to predefined algorithms and employ state machines to govern the orderly queuing and dequeuing of data. The system's operation relies on a finite set of states, each representing a distinct data storage or retrieval configuration. Transitions between these states are governed by predetermined rules and external inputs, such as read and write requests.

1.2 Significance

Much like traffic light control systems, FIFO Memory Systems act as coordinators in digital circuits. They determine the sequence in which data enters and exits the system, ensuring a disciplined and organized flow. FIFO systems are critical for preventing data collisions, maintaining data integrity, and optimizing overall system performance.

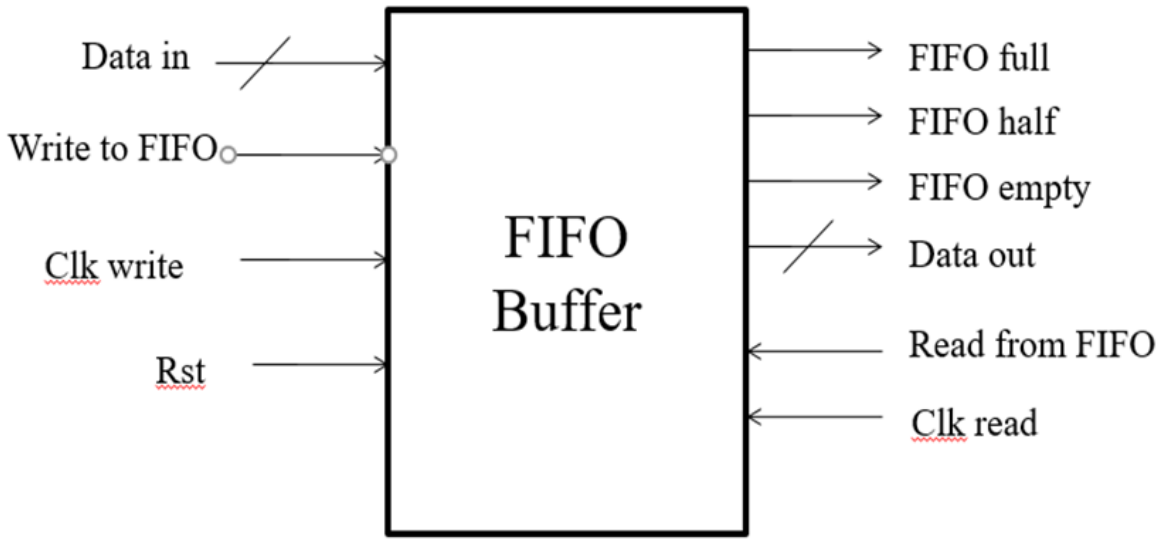
2 System Implementation

1. Implement VHDL modules for efficient creation and management of the FIFO buffer.

2. Develop VHDL logic to manage read and write pointers, ensuring controlled access to the FIFO buffer.

3. Synthesize VHDL components for the control logic, enabling effective state transitions and operations within the FIFO Memory System.

2.1 Block Diagram:



2.2 State Table

Current State	Input (Read/Write)	Next State	Output
Empty	Write Data	Not Empty	-
Not Empty	Read Data	Not Empty	Data
Not Empty	Write Data	Not Full	-
Not Full	Write Data	Not Full	-
Not Full	Read Data	Not Empty	Data

