Cranes Varsity

'Where Technology Meets Excellence'

PROJECT REPORT ON

"FIFO-Based Data Buffer for UART Communication"

SUBMITTED BY

AKSHAY GOWDA CH

UNDER THE GUIDENCE OF

MS.KANCHANA SHETTIGAR

TABLE OF CONTENTS

1. ABSTRACT

This project implements a FIFO-based data buffer system for UART (Universal Asynchronous Receiver Transmitter) communication using Verilog HDL. UART is a widely used protocol for serial communication, and incorporating FIFO (First-In-First-Out) buffering ensures smooth, reliable data handling between devices of different data rates. The FIFO buffer temporarily stores incoming or outgoing data bytes, reducing data loss due to timing mismatches between sender and receiver. This report describes the design, simulation, and functionality of a FIFO buffer integrated with UART communication for effective serial data handling.

2. INTRODUCTION

In digital communication systems, especially those using UART (Universal Asynchronous Receiver Transmitter), managing asynchronous data flow is a key challenge due to differences in data rates between transmitting and receiving devices. Since UART doesn't use a shared clock, timing mismatches can lead to data loss or errors. A FIFO (First-In, First-Out) buffer helps by temporarily storing incoming or outgoing data, allowing each side to process data at its own pace without losing information.

Integrating a FIFO buffer with UART enhances communication reliability and efficiency, especially in embedded systems and microcontroller applications. It prevents data overruns and under runs, supports bursty data transfer, and ensures smooth operation even when the CPU is busy with other tasks. FIFO buffers also make UART systems more scalable and easier to integrate into real-time and resource-constrained environments.

3. OBJECTIVES

1. **Project Focus**:
   Clearly define the primary focus of the project, which is the design and implementation of a FIFO (First-In-First-Out) based data buffering system to enhance UART (Universal Asynchronous Receiver/Transmitter) communication using Verilog as the hardware description language.

2. **Importance of Data Buffering**:
   Highlight the critical role of data buffering in digital communication systems, particularly in maintaining data integrity and smooth data transfer between systems with mismatched processing speeds.

3. **Relevance of FIFO in UART Communication**:
   Explain why FIFO architecture is chosen for this project, emphasizing its efficiency in managing asynchronous data flow, minimizing data loss, and its widespread use in UART systems where timing mismatches frequently occur.

4. **Project Goals**:
   Outline the specific objectives of the project, which include creating a reliable, scalable, and resource-efficient FIFO module that seamlessly integrates with UART hardware, supports variable data widths, and ensures accurate transmission and reception of serial data.

5. **System Components**:
   Briefly introduce the key components of the system, including the FIFO buffer, UART transmitter and receiver modules, control logic for data flow management, and status flags (such as full, empty, and count indicators), setting the foundation for more detailed descriptions in later sections.

6. **User-Friendly Functionality**:
   Emphasize the importance of designing a system that supports easy configurability, such as setting baud rates, enabling or disabling interrupts, and providing real-time status feedback to the user or host processor for efficient communication handling.

4. METHODOLOGY

1.Inputs:

Clk: Clock signal.
rst_n: Active-low reset signal.
write_en: Write enable signal for writing data to the FIFO.
read_en: Read enable signal for reading data from the FIFO.
data_in: 8-bit input data to be written into the FIFO.

2.Outputs:

data_out: 8-bit output data read from the FIFO.
fifo_full: FIFO full flag (high when FIFO is full).
fifo_empty: FIFO empty flag (high when FIFO is empty).
fifo_count: 5-bit binary output representing the current number of elements in the FIFO (0 to 32).

3. Registers:

fifo_mem [FIFO_DEPTH-1:0]: Internal memory for storing data in the FIFO, where the size is determined by FIFO_DEPTH.
write_ptr: 3-bit register for the write pointer (used to keep track of where the next data will be written).
read_ptr: 3-bit register for the read pointer (used to keep track of where the next data will be read).
fifo_count_reg: 5-bit register to track the current number of elements in the FIFO.

