Assignment No.2

EE5530 : Principles of SoC Functional Verification

Nakul C - 122101024

Problem Statement

Design and verify an Asynchronous FIFO with:

- **Register File** for data storage.
- **FIFO Controller** to manage pointers and flags.
- **B2G & G2B Converters** for CDC safety.
- **Double-Flop Synchronizers** to prevent metastability.

Testbench must:

- **Generate independent clocks** for write/read.
- **Use task-based self-checking** for reads/writes.
- **Trigger deadlock** via full/empty flags.

Asynchronous FIFO Design

Overview

An asynchronous FIFO (First-In-First-Out) buffer is an essential component in digital systems, enabling reliable data transfer between subsystems operating under different clock domains. In digital designs, it is common for various modules to function at distinct clock frequencies, leading to challenges in data synchronization and transfer. The asynchronous FIFO addresses these challenges by providing a buffer where data written by a producer in one clock domain can be safely stored and subsequently read by a consumer in another clock domain, even when these clocks are asynchronous. This mechanism ensures data integrity and prevents issues such as data loss or corruption that can arise from direct clock domain crossings.

Design Features

The design of an asynchronous FIFO incorporates several critical features to manage the complexities of inter-clock communication:

- **Gray Code Pointer Representation:** One key aspect is the use of Gray code for pointer representation, which minimizes the risk of metastability—a condition where a system can enter an unpredictable state due to asynchronous signal changes. In Gray code, only one bit changes between successive values, reducing the likelihood of errors during pointer synchronization between clock domains.
- **Dual-Clock Architecture:** The FIFO employs a dual-clock architecture, with separate write and read clocks governing the respective operations. This separation allows each subsystem to operate independently while coordinating data flow through the FIFO.
- **Status Flags:** Furthermore, the implementation of full and empty flags provides essential feedback to the system, indicating the FIFO's status and preventing scenarios where data is either overwritten or read prematurely.

Collectively, these features ensure that the asynchronous FIFO functions as a robust intermediary, facilitating seamless and error-free data transfer across differing clock domains.

