Implementation of Asynchronous FIFO Synchronizer for Clock-Domain Crossing using Verilog HDL

Personal Technical Project
(GitHub Repo)

Mohammad Zahid
f20220663@hyderabad.bits-pilani.ac.in

Date of Completion: June 2025

Abstract

This report details the design and implementation of an Asynchronous First-In, First-Out (FIFO) Synchronizer, a crucial component for reliable data transfer between digital systems operating across independent clock domains. The design utilizes a 8x8 dual-port synchronous RAM for data storage, managed by independent read and write control units. To ensure data integrity and prevent metastability issues during pointer synchronization, Gray codes are employed for the read/head and write/tail pointers, and a dedicated Brute-Force Synchronizer module (BFsync) is incorporated for cross-clock domain communication. The system features separate clock inputs for writing (`clkin`) and reading (`clkout`), along with independent reset signals (`rstin`, `rstout`). The `write_ctrl` module handles data input, manages the `tail` pointer and evaluates for FIFO full condition, while the `read_ctrl` module oversees data output, evaluates for FIFO empty condition and manages the `head` pointer. This asynchronous FIFO design provides a robust and efficient solution for inter-clock domain data transfer, essential for complex digital system architectures.

Table of Contents

3. Introduction

3.1 Purpose of the Report

This report documents the design, implementation, and verification of an asynchronous First-In, First-Out (FIFO) synchronizer. It details the architectural choices, theoretical underpinnings, and practical considerations involved in creating a robust solution for data transfer across different clock domains

3.2 Problem Statement

In complex digital systems, such as System-on-Chip (SoC) designs, it is common to have multiple functional blocks operating at different clock frequencies or even entirely asynchronous to one another. When data needs to be transferred between these independently clocked domains, we have to deal with the issue of Clock Domain Crossing (CDC). Direct transfer of signals can lead to metastability, an unstable state in flip-flops where the output is neither a definite logic high nor low, potentially causing unpredictable system behavior and data corruption.

3.3 Importance of Asynchronous FIFOs

Properly designed Asynchronous FIFOs prove to be a better solution than Handshake-Synchronizer, MUX-Synchronizer and other solutions specially when data correctness and high bandwidth are important design requirements. They act as buffers that decouple the read and write operations, allowing data to be written into the FIFO using one clock and read out using another, completely independent clock. This buffering capability is crucial for accommodating temporary rate mismatches between the clock domains, ensuring that data is neither lost nor duplicated during the transfer process.

3.4 Project Overview

This project focuses on the design of an 8-bit asynchronous FIFO synchronizer. The core of the design utilizes a dual-port synchronous RAM for efficient data storage. To manage the asynchronous read and write operations, the FIFO employs independent control logic for its head and tail pointers. A key aspect of this design is the use of Gray codes for these pointers, which, when synchronized across clock domains using dedicated synchronizer circuits, mitigate the risk of metastability and ensure accurate full/empty status detection. The report will detail the individual modules, their interactions, and the verification methodology employed to validate the FIFO's functionality under various asynchronous conditions.

4. Background and Theory

4.1 Clock Domain Crossing (CDC)

In complex designs like System-on-Chips (SoCs), it is often necessary to integrate multiple functional blocks that operate at different clock frequencies or even entirely without a common clock (asynchronous). While sampling asynchronous signals or crossing clock domains, there is a high probability that setup and hold time constraints will not be met in the sampling clock domain. This will lead to the system entering a metastable state and producing irreversible errors.

Metastability occurs when a flip-flop's setup or hold time requirements are violated, causing its output to enter an unstable, intermediate voltage level for an unpredictable duration. If this metastable state persists long enough to be sampled by a downstream flip-flop, it can propagate as an unknown logic value, leading to erroneous operation, data corruption, or even system failure.

Resolving metastability typically involves allowing sufficient time for the flip-flop to settle into a stable state before its output is sampled. The probability of taking longer than waiting time t_w for a flip-flop output to exit metastable/illegal state is given by:

$$P = \exp(-t_w / \boldsymbol{\tau}_s)$$

Where $\boldsymbol{\tau}_s$ is the time constant for the CMOS technology in use.

Basic synchronization techniques often involve using a series of two or more flip-flops (e.g., a two-flip-flop synchronizer) to re-clock the incoming signal into the new clock domain, giving the signal at least one clock period of waiting time to settle and thereby significantly reducing the probability of metastability.

4.2 FIFO Basics

A First-In, First-Out (FIFO) buffer is a fundamental data structure used in digital design to temporarily store data. As its name suggests, data written into a FIFO is read out in the same order it was written, much like a queue. FIFOs are essential for buffering data, accommodating bursty data transfers, and acting as rate adapters between components.

There are two primary types of FIFOs:

- Synchronous FIFOs: Both read and write operations are controlled by the same clock signal. These are simpler to design as they do not involve clock domain crossing issues.

- Asynchronous FIFOs: Read and write operations are controlled by independent clock signals. These are more complex due to the need for robust synchronization mechanisms to handle the distinct clock domains, but they are vital for inter-domain communication.