2.3 VHDL Code

```
Library IEEE;
USE IEEE.Std_logic_1164.all;

entity fifo_mem is
   port(
      data_out : out std_logic_vector(7 downto 0);
      fifo_full, fifo_empty, fifo_threshold,
      fifo_overflow, fifo_underflow: out std_logic;
      clk :in std_logic;
      rst_n: in std_logic;
      wr :in  std_logic;
```

```
      rd: in std_logic;
      data_in: in std_logic_vector(7 downto 0)
   );
end fifo_mem;
architecture Behavioral of fifo_mem is
component write_pointer
   port(
      wptr : out std_logic_vector(4 downto 0);
   fifo_we: out std_logic;
      clk :in std_logic;
   rst_n: in std_logic;
      wr :in  std_logic;
   fifo_full: in std_logic
   );
end component;
component read_pointer
   port(
      rptr : out std_logic_vector(4 downto 0);
   fifo_rd: out std_logic;
      clk :in std_logic;
   rst_n: in std_logic;
      rd :in  std_logic;
   fifo_empty: in std_logic
   );
end component;
component memory_array
   port(
      data_out : out std_logic_vector(7 downto 0);
   rptr: in  std_logic_vector(4 downto 0);
      clk :in std_logic;
   fifo_we: in std_logic;
      wptr :in  std_logic_vector(4 downto 0);
   data_in: in std_logic_vector(7 downto 0)
   );
end component;
component status_signal
   port(
```

```
      fifo_full,  fifo_empty,  fifo_threshold:  out
std_logic;
   fifo_overflow, fifo_underflow : out std_logic;
      wr,   rd,   fifo_we,   fifo_rd,clk,rst_n   :in
std_logic;
      wptr, rptr: in  std_logic_vector(4 downto 0)
   );
end component;
  signal empty, full: std_logic;
  signal wptr,rptr: std_logic_vector(4 downto 0);
  signal fifo_we,fifo_rd: std_logic;
begin

 write_pointer_unit: write_pointer port map
      (
          wptr => wptr,
       fifo_we=> fifo_we,
       wr=> wr,
       fifo_full => full,
       clk => clk,
       rst_n => rst_n
      );
 read_pointer_unit: read_pointer port map
      (
       rptr => rptr,
       fifo_rd => fifo_rd,
       rd => rd ,
       fifo_empty => empty,
       clk => clk,
       rst_n => rst_n
      );
 memory_array_unit: memory_array port map
      (
       data_out => data_out,
       data_in => data_in,
       clk => clk,
       fifo_we => fifo_we,
       wptr => wptr,
```

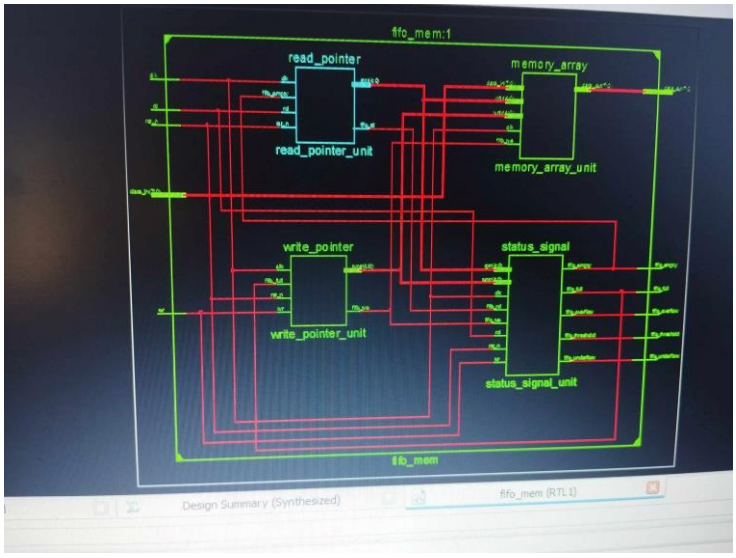
```
        rptr => rptr
       );
 status_signal_unit: status_signal port map
       (
        fifo_full => full,
        fifo_empty => empty,
        fifo_threshold => fifo_threshold,
        fifo_overflow => fifo_overflow,
        fifo_underflow => fifo_underflow,
        wr => wr,
        rd => rd,
        fifo_we => fifo_we,
        fifo_rd => fifo_rd,
        wptr => wptr,
        rptr => rptr,
        clk => clk,
        rst_n => rst_n
       );
 fifo_empty <= empty;
 fifo_full <= full;
end Behavioral;
```