DUT

```
module async_fifo #(
    parameter DATA_WIDTH = 8,
    parameter ADDR_WIDTH = 4
)(
    input wire wr_clk,              // Write clock
    input wire rd_clk,              // Read clock
    input wire wr_rst_n,            // Write domain reset (active low)
    input wire rd_rst_n,            // Read domain reset (active low)
    input wire wr_en,               // Write enable
    input wire rd_en,               // Read enable
    input wire [DATA_WIDTH-1:0] wr_data, // Write data
    output reg [DATA_WIDTH-1:0] rd_data, // Read data
    output wire fifo_full,          // FIFO full flag
    output wire fifo_empty          // FIFO empty flag
);

    // FIFO Depth
    localparam FIFO_DEPTH = 1 << ADDR_WIDTH;

    // FIFO Memory
    reg [DATA_WIDTH-1:0] fifo_mem [0:FIFO_DEPTH-1];

    // Write Pointer (Gray Code)
    reg [ADDR_WIDTH:0] wr_ptr_bin = 0;
    reg [ADDR_WIDTH:0] wr_ptr_gray = 0;
    reg [ADDR_WIDTH:0] wr_ptr_gray_sync1 = 0;
    reg [ADDR_WIDTH:0] wr_ptr_gray_sync2 = 0;

    // Read Pointer (Gray Code)
    reg [ADDR_WIDTH:0] rd_ptr_bin = 0;
    reg [ADDR_WIDTH:0] rd_ptr_gray = 0;
    reg [ADDR_WIDTH:0] rd_ptr_gray_sync1 = 0;
```

```
33      reg [ADDR_WIDTH:0] rd_ptr_gray_sync2 = 0;
34
35      // Write Pointer Synchronization to Read Clock Domain
36      always @(posedge rd_clk or negedge rd_rst_n) begin
37          if (!rd_rst_n) begin
38              wr_ptr_gray_sync1 <= 0;
39              wr_ptr_gray_sync2 <= 0;
40          end else begin
41              wr_ptr_gray_sync1 <= wr_ptr_gray;
42              wr_ptr_gray_sync2 <= wr_ptr_gray_sync1;
43          end
44      end
45
46      // Read Pointer Synchronization to Write Clock Domain
47      always @(posedge wr_clk or negedge wr_rst_n) begin
48          if (!wr_rst_n) begin
49              rd_ptr_gray_sync1 <= 0;
50              rd_ptr_gray_sync2 <= 0;
51          end else begin
52              rd_ptr_gray_sync1 <= rd_ptr_gray;
53              rd_ptr_gray_sync2 <= rd_ptr_gray_sync1;
54          end
55      end
56
57      // Binary to Gray Code Conversion
58      function [ADDR_WIDTH:0] bin2gray(input [ADDR_WIDTH:0] bin);
59          bin2gray = bin ^ (bin >> 1);
60      endfunction
61
62      // Gray to Binary Code Conversion
63      function [ADDR_WIDTH:0] gray2bin(input [ADDR_WIDTH:0] gray);
64          integer i;
65          begin
66              gray2bin[ADDR_WIDTH] = gray[ADDR_WIDTH];
67              for (i = ADDR_WIDTH-1; i >= 0; i = i - 1)
68                  gray2bin[i] = gray2bin[i+1] ^ gray[i];
69          end
70      endfunction
71
72      // Write Operation
73      always @(posedge wr_clk or negedge wr_rst_n) begin
74          if (!wr_rst_n)
75            begin
76              wr_ptr_bin <= 0;
77              wr_ptr_gray <= 0;
78          end
79        else if (wr_en && !fifo_full)
80          begin
81              fifo_mem[wr_ptr_bin[ADDR_WIDTH-1:0]] <= wr_data;
82              wr_ptr_bin <= wr_ptr_bin + 1;
83              wr_ptr_gray <= bin2gray(wr_ptr_bin + 1);
84          end
85      end
```

```
86
87      // Read Operation
88      always @(posedge rd_clk or negedge rd_rst_n) begin
89        if (!rd_rst_n)
90          begin
91              rd_ptr_bin <= 0;
92              rd_ptr_gray <= 0;
93              rd_data <= 0;
94          end
95        else if (rd_en && !fifo_empty)
96          begin
97              rd_data <= fifo_mem[rd_ptr_bin[ADDR_WIDTH-1:0]];
98              rd_ptr_bin <= rd_ptr_bin + 1;
99              rd_ptr_gray <= bin2gray(rd_ptr_bin + 1);
100         end
101     end
102
103     // FIFO Full Condition
104     assign fifo_full = (wr_ptr_gray == {~rd_ptr_gray_sync2[ADDR_WIDTH:
          ADDR_WIDTH-1], rd_ptr_gray_sync2[ADDR_WIDTH-2:0]});
105
106     // FIFO Empty Condition
107     assign fifo_empty = (rd_ptr_gray == wr_ptr_gray_sync2);
108
109 endmodule
```

Code Explanation

Parameterization

The FIFO design includes two primary parameters:

- `DATA_WIDTH`: Specifies the width of the data in bits, determining the size of each data word stored in the FIFO.
- `ADDR_WIDTH`: Defines the width of the address pointers, which in turn determines the depth of the FIFO. The depth is calculated as $2^{\texttt{ADDR\_WIDTH}}$, allowing for a scalable number of storage locations based on the address width.

Memory Architecture

The core component of the FIFO is a dual-port memory array, `fifo_mem`, which allows simultaneous read and write operations. This memory array has a depth of $2^{\texttt{ADDR\_WIDTH}}$ and a width of `DATA_WIDTH`. The dual-port nature facilitates concurrent access from both the write and read sides, essential for asynchronous operations where the write and read clocks are independent.

Pointer Management with Gray Code

To manage read and write operations, the FIFO employs binary counters (`wr_ptr_bin` and `rd_ptr_bin`) for the write and read pointers, respectively. These binary pointers are converted

to Gray code (wr_ptr_gray and rd_ptr_gray) to facilitate safe synchronization across clock domains. Gray code is advantageous in this context because only one bit changes between successive values, reducing the risk of metastability during pointer synchronization.