4.3 Asynchronous FIFO Principles

An asynchronous FIFO enables data transfer between two circuits operating on different clocks. It achieves this by using a shared memory (typically a dual-port RAM) and separate control logic for the read and write sides.
The core principle involves:

- Write Side: Data is written into the FIFO's memory using a write pointer (tail), which increments with each write operation. The write logic also monitors the state of the read pointer (from the read clock domain, synchronized to the write domain) to determine if the FIFO is full.

- Read Side: Data is read from the FIFO's memory using a read pointer (head), which increments with each read operation. The read logic monitors the state of the write pointer (from the write clock domain, synchronized to the read domain) to determine if the FIFO is empty.

The critical challenge lies in safely transferring the read pointer from the read clock domain to the write clock domain, and the write pointer from the write clock domain to the read clock domain, without introducing metastability or misinterpreting the pointer values.

4.4 Gray Code for Pointer Synchronization

To ensure reliable full/empty status detection and prevent race conditions when pointers cross clock domains, Gray codes are used for the read and write pointers. A Gray code is a binary numeral system where two successive values differ in only one bit.

- When a binary counter increments, multiple bits can change simultaneously (e.g., 011 to 100). If these bits are sampled asynchronously, some bits might be seen as changed while others are not, leading to an incorrect sampled value.

- With Gray code, only one bit changes at a time during an increment. When this single-bit change is synchronized across clock domains using a Brute-Force Two-flip-flop synchronizer, the risk of sampling an intermediate, invalid value is drastically reduced.

- Even if metastability occurs on the single changing bit, it is much more likely to resolve to the correct new value or the correct old value, rather than an entirely different, erroneous pointer value. This ensures that the synchronized pointer value is always a valid Gray code, preventing the FIFO from incorrectly indicating full or empty conditions.

4.5 Flow Control - Full and Empty Conditions

The proper operation of an asynchronous FIFO heavily relies on accurately detecting its full and empty states to prevent data overflow (writing to a full FIFO) or underflow (reading from an empty FIFO). These conditions are determined by comparing the synchronized read and write pointers.

For a 2^N depth FIFO we use N-bit Gray code pointers:

- Empty/Underflow Condition: The FIFO is empty when the synchronized write pointer (in the read domain) is equal to the read pointer (in the read domain). We evaluate for empty condition in the read domain in order to prevent output of invalid data. In simpler terms, if tail_o == head, the FIFO is empty from the perspective of the read side.

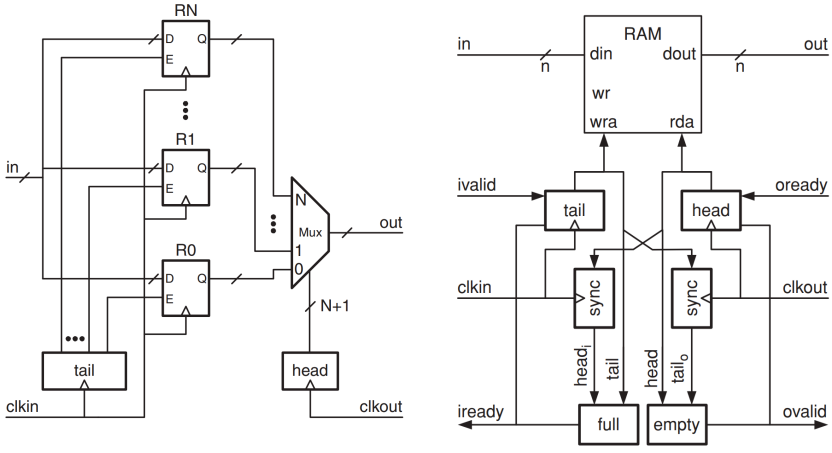
- Full/Overflow Condition: The full condition is detected when the synchronized read pointer in the write domain (head_i) is equal to the incremented value of the current write pointer (inc_tail). We evaluate for full condition in the write domain in order to prevent overwriting valid data present in FIFO. In simpler terms, if inc_tail == head, the FIFO is empty from the perspective of the write side.

5. Design Methodology

5.1 Overall Architecture

The asynchronous FIFO synchronizer's top-level module, `asyncfifo`, orchestrates the interaction between several key sub-modules to achieve reliable data transfer between independent clock domains. As shown in Figure 1 , the `asyncfifo` module receives data (`din`) and a write enable (`ivalid`) on the input clock domain (`clkin`), and outputs data (`dout`) with a read enable (`ovalid`) on the output clock domain (`clkout`). Independent reset signals (`rstin`, `rstout`) are also provided for each domain.

The core data storage is handled by a `dp_ram` (dual-port RAM). Write operations are controlled by the `write_ctrl` module, which manages the `tail` (write) pointer, while read operations are controlled by the `read_ctrl` module, managing the `head` (read) pointer. Crucially, the `BFsync` modules are used to safely synchronize the `head` pointer from the read domain to the write domain (`head_i`) and the `tail` pointer from the write domain to the read domain (`tail_o`). These synchronized pointers are then used by the respective control modules to determine the FIFO's full (`iready`) and empty (`ovalid`) states, preventing overflow and underflow.



