DESIGN AND VERIFICATION OF ASYNCHRONOUS FIFO

Achyuth Krishna, Amrutha Regalla,

Sathwik Reddy Madireddy, Sai Sri Harsha Atmakuri

Abstract. In modern integrated circuits and systems-on-chip (SoCs), asynchronous FIFOs (First-In-First-Out) play a crucial role in enabling reliable communication and data transfer across clock domain boundaries. As SoCs incorporate multiple modules operating at diverse clock frequencies, asynchronous FIFOs provide a robust solution for clock domain crossing, decoupling read and write operations. These components offer several advantages, including elastic buffering to handle data rate mismatches, promotion of modular design practices, improved timing closure, and power optimization through techniques like clock gating and dynamic voltage and frequency scaling (DVFS). This paper will detail one method that is used to design, synthesize and analyse a Asynchronous FIFO between different clock domains using Gray code pointers that are synchronized into a different clock domain before testing for "FIFO full" or "FIFO empty" conditions.

Keywords: Asynchronous FIFO; Empty - full signal; Pointer – Synchronization, Metastability.

1. Introduction

An asynchronous FIFO refers to a FIFO design where data values are written to a FIFO buffer from one clock domain and the data values are read from the same FIFO buffer from another clock domain, where the two clock domains are asynchronous to each other. However, designing asynchronous FIFOs presents several challenges, particularly in generating and synchronizing FIFO pointers, and accurately determining the full and empty status of the buffer. To overcome these challenges, advanced techniques are employed, such as using Gray code for pointers to minimize synchronization errors and implementing dual-clock domain synchronizers to safely transfer pointer values across clock domains.This paper delves into these challenges and solutions, providing a comprehensive understanding of asynchronous FIFO design for reliable data transfer in complex digital systems.

2. Design Overview and Implementation

Asynchronous FIFO consists of the following components

2.1 Dual port RAM (DPRAM) has two ports, allowing the read controller and the write controller to access the storage unit asynchronously at the same time.

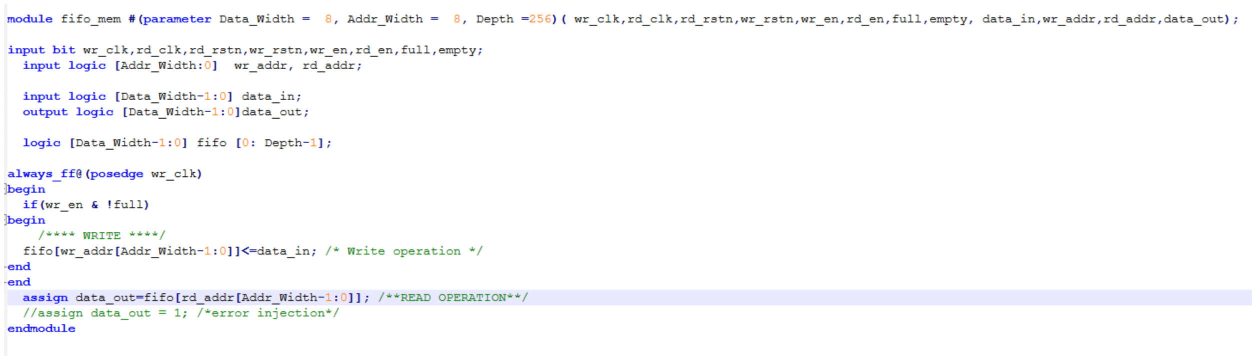
2.2***Write Control Module*****:** The function of the write control module is to control the write data and determine whether the DPRAM is full. The write address pointer is generated by the simultaneous action of the write clock and the write enable signal, and data is sequentially written to the dual port RAM.

2.3***Read Control Module:*** The function of the read control module is to control the read data and judge whether the DPRAM is empty. The read address pointer is created by the simultaneous operation of the read clock and the read enable signal, and it reads data sequentially from the DPRAM.

2.4***Clock Synchronization Module:*** This module compares the read address pointer with the write address pointer in the write clock domain after a delay of two beats and generates a full write signal.