The conversion between binary and Gray code is handled by two functions: bin2gray and gray2bin. The bin2gray function computes the Gray code equivalent of a binary number by XORing the binary number with itself right-shifted by one bit. Conversely, the gray2bin function converts a Gray code number back to binary by iteratively XORing the bits.

Pointer Synchronization Across Clock Domains

Synchronization of pointers between the write and read clock domains is crucial for maintaining data integrity. The write pointer, in Gray code form (wr_ptr_gray), is synchronized into the read clock domain using two flip-flop stages (wr_ptr_gray_sync1 and wr_ptr_gray_sync2). Similarly, the read pointer (rd_ptr_gray) is synchronized into the write clock domain. This dual-stage synchronization minimizes the risk of metastability by allowing transient states to settle before the pointer is used in the new clock domain.

Write Operation

In the write clock domain, when the write enable signal (wr_en) is asserted and the FIFO is not full (fifo_full is low), the data present on the write data input (wr_data) is written into the memory location pointed to by the binary write pointer (wr_ptr_bin). After the write operation, the binary write pointer is incremented, and the Gray code equivalent (wr_ptr_gray) is updated accordingly.

Read Operation

In the read clock domain, when the read enable signal (rd_en) is asserted and the FIFO is not empty (fifo_empty is low), the data at the memory location pointed to by the binary read pointer (rd_ptr_bin) is read and presented on the read data output (rd_data). The binary read pointer is then incremented, and the Gray code equivalent (rd_ptr_gray) is updated.

FIFO Full and Empty Conditions

The FIFO full condition is determined by comparing the synchronized read pointer in the write clock domain (rd_ptr_gray_sync2) with the write pointer (wr_ptr_gray). The FIFO is considered full when the write pointer equals the read pointer with the most significant bits inverted.

The FIFO empty condition is identified when the synchronized write pointer in the read clock domain (wr_ptr_gray_sync2) matches the read pointer (rd_ptr_gray). In this state, the FIFO has no new data to read.

By integrating these components—parameterization, dual-port memory architecture, Gray code pointer management, cross-clock domain synchronization, and control logic for read and write operations—the asynchronous FIFO ensures reliable and efficient data transfer between subsystems operating under different clock domains.