(a) Data-path (b) Control-Path

Figure 1: High-Level Block Diagram of Asynchronous FIFO Synchronizer (Adapted from *Digital Design: A Systems Approach* by Bill Dally and R. Curtis Harting, Section 29.4)

5.2 Module-Level Design

5.2.1 dp_ram (Dual-Port RAM)

Functionality: This module implements an 8x8 memory array, allowing simultaneous read and write operations. It features a synchronous write port and an asynchronous read port. Data is written into the memory on the positive edge of the clk signal when the write enable is asserted. Data is combinatorially read from the memory based on the raddr input, meaning the rdata output immediately reflects the content of the addressed location.

Inputs: clk (write clock), waddr (write address), raddr (read address), write (write enable), wdata (data to write).

Output: rdata (data read).

5.2.2 write_ctrl (Write Control Logic)

Functionality: This module manages the write operations into the FIFO. It tracks the tail (write) pointer and generates the iready signal, checking whether the FIFO is not full and ready to accept new data.

Inputs: clkin (write domain clock), rstin (write domain reset), head_i (read pointer synchronized to the write domain), ivalid (input data valid).

Outputs: iready (write ready or FIFO not full), tail (current write pointer).

Logic: It increments the tail pointer on each successful write operation. The iready signal is asserted as long as the FIFO is not full, which is determined by comparing the head_i with the incremented tail pointer.

5.2.3 `read_ctrl` (read Control Logic)

Functionality: This module manages the read operations from the FIFO. It tracks the head (read) pointer and generates the ovalid signal, indicating whether valid data is available to be read from the FIFO (i.e., not empty).

Inputs: clkout (read clock), rstout (read-side reset), tail_o (write pointer synchronized to the read domain), oready (output ready/receiver ready).

Outputs: ovalid (output valid or FIFO not empty), head (current read pointer).

Logic: It increments the head pointer on each successful read operation. The ovalid signal is asserted when the FIFO is not empty, which is determined by comparing the head pointer with the synchronized tail_o pointer.

5.2.4 BFsync (Brute-Force Synchronizer)

Functionality: This module is a two-flip-flop synchronizer, used for safely transferring asynchronous signals (Gray-coded pointers) between clock domains. Its primary role is to reduce the probability of metastability.

Internal Ciruit: It consists of two cascaded D-flip-flops clocked by the destination clock. The first flip-flop captures the asynchronous input, and the second flip-flop re-clocks the output of the first, providing a synchronized and stable signal to the destination domain.

Usage in FIFO: Instantiated twice: one to synchronize the head pointer from the clkout domain to the clkin domain (head2in), and another to synchronize the tail pointer from the clkin domain to the clkout domain (tail2out).

5.2.5 gray_count Module (Pointer Incrementer)

Functionality: This module is used to increment the Gray Code Pointers. It takes a Gray code input, converts it to binary, increments the binary value, and then converts the incremented binary back to Gray code for output.

5.3 Design Considerations and Choices

5.3.1 FIFO RAM (dp_ram.v)

- The memory block implemented is an 8x8 synchronous write, asynchronous read dual-port RAM. This means data is written into the memory on the positive edge of the input/sender clock (clkin), while data can be read combinatorially from any address at any time.
- This specific type of RAM was chosen as it aligns with common asynchronous FIFO design patterns and was a direct reference from "Digital Design Using VHDL: A Systems Approach".
- While other approaches, such as fully asynchronous dual-port RAMs that latch data, exist (as seen in University of Oslo IN3160/IN4160 lecture slides), the synchronous write/asynchronous read model offered a straightforward and effective solution for the initial implementation.
- Future work may explore the fully asynchronous RAM for comparison.

5.3.2 Full and Empty Logic

The implemented full and empty logic utilizes 3-bit write (tail) and read (head) pointers.

- The empty condition is detected when the write pointer (tail) is equal to the synchronized read pointer in the write domain (head_i).
- The full condition is detected when the synchronized read pointer in the write domain (head_i) is equal to the incremented value of the current write pointer (inc_tail).
- As a result a maximum of seven locations in the memory can be filled, at least one location will always remain unused.

This "one-empty-location" scheme is a simple and reliable method for asynchronous FIFO control. A known drawback of this approach is that one RAM location will always remain unused to distinguish between the full and empty states. My primary objective was to achieve a functional and understandable solution, leading to the selection of this simpler logic.

Alternatives Considered:

- (N+1)-bit Pointer Scheme: As described by Clifford E. Cummings in his paper, this method expands the pointer size to (N+1) bits for a 2^N depth FIFO. The additional bit is used to differentiate between full and empty conditions when the lower N bits of the pointers are equal. This allows for the utilization of all 2N memory locations, maximizing memory efficiency.
- State Flip-Flops: Another approach involves having one or more dedicated flip-flops in each clock domain to track the FIFO's state (e.g., empty, full). Control signals (iready, ivalid, oready, ovalid) are then used to transition between these states. This method can sometimes offer more flexibility in state management

At the moment, my main objective was to implement a simple and working solution, hence the decision to go with the current alternative. The exploration and implementation of these alternative full/empty detection schemes are considered for future work.

5.3.3 Pointer Synchronization

The selection of Gray code for pointers in conjunction with the BFsync (two-flip-flop) synchronizer is a fundamental design choice to minimize metastability. This combination provides a robust yet also simple solution for handling clock domain crossings of the pointers.

5.3.4 FIFO Depth

The FIFO is designed with 3-bit pointers, which corresponds to a total of 2^3=8 memory locations. This depth was chosen to demonstrate the core principles of asynchronous FIFO design with a manageable memory size for educational purposes and initial verification.