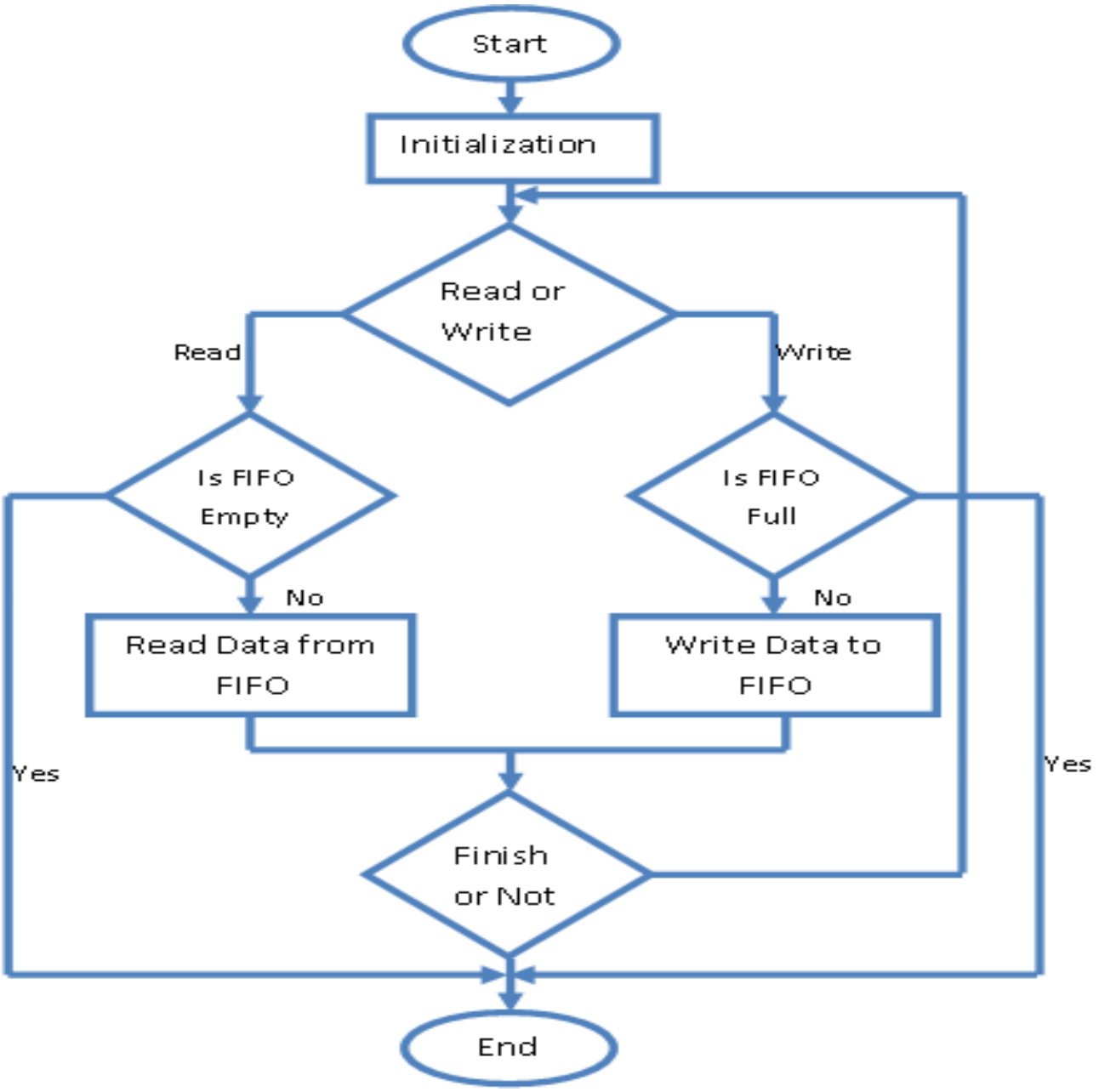
4.Always Block:

The module uses two "always" blocks:
One block is sensitive to the rising edge of clk and the active-low reset signal (rst_n) to handle the write operation.
Another block is sensitive to the rising edge of clk and the active-low reset signal (rst_n) to handle the read operation.

5.Reset Condition:

If the reset signal (rst_n) is asserted (active-low), all registers and pointers are reset to their initial states:
write_ptr is set to 0.
read_ptr is set to 0.
fifo_count_reg is set to 0, indicating the FIFO is empty.
data_out is set to 8'b0.