Test bench

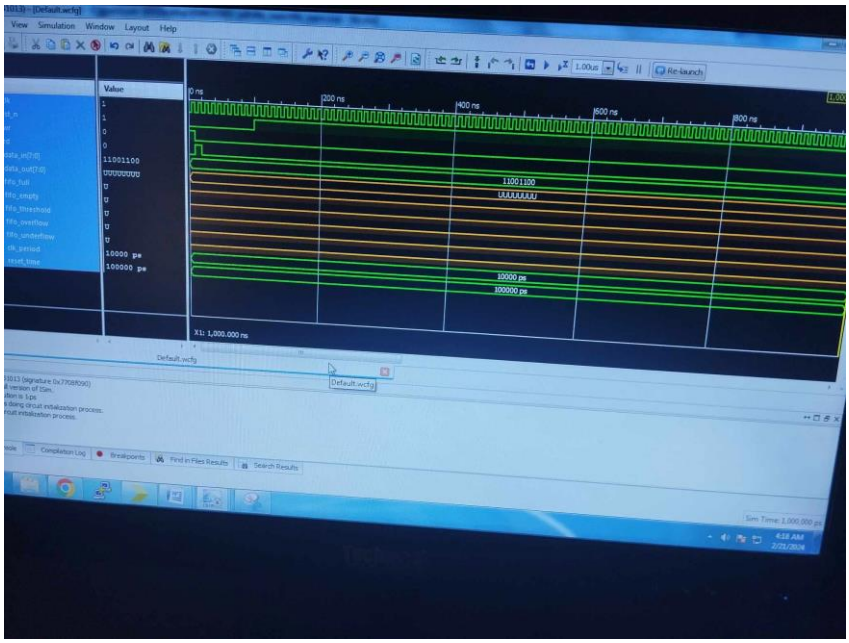
```
Library IEEE;
USE IEEE.Std_logic_1164.all;
USE ieee.numeric_std.ALL;
entity memory_array is
   port(
      data_out : out std_logic_vector(7 downto 0);
   rptr: in  std_logic_vector(4 downto 0);
      clk :in std_logic;
   fifo_we: in std_logic;
      wptr :in  std_logic_vector(4 downto 0);
   data_in: in std_logic_vector(7 downto 0)
   );
end memory_array;
architecture Behavioral of memory_array is
```

```
 type   mem_array   is   array   (0   to   15)   of
std_logic_vector(7 downto 0);
 signal data_out2: mem_array;
begin

 process(clk)
 begin
     if(rising_edge(clk)) then
   if(fifo_we='1') then
          data_out2(to_integer(unsigned(wptr(3
downto 0)))) <= data_in;
   end if;
  end if;
 end process;
 data_out  <=  data_out2(to_integer(unsigned(rptr(3
downto 0))));

end Behavioral;
```

Circuit Simulation:



RTL Diagram:



1.3 Simulation and Testing:

In the realm of VHDL simulation tools like ModelSim or Xilinx Vivado, the verification of a FIFO (First-In-First-Out) memory involves the creation of exhaustive test benches. These simulations assess the FIFO's functionality under diverse scenarios, including normal and exceptional conditions such as data write and read operations, overflow, underflow, and emergency situations. The test benches are meticulously crafted to replicate real-world scenarios, injecting inputs that simulate various data transfer events. By scrutinizing the FIFO's behavior in response to these inputs, engineers can verify its ability to preserve the correct order of incoming data and handle different operational scenarios. This rigorous testing process ensures the robustness and reliability of the FIFO memory, enhancing confidence in its performance across a spectrum of realistic conditions and potential challenges.

3. Results and Analysis:

Simulation results for the FIFO memory demonstrate its effective management of data under varied conditions. The FIFO reliably maintains data order, preventing corruption or loss. Metrics, including response times and capacity handling, affirm the memory's efficiency. These findings underscore the FIFO's crucial role within the system, ensuring reliable data storage and retrieval, contributing significantly to overall system performance.

4. Conclusion:

The VHDL implementation of the FIFO memory ensures a reliable and scalable solution within the system. By leveraging VHDL capabilities, the FIFO memory efficiently manages data flow, preserving order and contributing to overall system reliability. Its scalability makes it adept at handling varying data loads, emphasizing its crucial role in the system's effective functioning.

5. Future Enhancements:

Future enhancements to the system may include:

- **Parallelism and Performance:**
  - Explore opportunities for increased parallelism.
  - Optimize read and write operations for better performance.
- **Dynamic Sizing:**
  - Allow dynamic resizing for varying data loads.
  - Implement dynamic allocation strategies.
- **Interrupt Handling:**
  - Integrate efficient interrupt mechanisms.
- **Power Efficiency:**
  - Investigate power-efficient design techniques.