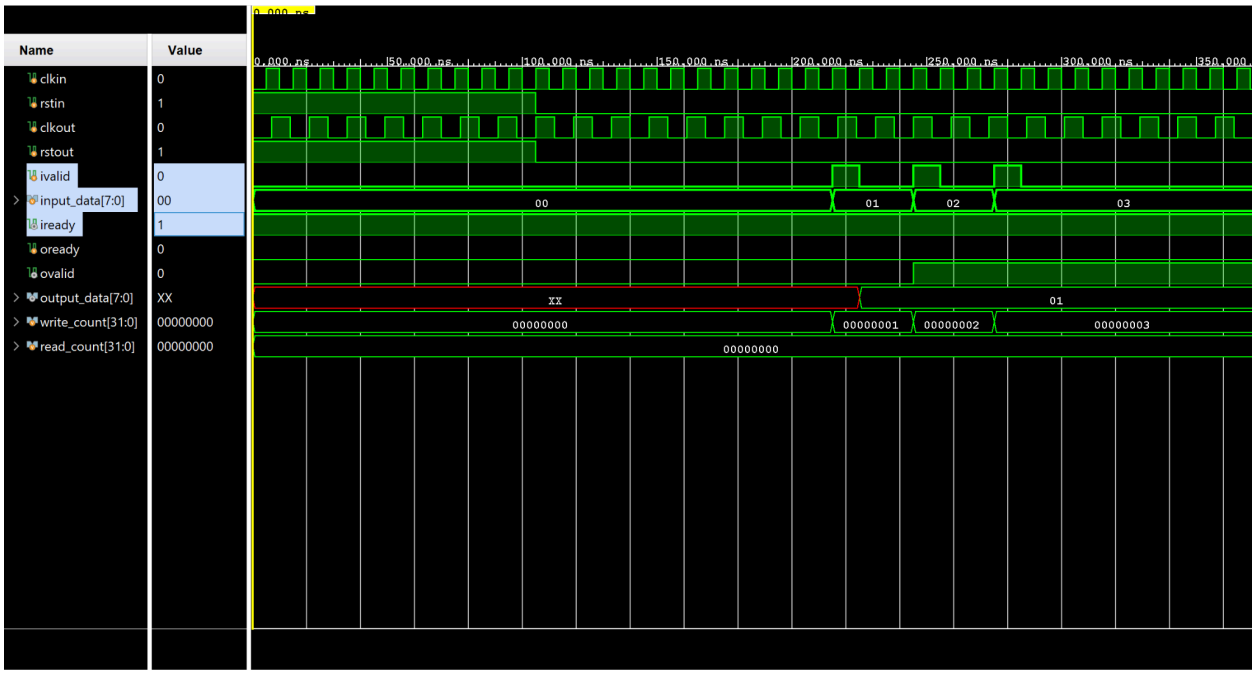
5.3.5 Reset Strategy

Asynchronous reset signals (rstin for the write side and rstout for the read side) are employed. This ensures that both sides of the FIFO can be independently reset, bringing their respective pointers to a known initial state regardless of the clock signal, which is important for robust system initialization.
However in case of synchronised pointer, reset signal needs to be given time to propagate. The 2-FF synchronizer introduces a 2 clock cycle delay in the resetting of synchronised pointers.