5. FLOW CHART


Start
Initialization
Read or Write
Read
Write
Is FIFO Empty
Is FIFO Full
No
No
Read Data from FIFO
Write Data to FIFO
Yes
Yes
Finish or Not
End

6. PROGRAM :

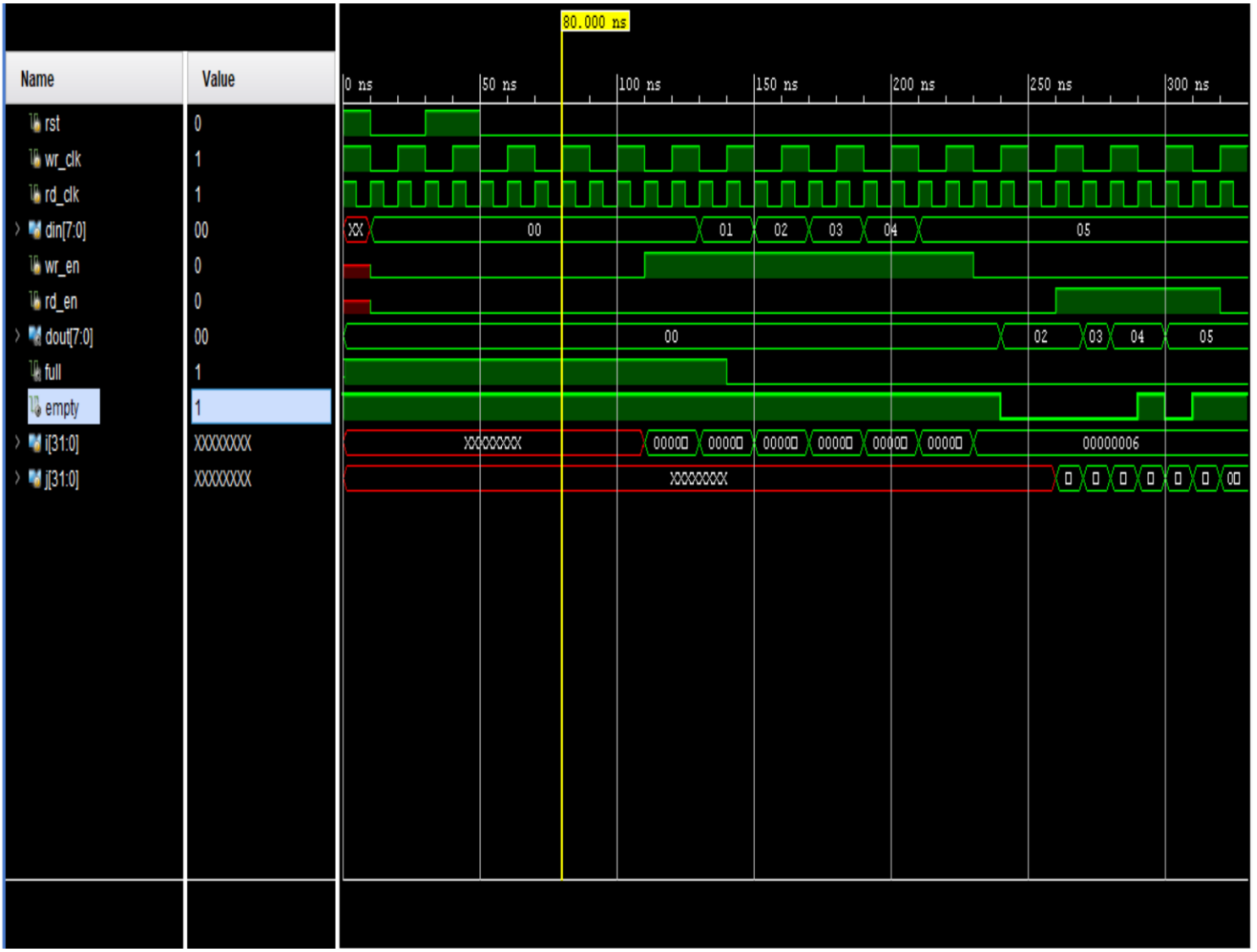
```
module fifo (
    input wire clk,              // Clock signal
    input wire rst_n,            // Active-low reset
    input wire write_en,         // Write enable
    input wire read_en,          // Read enable
    input wire [7:0] data_in,    // Input data (8-bit UART data)
    output reg [7:0] data_out,   // Output data
    output wire fifo_full,       // FIFO full flag
    output wire fifo_empty,      // FIFO empty flag
    output wire [4:0] fifo_count  // Number of stored elements
);
    // Parameters
    parameter FIFO_DEPTH = 6;
    parameter DATA_WIDTH = 8;
    // Internal memory (FIFO storage)
    reg [DATA_WIDTH-1:0] fifo_mem [FIFO_DEPTH-1:0];
    // Pointers and counter
    reg [2:0] write_ptr;   // 3 bits enough for depth 6
    reg [2:0] read_ptr;
    reg [4:0] fifo_count_reg;
    // Write operation
    always @(posedge clk or negedge rst_n) begin
        if (~rst_n) begin
            write_ptr <= 3'b0;
```

```
            fifo_count_reg <= 5'b0;
        end else if (write_en && !fifo_full) begin
            fifo_mem[write_ptr] <= data_in;
            write_ptr <= (write_ptr == FIFO_DEPTH - 1) ? 0 : write_ptr + 1'b1;
            fifo_count_reg <= fifo_count_reg + 1'b1;
        end
    end
    // Read operation
    always @(posedge clk or negedge rst_n) begin
        if (~rst_n) begin
            read_ptr <= 3'b0;
            data_out <= 8'b0;
        end else if (read_en && !fifo_empty) begin
            data_out <= fifo_mem[read_ptr];
            read_ptr <= (read_ptr == FIFO_DEPTH - 1) ? 0 : read_ptr + 1'b1;
            fifo_count_reg <= fifo_count_reg - 1'b1;
        end
    end
// FIFO status signals
    assign fifo_full = (fifo_count_reg == FIFO_DEPTH);
    assign fifo_empty = (fifo_count_reg == 0);
    assign fifo_count = fifo_count_reg;
endmodule
```