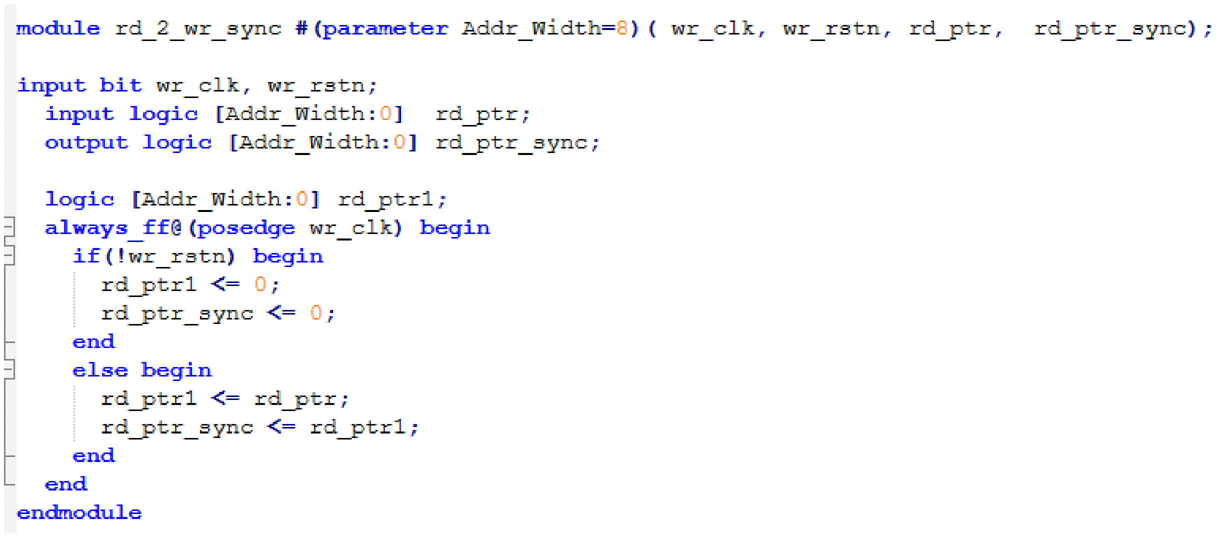
3. System Verilog Implementation

FIFO Memory:



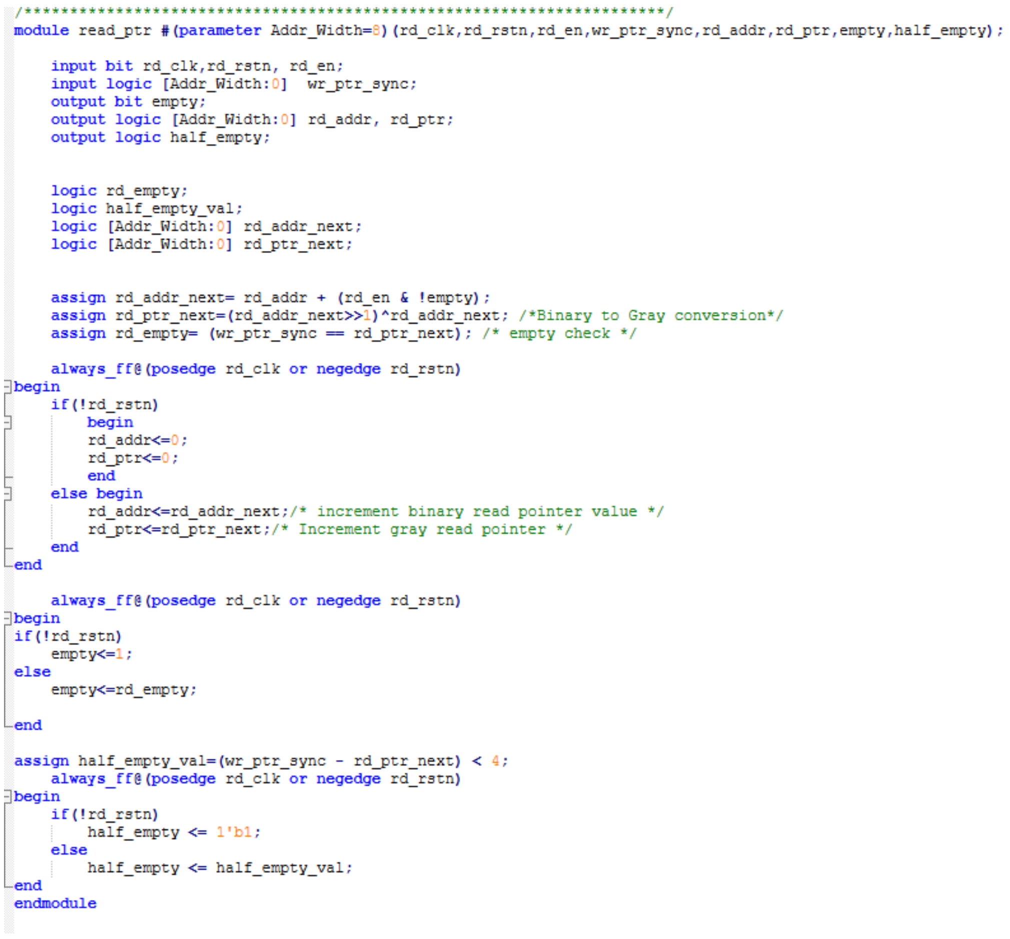
```
module fifo_mem #(parameter Data_Width =  8, Addr_Width =  8, Depth =256)( wr_clk,rd_clk,rd_rstn,wr_rstn,wr_en,rd_en,full,empty, data_in,wr_addr,rd_addr,data_out);

input bit wr_clk,rd_clk,rd_rstn,wr_rstn,wr_en,rd_en,full,empty;
  input logic [Addr_Width:0]  wr_addr, rd_addr;

  input logic [Data_Width-1:0] data_in;
  output logic [Data_Width-1:0]data_out;

  logic [Data_Width-1:0] fifo [0: Depth-1];

always_ff@(posedge wr_clk)
begin
  if(wr_en & !full)
begin
    /**** WRITE ****/
  fifo[wr_addr[Addr_Width-1:0]]<=data_in; /* Write operation */
end
end
  assign data_out=fifo[rd_addr[Addr_Width-1:0]]; /**READ OPERATION**/
  //assign data_out = 1; /*error injection*/
endmodule
```