6. Results and Waveforms

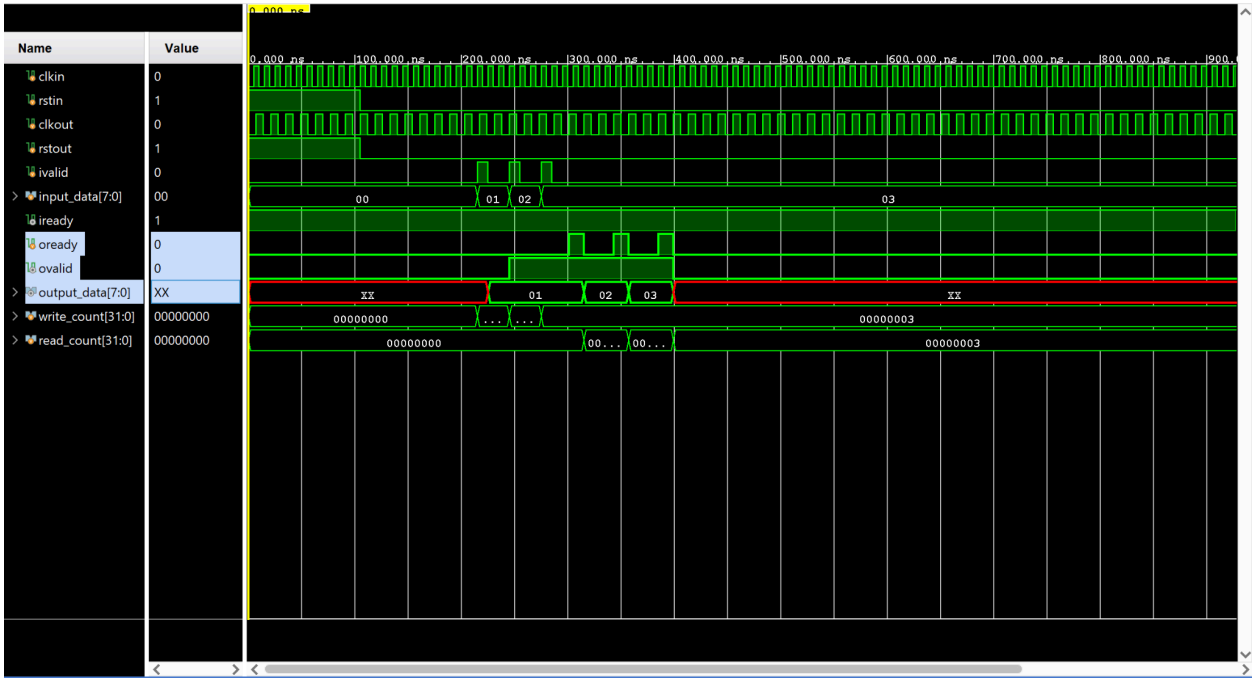
6.1 Writing Data into Initialized FIFO

- The FIFO is initialized by asserting (rstin, rstout) and deasserting the control signals (ivalid, oready). Reset signals have to be given time to propagate through the synchronization stage.
- To perform write operation, ivalid signal is asserted and input is given to the RAM module.
- Since FIFO has not been filled yet, iready signal remains high.
- After the first data is written, ovalid signal goes high indicating FIFO is no longer empty and valid data is available to be sampled.
- The ovalid signal goes high after a latency of two clock periods in the read domain. This latency is due to the synchronization of write pointer into the read domain.



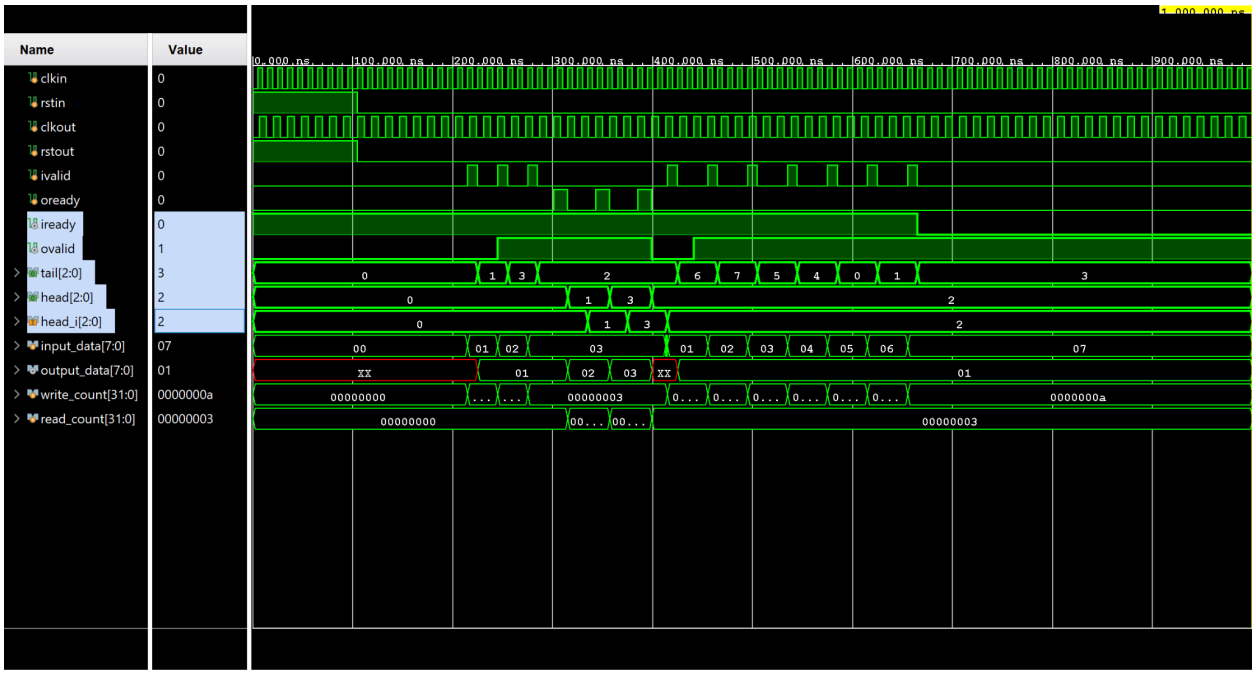
6.2 Reading Data from FIFO

- To initiate the reading process, oready is asserted to indicate that data is ready to be captured.
- After all valid data has been read, FIFO is empty and ovalid signal indicates the same by going low.