TEST BENCH MODULE

```
timescale 1ns/1ps
module fifo_tb;
   reg clk;
   reg rst_n;
   reg write_en;
   reg read_en;
   reg [7:0] data_in;
   wire [7:0] data_out;
   wire fifo_full;
   wire fifo_empty;
   wire [4:0] fifo_count;
   fifo uut (
      .clk(clk),
      .rst_n(rst_n),
      .write_en(write_en),
      .read_en(read_en),
      .data_in(data_in),
      .data_out(data_out),
      .fifo_full(fifo_full),
      .fifo_empty(fifo_empty),
      .fifo_count(fifo_count)
   );
   always #5 clk = ~clk;
   initial begin
      clk = 0;
      rst_n = 0;
      write_en = 0;
      read_en = 0;
      data_in = 8'd0;
      #10;
      rst_n = 1;
      // Write specific values: 1, 2, 3, 4
      @(posedge clk); write_en = 1; data_in = 8'd1;
      @(posedge clk); data_in = 8'd2;
      @(posedge clk); data_in = 8'd3;
      @(posedge clk); data_in = 8'd4;
      @(posedge clk); write_en = 0;
      #20;
      repeat(4) begin
         @(posedge clk);
         read_en = 1;
      end
      @(posedge clk); read_en = 0;
      #20;
      $finish;
   end
   initial begin
      $monitor("Time=%0t | wr_en=%b rd_en=%b data_in=%d data_out=%d full=%b empty=%b
count=%d",
            $time, write_en, read_en, data_in, data_out, fifo_full, fifo_empty, fifo_count);
   end
endmodule
```

7. PROGRAM EXECUTION RESULTS



8. ADVANTAGES

1. Smooth Data Flow
2. Reduced Data Loss
3. Efficient CPU Utilization
4. supports Burst Transfers
5. Simplifies Flow Control
6. Better timing tolerance
7. Flexible Design
8. Hardware Efficiency

9. CONCLUSION

The design and implementation of a FIFO-based data buffer for UART communication using Verilog has been a valuable learning experience, offering practical insights into digital design, data buffering techniques, and serial communication protocols. This project aimed to develop a reliable and efficient data handling mechanism that ensures smooth and error-free UART communication by decoupling data transmission and reception from processor timing constraints.