Read Synchronizer:



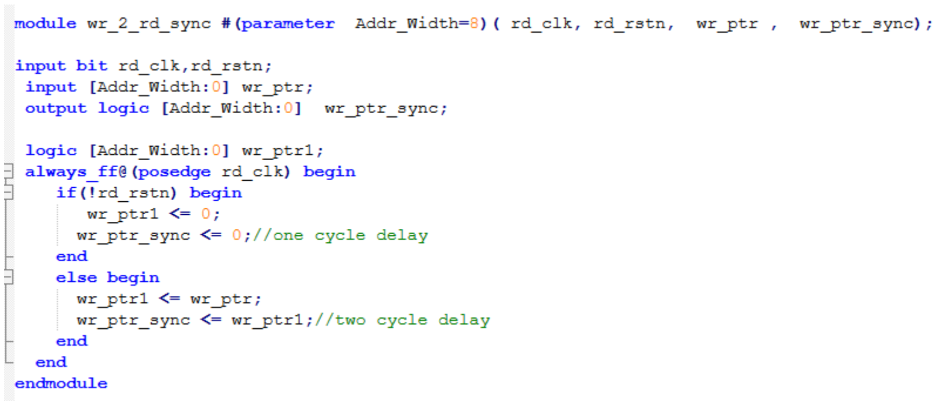
```
module rd_2_wr_sync #(parameter Addr_Width=8)( wr_clk, wr_rstn, rd_ptr,  rd_ptr_sync);

input bit wr_clk, wr_rstn;
  input logic [Addr_Width:0]  rd_ptr;
  output logic [Addr_Width:0] rd_ptr_sync;

  logic [Addr_Width:0] rd_ptr1;
  always_ff@(posedge wr_clk) begin
    if(!wr_rstn) begin
      rd_ptr1 <= 0;
      rd_ptr_sync <= 0;
    end
    else begin
      rd_ptr1 <= rd_ptr;
      rd_ptr_sync <= rd_ptr1;
    end
  end
endmodule
```

Read Pointer:



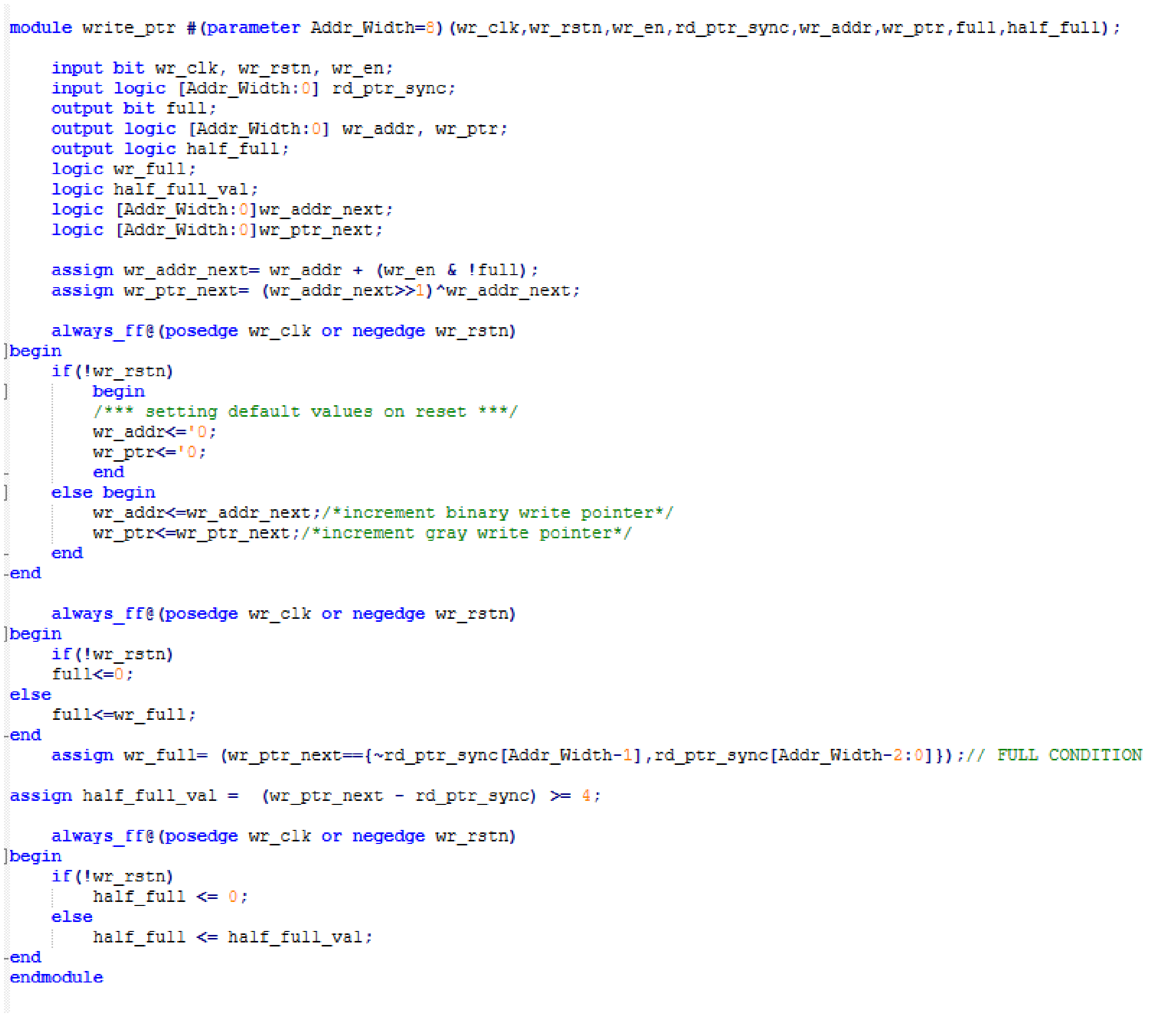
```
/************************************************************************/
module read_ptr #(parameter Addr_Width=8)(rd_clk,rd_rstn,rd_en,wr_ptr_sync,rd_addr,rd_ptr,empty,half_empty);

    input bit rd_clk,rd_rstn, rd_en;
    input logic [Addr_Width:0]  wr_ptr_sync;
    output bit empty;
    output logic [Addr_Width:0] rd_addr, rd_ptr;
    output logic half_empty;


    logic rd_empty;
    logic half_empty_val;
    logic [Addr_Width:0] rd_addr_next;
    logic [Addr_Width:0] rd_ptr_next;


    assign rd_addr_next= rd_addr + (rd_en & !empty);
    assign rd_ptr_next=(rd_addr_next>>1)^rd_addr_next; /*Binary to Gray conversion*/
    assign rd_empty= (wr_ptr_sync == rd_ptr_next); /* empty check */

    always_ff@(posedge rd_clk or negedge rd_rstn)
begin
    if(!rd_rstn)
        begin
        rd_addr<=0;
        rd_ptr<=0;
        end
    else begin
        rd_addr<=rd_addr_next;/* increment binary read pointer value */
        rd_ptr<=rd_ptr_next;/* Increment gray read pointer */
    end
end

    always_ff@(posedge rd_clk or negedge rd_rstn)
begin
if(!rd_rstn)
    empty<=1;
else
    empty<=rd_empty;

end

assign half_empty_val=(wr_ptr_sync - rd_ptr_next) < 4;
    always_ff@(posedge rd_clk or negedge rd_rstn)
begin
    if(!rd_rstn)
        half_empty <= 1'b1;
    else
        half_empty <= half_empty_val;
end
endmodule
```