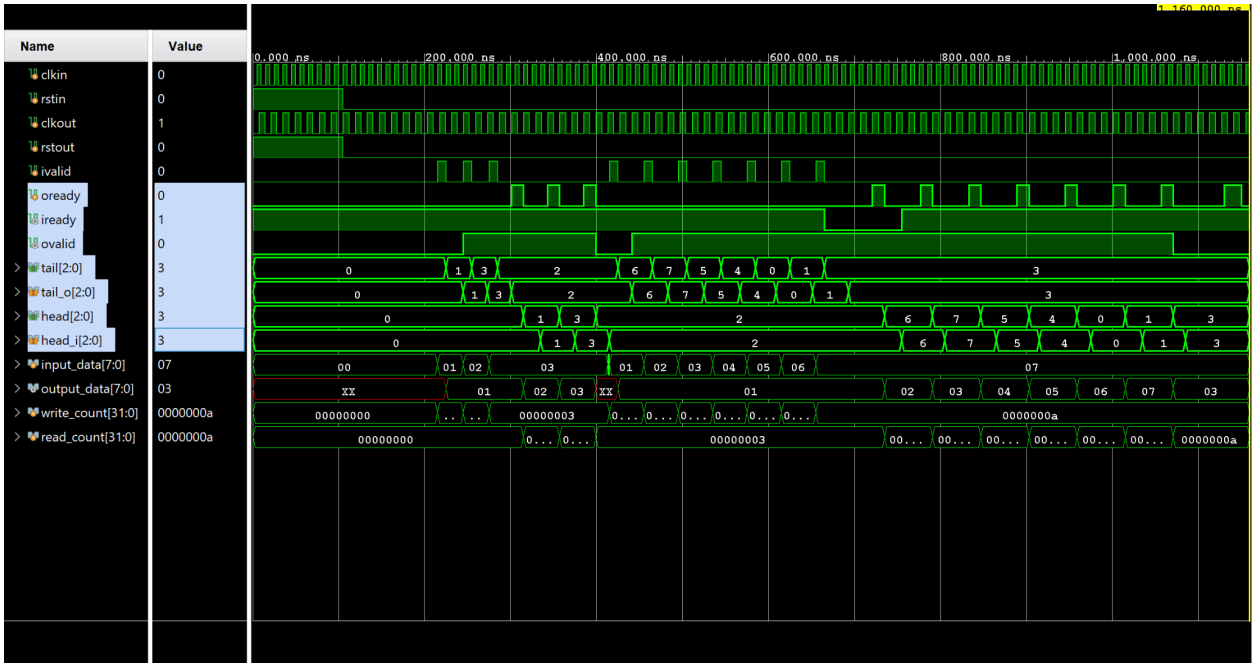
6.3 Filling FIFO to Capacity

- FIFO filled to capacity by giving 7 data inputs.
- iready signal goes low to indicate the full status.
- Full status is evaluated true since (tail + 1 == head_i). (Gray Code addition)
- To indicate FIFO full, iready signal deasserts preventing any more writing from taking place.



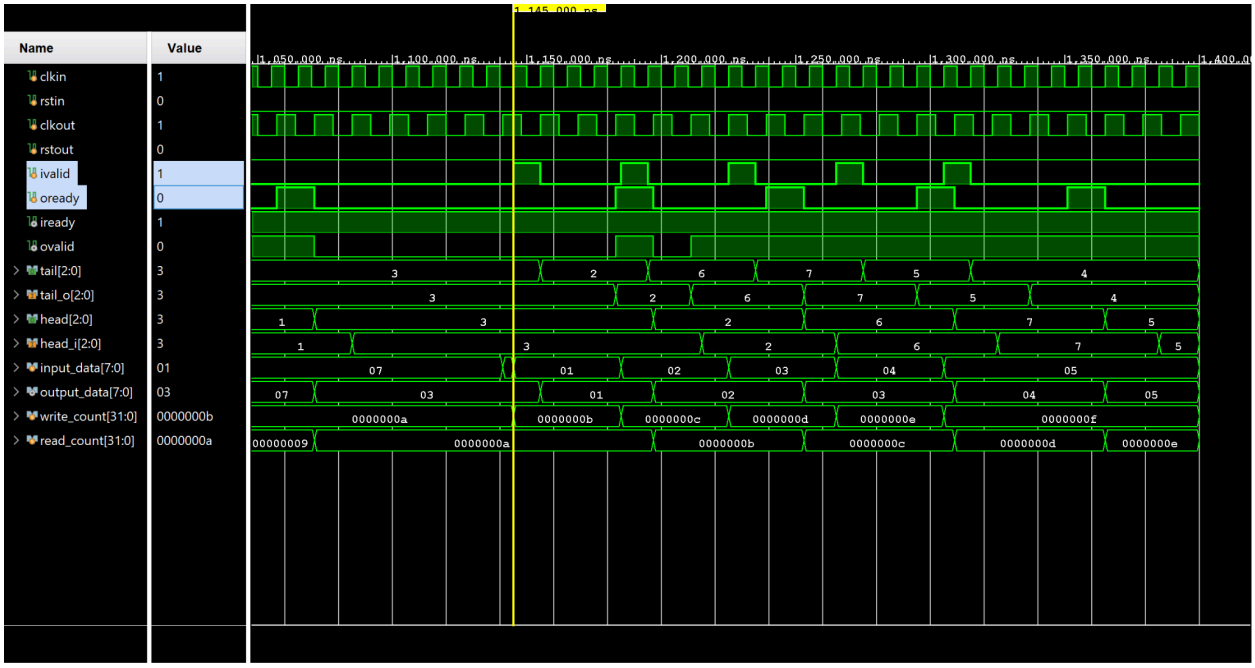
6.4 Emptying FIFO

- FIFO is emptied by applying oready pulse.
- Once all 7 data items are received as outputs, (tail_o == head) condition is satisfied and ovalid goes low to indicate empty condition.
- Further application of oready pulse doesn't trigger any reading.