Testbench

```
1  `timescale 1ns / 1ps
2
3  module tb_async_fifo;
4
5      // Parameters
6      parameter DATA_WIDTH = 8;
7      parameter ADDR_WIDTH = 4;
8      parameter FIFO_DEPTH = 1 << ADDR_WIDTH;
9
10     // DUT Signals
11     reg wr_clk = 0, rd_clk = 0;
12     reg wr_rst_n = 0, rd_rst_n = 0;
13     reg wr_en = 0, rd_en = 0;
14     reg [DATA_WIDTH-1:0] wr_data = 0;
15     wire [DATA_WIDTH-1:0] rd_data;
16     wire fifo_full, fifo_empty;
17
18     // Instantiate the FIFO
19     async_fifo #(
20         .DATA_WIDTH(DATA_WIDTH),
21         .ADDR_WIDTH(ADDR_WIDTH)
22     ) dut (
23         .wr_clk(wr_clk),
24         .rd_clk(rd_clk),
25         .wr_rst_n(wr_rst_n),
26         .rd_rst_n(rd_rst_n),
27         .wr_en(wr_en),
28         .rd_en(rd_en),
29         .wr_data(wr_data),
30         .rd_data(rd_data),
31         .fifo_full(fifo_full),
32         .fifo_empty(fifo_empty)
33     );
34
35     // Testbench Variables
36     reg [DATA_WIDTH-1:0] test_vector [0:FIFO_DEPTH-1];
37     integer write_ptr = 0, read_ptr = 0;
38
39     // Clock Generation
40     initial begin
41         forever #5 wr_clk = ~wr_clk; // 10ns period
42     end
43
44     initial begin
45         forever #7 rd_clk = ~rd_clk; // 14ns period
46     end
47
48     // Automatic keyword used with tasks (and functions) specifies that
       the task should have automatic storage for its local variables
49
50
51     // Task: Write to FIFO
52     task automatic write_fifo(input [DATA_WIDTH-1:0] data);
```

```
53          begin
54              @(posedge wr_clk);
55              if (!fifo_full) begin
56                  wr_data = data;
57                  wr_en = 1;
58                  @(posedge wr_clk);
59                  wr_en = 0;
60                  test_vector[write_ptr] = data; // Store for verification
61                  write_ptr = (write_ptr + 1) % FIFO_DEPTH;
62                  $display("WRITE: Data = %h at time %t", data, $time);
63              end else begin
64                  $display("ATTEMPTED WRITE WHEN FULL: Data = %h at time %
                        t", data, $time);
65              end
66          end
67      endtask
68
69      // Task: Read from FIFO
70      task automatic read_fifo;
71          begin
72              @(posedge rd_clk);
73              if (!fifo_empty) begin
74                  rd_en = 1;
75                  @(posedge rd_clk); // First cycle: rd_en asserted
76                  rd_en = 0;
77                  @(posedge rd_clk);  // Second cycle: data available
78                  if (rd_data !== test_vector[read_ptr]) begin
79                      $display("ERROR: Data Mismatch! Expected = %h, Got =
                            %h at time %t", test_vector[read_ptr], rd_data,
                            $time);
80                  end else begin
81                      $display("READ: Data = %h at time %t", rd_data,
                            $time);
82                  end
83                  read_ptr = (read_ptr + 1) % FIFO_DEPTH;
84              end else begin
85                  $display("ATTEMPTED READ WHEN EMPTY at time %t", $time);
86              end
87          end
88      endtask
89
90
91      // Assertions
92      always @(posedge wr_clk) begin
93          if (wr_en && fifo_full) begin
94              $display("ASSERTION FAILED: Write attempted when FIFO is
                    full at time %t", $time);
95              $stop;
96          end
97      end
98
99      always @(posedge rd_clk) begin
100         if (rd_en && fifo_empty) begin
```

```
101                 $display("ASSERTION FAILED: Read attempted when FIFO is
                        empty at time %t", $time);
102                 $stop;
103             end
104         end
105
106         // Test Sequence
107         initial begin
108             // Apply reset
109             #20;
110             wr_rst_n = 1;
111             rd_rst_n = 1;
112
113
114             // Corner Case: Attempt to Write when Full
115             repeat (FIFO_DEPTH) begin
116                 write_fifo($random);
117                 #10;
118             end
119             write_fifo($random); // This should trigger full condition
120
121
122             // Corner Case: Attempt to Read when Empty
123             repeat (FIFO_DEPTH) begin
124
125                 read_fifo();
126                 #14;
127             end
128             read_fifo(); // This should trigger empty condition
129
130
131             // Finish Simulation
132             $display("TEST COMPLETED");
133             $finish;
134         end
135     endmodule
```

Testbench Explanation

Introduction

The testbench for the **asynchronous FIFO** verifies the functional correctness of the design by simulating **read** and **write** operations under different conditions. The FIFO operates with independent **read** and **write clocks**, making it suitable for scenarios requiring **clock domain crossing**. The testbench checks the FIFO's behavior in normal operation, as well as **corner cases** such as attempting to **write when full** and attempting to **read when empty**.

Testbench Implementation

The testbench is implemented using **Verilog**, where a FIFO with parameterized **data width** and **address width** is instantiated and tested under various conditions. The key elements of the testbench include:

- **Clock generation** for independent **read** and **write** operations.
- **Tasks** for **writing** to and **reading** from the FIFO.
- **Assertions** to check for **erroneous conditions**.
- **Simulation sequence** covering **normal and edge cases**.

Clock Generation

Two independent **clock signals** are generated using an **infinite loop** with delays:

- **Write clock** (wr_clk) toggles every **5 ns**, resulting in a **10 ns period**.
- **Read clock** (rd_clk) toggles every **7 ns**, resulting in a **14 ns period**.