Write Synchronizer:



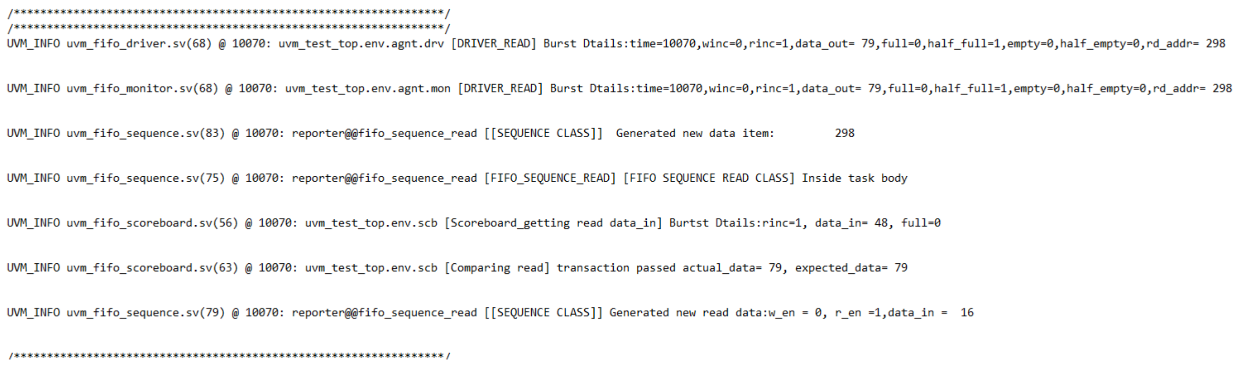
```
module wr_2_rd_sync #(parameter  Addr_Width=8)( rd_clk, rd_rstn,  wr_ptr ,  wr_ptr_sync);

input bit rd_clk,rd_rstn;
 input [Addr_Width:0] wr_ptr;
 output logic [Addr_Width:0]  wr_ptr_sync;

 logic [Addr_Width:0] wr_ptr1;
 always_ff@(posedge rd_clk) begin
    if(!rd_rstn) begin
       wr_ptr1 <= 0;
      wr_ptr_sync <= 0;//one cycle delay
    end
    else begin
      wr_ptr1 <= wr_ptr;
      wr_ptr_sync <= wr_ptr1;//two cycle delay
    end
  end
endmodule
```

Write Pointer:



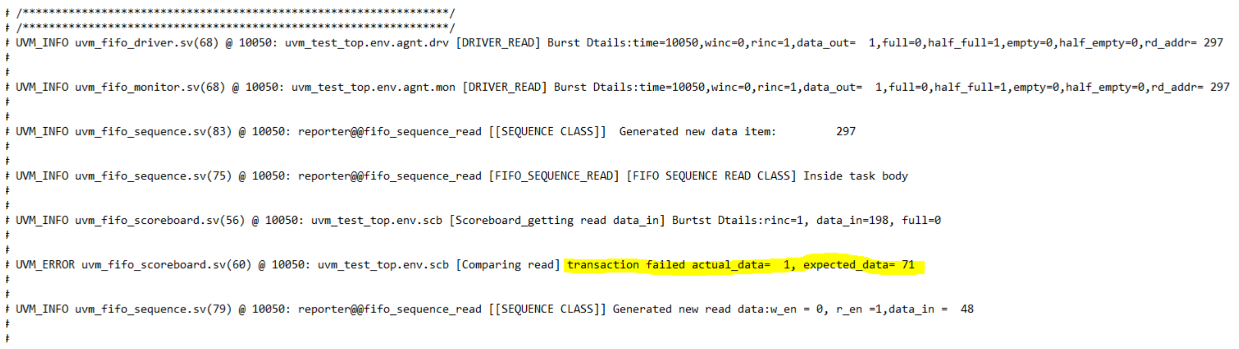
```
module write_ptr #(parameter Addr_Width=8)(wr_clk,wr_rstn,wr_en,rd_ptr_sync,wr_addr,wr_ptr,full,half_full);

    input bit wr_clk, wr_rstn, wr_en;
    input logic [Addr_Width:0] rd_ptr_sync;
    output bit full;
    output logic [Addr_Width:0] wr_addr, wr_ptr;
    output logic half_full;
    logic wr_full;
    logic half_full_val;
    logic [Addr_Width:0]wr_addr_next;
    logic [Addr_Width:0]wr_ptr_next;

    assign wr_addr_next= wr_addr + (wr_en & !full);
    assign wr_ptr_next= (wr_addr_next>>1)^wr_addr_next;

    always_ff@(posedge wr_clk or negedge wr_rstn)
begin
    if(!wr_rstn)
        begin
        /*** setting default values on reset ***/
        wr_addr<='0;
        wr_ptr<='0;
        end
    else begin
        wr_addr<=wr_addr_next;/*increment binary write pointer*/
        wr_ptr<=wr_ptr_next;/*increment gray write pointer*/
    end
end

    always_ff@(posedge wr_clk or negedge wr_rstn)
begin
    if(!wr_rstn)
    full<=0;
else
    full<=wr_full;
end
    assign wr_full= (wr_ptr_next=={~rd_ptr_sync[Addr_Width-1],rd_ptr_sync[Addr_Width-2:0]});// FULL CONDITION

assign half_full_val =  (wr_ptr_next - rd_ptr_sync) >= 4;

    always_ff@(posedge wr_clk or negedge wr_rstn)
begin
    if(!wr_rstn)
        half_full <= 0;
    else
        half_full <= half_full_val;
end
endmodule
```

4. Simulation and Results

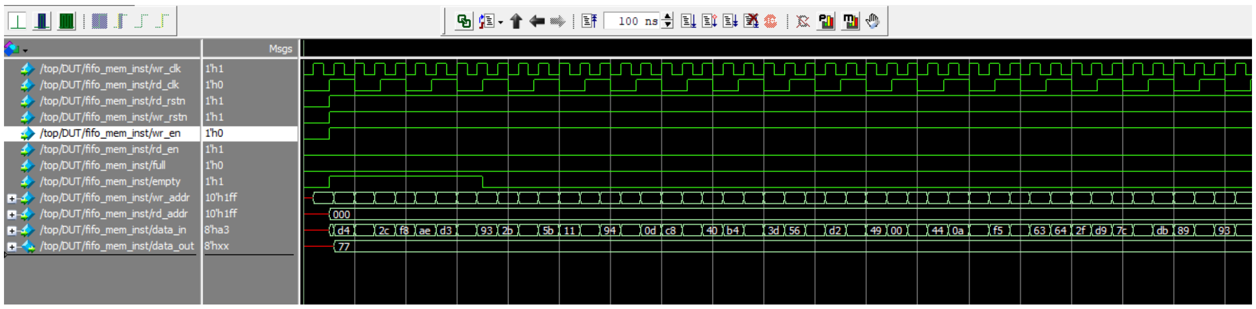


```
/*******************************************************************/
/*******************************************************************/
UVM_INFO uvm_fifo_driver.sv(68) @ 10070: uvm_test_top.env.agnt.drv [DRIVER_READ] Burst Dtails:time=10070,winc=0,rinc=1,data_out= 79,full=0,half_full=1,empty=0,half_empty=0,rd_addr= 298


UVM_INFO uvm_fifo_monitor.sv(68) @ 10070: uvm_test_top.env.agnt.mon [DRIVER_READ] Burst Dtails:time=10070,winc=0,rinc=1,data_out= 79,full=0,half_full=1,empty=0,half_empty=0,rd_addr= 298


UVM_INFO uvm_fifo_sequence.sv(83) @ 10070: reporter@@fifo_sequence_read [[SEQUENCE CLASS]]  Generated new data item:         298


UVM_INFO uvm_fifo_sequence.sv(75) @ 10070: reporter@@fifo_sequence_read [FIFO_SEQUENCE_READ] [FIFO SEQUENCE READ CLASS] Inside task body


UVM_INFO uvm_fifo_scoreboard.sv(56) @ 10070: uvm_test_top.env.scb [Scoreboard_getting read data_in] Burtst Dtails:rinc=1, data_in= 48, full=0


UVM_INFO uvm_fifo_scoreboard.sv(63) @ 10070: uvm_test_top.env.scb [Comparing read] transaction passed actual_data= 79, expected_data= 79


UVM_INFO uvm_fifo_sequence.sv(79) @ 10070: reporter@@fifo_sequence_read [[SEQUENCE CLASS]] Generated new read data:w_en = 0, r_en =1,data_in =  16


/*******************************************************************/
```