6.5 Concurrent Read and Write Operations

- FIFO demonstrates robust operation when concurrent read and write requests are sent.
- Data order is preserved while sampling outputs.



7. Conclusion

This project successfully designed and implemented a functional **Asynchronous First-In, First-Out (FIFO) synchronizer** using Verilog HDL. The core objective of enabling reliable data transfer between independent clock domains was achieved by employing a dual-port RAM for storage, coupled with independent read and write control logic. A critical aspect of the design was the strategic use of **Gray-coded pointers** and **two-flip-flop synchronizers (BFsync modules)** to effectively mitigate metastability risks inherent in clock domain crossings, ensuring robust and error-free pointer synchronization.

Through rigorous simulation, the FIFO's functionality was validated across various scenarios, demonstrating correct data buffering, accurate detection of full and empty conditions, and reliable operation under asynchronous clock relationships. This project has significantly enhanced our understanding of fundamental digital design principles, particularly in handling complex clock domain crossing challenges, and has provided valuable hands-on experience in designing and verifying a critical component for modern SoC architectures. The successful implementation of this asynchronous FIFO serves as a strong foundation for exploring more advanced synchronization techniques and larger-scale digital system designs.

8. References

1. Dally, W. J., & Harting, R. C. (2012). *Digital Design: A Systems Approach*. Cambridge University Press.

2. University of Oslo. (2021) *IN3160/IN4160 Lecture Slides on Clock Domain Crossing*. Retrieved from https://www.uio.no/studier/emner/matnat/ifi/IN3160/v21/timeplan/in3160-l92-clock-domains.pdf

3. Cummings, C. E. (2002). *Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs*. Sunburst Design, Inc. Retrieved from http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf

Appendix A: Complete Verilog Code

asyncfifo.v

```
module asyncfifo(
    input clkin, clkout,
    input rstin, rstout,
    input ivalid, oready,
    input [7:0] din,
    output iready, ovalid,
    output [7:0] dout
    );

    wire [2:0] tail, next_tail, head, next_head;
    wire [2:0] inc_head, inc_tail;
    wire [2:0] head_i, tail_o;

    dp_ram ram(
        .clk(clkin),
        .wdata(din),
        .waddr(tail),
        .raddr(head),
        .write(iready && ivalid),
        .rdata(dout)
    );

    BFsync head2in(
        .d(head),
        .clk(clkin),
        .rst(rstin),
        .q(head_i)
        );

    BFsync tail2out(
        .d(tail),
        .clk(clkout),
        .rst(rstout),
        .q(tail_o)
        );
```

```
    write_ctrl wr_ctrl(
        .clkin(clkin),
        .rstin(rstin),
        .head_i(head_i),
        .ivalid(ivalid),
        .iready(iready),
        .tail(tail)
    );

    read_ctrl re_ctrl(
        .clkout(clkout),
        .rstout(rstout),
        .tail_o(tail_o),
        .oready(oready),
        .ovalid(ovalid),
        .head(head)
    );

endmodule
```

dp_ram.v

```
module dp_ram(
    input clk,
    input [2:0] waddr,raddr,
    input write,
    input [7:0] wdata,
    output [7:0] rdata
    );
    reg [7:0] memory [7:0];

    assign rdata = memory[raddr];

    always@(posedge clk) begin
        if (write) begin
            memory[waddr] <= wdata;
```

```
        end
    end

endmodule
```

write_ctrl.v

```
module write_ctrl(
    input clkin, rstin,
    input [2:0] head_i,
    input ivalid,
    output reg iready,
    output reg [2:0] tail
    );

    //tail pointer update
    reg [2:0] next_tail;
    always@(posedge clkin or posedge rstin) begin
        if (rstin) tail <= 3'b000;
        else tail <= next_tail;
    end

    //tail pointer incrementer
    wire [2:0] inc_tail;
    gray_count cnt_tail(.gin(tail), .gout(inc_tail));


    //logic for checking full condition
    always @(*) begin
        iready = (head_i != inc_tail);
    end

    //tail pointer update logic
    always @(*) begin
        if (rstin) next_tail <= 3'b000;
        else if (iready && ivalid) next_tail <= inc_tail;
//if valid data at input and FIFO not full -> write and increment tail
        else next_tail <= tail;
```

```
    end

endmodule
```

read_ctrl.v

```
module read_ctrl(
    input clkout, rstout,
    input oready,
    input [2:0] tail_o,
    output reg ovalid,
    output reg [2:0] head
    );

    //head pointer update
    reg [2:0] next_head;
    always @(posedge clkout or posedge rstout) begin
        if (rstout) head <= 3'b000;
        else head <= next_head;
    end

    //head pointer incrementer
    wire [2:0] inc_head;
    gray_count cnt_head(.gin(head), .gout(inc_head));

    //checking for empty condition
    always@(*) begin
        ovalid = (tail_o != head);
    end
    //head pointer empty logic
    always @(*) begin
        if (rstout) next_head <= 3'b000;
        else if (ovalid && oready) next_head <= inc_head;
//if output line ready and FIFO not empty -> read and increment head
        else next_head <= head;
    end

endmodule
```