Write and Read Operations

The testbench defines two **tasks** for **writing** and **reading** FIFO data:

- **Write Task:** Writes **data** to the FIFO when it is **not full** and stores it in a **test vector** for verification.
- **Read Task:** Reads **data** from the FIFO when it is **not empty** and compares it with the expected value stored in the **test vector**.

Each task ensures that **data integrity** is maintained and logs messages indicating **successful operations** or **mismatches**.

Assertions and Error Handling

To ensure **proper FIFO operation**, the testbench includes **assertions** that halt the simulation in case of **invalid operations**:

- An assertion triggers if a **write is attempted when the FIFO is full**.
- An assertion triggers if a **read is attempted when the FIFO is empty**.

Simulation Sequence

The test sequence follows these steps:

1. Apply an **initial reset**.
2. Perform **write operations** until the FIFO is **full**, followed by an additional **write attempt** to test the **full condition**.
3. Perform **read operations** until the FIFO is **empty**, followed by an additional **read attempt** to test the **empty condition**.
4. Log **test completion** and terminate the **simulation**.

Results

```
# KERNEL: WRITE: Data = 24 at time                    35000
# KERNEL: WRITE: Data = 81 at time                    55000
# KERNEL: WRITE: Data = 09 at time                    75000
# KERNEL: WRITE: Data = 63 at time                    95000
# KERNEL: WRITE: Data = 0d at time                   115000
# KERNEL: WRITE: Data = 8d at time                   135000
# KERNEL: WRITE: Data = 65 at time                   155000
# KERNEL: WRITE: Data = 12 at time                   175000
# KERNEL: WRITE: Data = 01 at time                   195000
# KERNEL: WRITE: Data = 0d at time                   215000
# KERNEL: WRITE: Data = 76 at time                   235000
# KERNEL: WRITE: Data = 3d at time                   255000
# KERNEL: WRITE: Data = ed at time                   275000
# KERNEL: WRITE: Data = 8c at time                   295000
# KERNEL: WRITE: Data = f9 at time                   315000
# KERNEL: WRITE: Data = c6 at time                   335000
# KERNEL: ATTEMPTED WRITE WHEN FULL: Data = c5 at time
    345000
# KERNEL: READ: Data = 24 at time                    385000
# KERNEL: READ: Data = 81 at time                    427000
# KERNEL: READ: Data = 09 at time                    469000
# KERNEL: READ: Data = 63 at time                    511000
# KERNEL: READ: Data = 0d at time                    553000
# KERNEL: READ: Data = 8d at time                    595000
# KERNEL: READ: Data = 65 at time                    637000
# KERNEL: READ: Data = 12 at time                    679000
# KERNEL: READ: Data = 01 at time                    721000
# KERNEL: READ: Data = 0d at time                    763000
# KERNEL: READ: Data = 76 at time                    805000
# KERNEL: READ: Data = 3d at time                    847000
# KERNEL: READ: Data = ed at time                    889000
# KERNEL: READ: Data = 8c at time                    931000
# KERNEL: READ: Data = f9 at time                    973000
# KERNEL: READ: Data = c6 at time                   1015000
# KERNEL: ATTEMPTED READ WHEN EMPTY at time                  1029000
# KERNEL: TEST COMPLETED
```

The results of the testbench simulation are summarized below:

- **Successful Writes:** Data was correctly written to the FIFO at expected timestamps.
- **Write When Full:** An attempted write operation when the FIFO was full was detected and logged.
- **Successful Reads:** Data was correctly read from the FIFO at expected timestamps, matching the written values.
- **Read When Empty:** An attempted read operation when the FIFO was empty was detected and logged.
- **Test Completion:** The simulation concluded successfully without unexpected errors.

Conclusion

The testbench effectively verifies the **correct functionality** of the **asynchronous FIFO**. It ensures that **data** is properly **written** and **read** while handling **full** and **empty conditions** correctly. The use of **assertions** helps in identifying **critical issues**, making the design more **robust**.

EDAPLAYGROUND Link

You can view or run the Verilog code in EDAPLAYGROUND
Click here to visit EDAPLAYGROUND