Error injection:



```
# /****************************************************************/
# /****************************************************************/
# UVM_INFO uvm_fifo_driver.sv(68) @ 10050: uvm_test_top.env.agnt.drv [DRIVER_READ] Burst Dtails:time=10050,winc=0,rinc=1,data_out=  1,full=0,half_full=1,empty=0,half_empty=0,rd_addr= 297
#
#
# UVM_INFO uvm_fifo_monitor.sv(68) @ 10050: uvm_test_top.env.agnt.mon [DRIVER_READ] Burst Dtails:time=10050,winc=0,rinc=1,data_out=  1,full=0,half_full=1,empty=0,half_empty=0,rd_addr= 297
#
#
# UVM_INFO uvm_fifo_sequence.sv(83) @ 10050: reporter@@fifo_sequence_read [[SEQUENCE CLASS]]  Generated new data item:         297
#
#
# UVM_INFO uvm_fifo_sequence.sv(75) @ 10050: reporter@@fifo_sequence_read [FIFO_SEQUENCE_READ] [FIFO SEQUENCE READ CLASS] Inside task body
#
#
# UVM_INFO uvm_fifo_scoreboard.sv(56) @ 10050: uvm_test_top.env.scb [Scoreboard_getting read data_in] Burtst Dtails:rinc=1, data_in=198, full=0
#
#
# UVM_ERROR uvm_fifo_scoreboard.sv(60) @ 10050: uvm_test_top.env.scb [Comparing read] transaction failed actual_data=  1, expected_data= 71
#
#
# UVM_INFO uvm_fifo_sequence.sv(79) @ 10050: reporter@@fifo_sequence_read [[SEQUENCE CLASS]] Generated new read data:w_en = 0, r_en =1,data_in =  48
#
#
```



```
                                     Msgs
/top/DUT/fifo_mem_inst/wr_clk        1'h1
/top/DUT/fifo_mem_inst/rd_clk        1'h0
/top/DUT/fifo_mem_inst/rd_rstn       1'h1
/top/DUT/fifo_mem_inst/wr_rstn       1'h1
/top/DUT/fifo_mem_inst/wr_en         1'h0
/top/DUT/fifo_mem_inst/rd_en         1'h1
/top/DUT/fifo_mem_inst/full          1'h0
/top/DUT/fifo_mem_inst/empty         1'h1
/top/DUT/fifo_mem_inst/wr_addr       10'h1ff
/top/DUT/fifo_mem_inst/rd_addr       10'h1ff
/top/DUT/fifo_mem_inst/data_in       8'ha3
/top/DUT/fifo_mem_inst/data_out      8'hxx
```

5.Conclusion

In this paper, the basic of asynchronous FIFO is designed and implemented by Verilog. This design is vital for applications requiring robust and high-speed data communication across different clock domains. Using Verilog HDL, the asynchronous FIFO design that is presented efficiently handles clock domain crossings. Synchronization of FIFO pointers into the opposite clock domain is safely accomplished using Gray code pointers. When the data in the asynchronous FIFO reaches or approaches the maximum depth of double- port RAM, Full signal is generated.

Reference

[1] Sunburst Design, "Simulation and Synthesis Techniques for Asynchronous FIFO Design".

[2] IEEE Xplore, "Implementation of an RTL synthesizable asynchronous FIFO"

[3] Putta Satish, "FIFO Depth Calculation Made Easy,"

[4] Professor Slides & sample Academic paper

[5]Open AI, "Chat GPT," [Online]