BFsync.v

```
module BFsync(
    input [2:0] d,
    input clk, rst,
    output reg [2:0] q
    );

    reg [2:0] qi;
    always@(posedge clk or posedge rst) begin
        if (rst) begin
            q <= 3'b000;
            qi<= 3'b000;
        end
        else {q,qi} <= {qi,d};
    end

Endmodule
```

gray_count.v

```
module gray_count(
    input [2:0] gin,
    output [2:0] gout
    );

    wire [2:0] bin, bin_inc;
    // gray to bin
    assign  bin[2] = gin[2],
            bin[1] = gin[1] ^ bin[2],
            bin[0] = gin[0] ^ bin[1];
    assign bin_inc = bin + 1;
    //bin to gray
    assign gout = bin_inc ^ (bin_inc>>1);

endmodule
```

Appendix B: Testbench Code

```
module tb();

    reg clkin, rstin, clkout, rstout;
    reg ivalid, oready;
    wire iready, ovalid;
    reg [7:0] input_data;
    wire [7:0] output_data;

    //test metrics
    integer write_count = 0;
    integer read_count = 0;
    integer error_count = 0;
    reg [7:0] expected_data = 8'h01;

    //instantiate FIFO
    asyncfifo dut (
        .clkin(clkin),
        .rstin(rstin),
        .clkout(clkout),
        .rstout(rstout),
        .ivalid(ivalid),
        .oready(oready),
        .iready(iready),
        .ovalid(ovalid),
        .din(input_data),
        .dout(output_data)
    );


    //clock generation
    initial begin
        clkin = 0;
        forever #5 clkin = ~clkin;  // 100MHz (10ns period)
    end

    initial begin
        clkout = 0;
```

```
    forever #7 clkout = ~clkout;  // ~71MHz (14ns period)
end

initial begin
    $display("=== Testbench Starting ===");
    $display("Input Clock: 100MHz, Output Clock: 71MHz");
    $display("");

    //initialization
    rstin = 1;
    rstout = 1;
    ivalid = 0;
    oready = 0;
    input_data = 0;

    repeat(5) @(posedge clkin);
    repeat(5) @(posedge clkout);

    $display("Time=%0t: releasing resets", $time);
    rstin = 0;
    rstout = 0;

    //reset propogation interval
    repeat(10) @(posedge clkin);

    write_operation();
    read_operation();
    reset_inputdata();
    write_full();
    read_till_empty();
    reset_inputdata();
    concurrent_readnwrite();
end

task write_operation;
    repeat(3) begin
        @(posedge clkin);
        wait (iready);
```

```
            input_data = input_data + 1;
            ivalid = 1'b1;
            write_count = write_count + 1;
            $display("time=%0t: writing data=0x%02h,
write_count=%0d",$time, input_data, write_count);
            @(posedge clkin);
            ivalid = 1'b0;
            @(posedge clkin);
        end
    endtask

    task read_operation;
        repeat(3) begin
            @(posedge clkout);
            wait (ovalid);
            oready = 1'b1;
            @(posedge clkout);
            read_count = read_count + 1;
            $display("time=%0t: reading data=0x%02h,
read_count=%0d",$time, output_data, read_count);
            oready = 1'b0;
            @(posedge clkout);
        end
    endtask

   task reset_inputdata;
        input_data = 0;
   endtask


    task write_full;
        repeat(8) begin
            @(posedge clkin);
            if (iready) begin
                input_data = input_data + 1;
                ivalid = 1'b1;
                write_count = write_count + 1;
```

```
                $display("time=%0t: writing data=0x%02h,
write_count=%0d",$time, input_data, write_count);
                @(posedge clkin);
                ivalid = 1'b0;
            end
            repeat(2) @(posedge clkin);
        end
    endtask

    task read_till_empty;
        repeat(8) begin
            @(posedge clkout);
            if (ovalid) begin
                oready = 1'b1;
                @(posedge clkout);
                read_count = read_count + 1;
                $display("time=%0t: reading data=0x%02h,
read_count=%0d",$time, output_data, read_count);
                oready = 1'b0;
            end
            repeat(2)@(posedge clkout);
        end
    endtask

    task concurrent_readnwrite;
        begin
            fork
                //write
                begin
                    repeat(5) begin
                        @(posedge clkin);
                        wait(iready);
                        input_data = input_data + 1;
                        ivalid = 1;
                        write_count = write_count + 1;
                        $display("Time=%0t: Concurrent write
data=0x%02h", $time, input_data);
                        @(posedge clkin);
```

```
                        ivalid = 0;
                        repeat(2) @(posedge clkin); //random delay
                    end
                end

                //read
                begin
                    repeat(5) begin
                        @(posedge clkout);
                        wait(ovalid);
                        oready = 1;
                        @(posedge clkout);
                        read_count = read_count + 1;
                        $display("Time=%0t: Concurrent read
data=0x%02h, expected=0x%02h",
                            $time, output_data, expected_data);
                        if (output_data !== expected_data) begin
                            $display("ERROR: Data mismatch in
concurrent test!");
                            error_count = error_count + 1;
                        end
                        expected_data = expected_data + 1;
                        oready = 0;
                        repeat(2) @(posedge clkout);
                    end
                end
            join
        end
    endtask

endmodule
```