High Level Design Specification (HLDS)

for

Asynchronous FIFO

Version 2.0

Prepared by Daniyal Ahmad, Fardeen Wasey, Adeel Ahmed

ECE-593: Fundamentals of Pre-Silicon Validation – Venkatesh Patil

02/11/2024

Table of Contents

Revision History

Name	Date	Reason For Changes	Version
Async FIFO	02/3/24	Implemented Design for Async FIFO and tested with some conventional test cases.	1.0
Async FIFO	02/11/24	Added UVM testbench with working interface and complete transaction between uvm_sequence and uvm_driver.	2.0

1. Introduction

1.1 Purpose

This High-Level Design Specification (HLDS) describes the implementation of the Async FIFO revision 2.0.

1.2 Document Conventions

Following fonts have special significance:

Bold (Times) is used for headings and subheadings.
Italics (Arial) is used for references and links, and *Italic* (Times) for footer and header.

1.3 Intended Audience and Reading Suggestions

This High-Level Design Specification (HLDS) is intended for Logic Design engineers looking for designing techniques and debugging of Async FIFO, Design Verification Engineers for writing effective test cases, Design Verification/Validation Manager for looking over the Design progress and proper verification plans and coverage closure.

1.4 Product Scope

This implementation of FIFO aims to develop an effective and highly accurate way of providing buffer between two different clock domains in such a way that the address which is written first will be the first to be read. It contains adjustable parameters which makes it scalable as per the design needs. It befits with its scalable design for different applications, high accuracy, efficient use of combination logics, flip flops, improved bandwidth and latency, usage of gray code counter for synchronization.

Some of the major objectives of this implementation are to make it scalable for most of the design and applications, have proper and efficient synchronization between different clocks of different domains so that the correct data is retrieved. So, the goal of the design is to achieve less resourceful and accurate data synchronization.

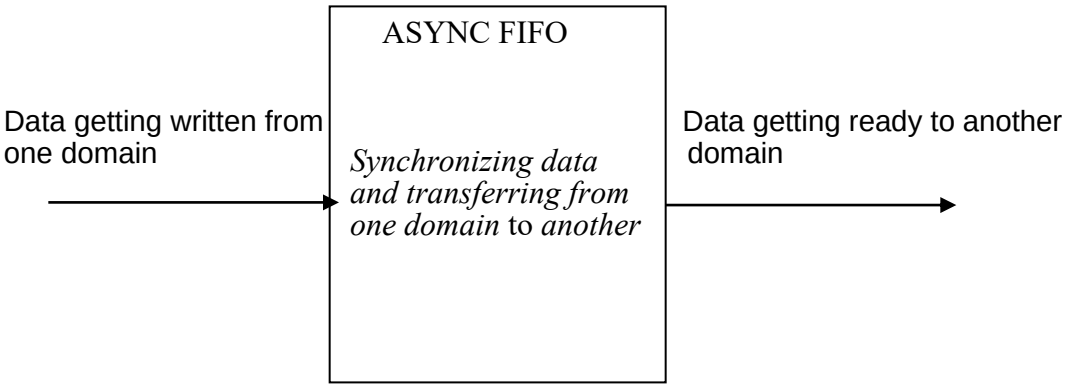
1.5 References

http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf

2. Overall Description

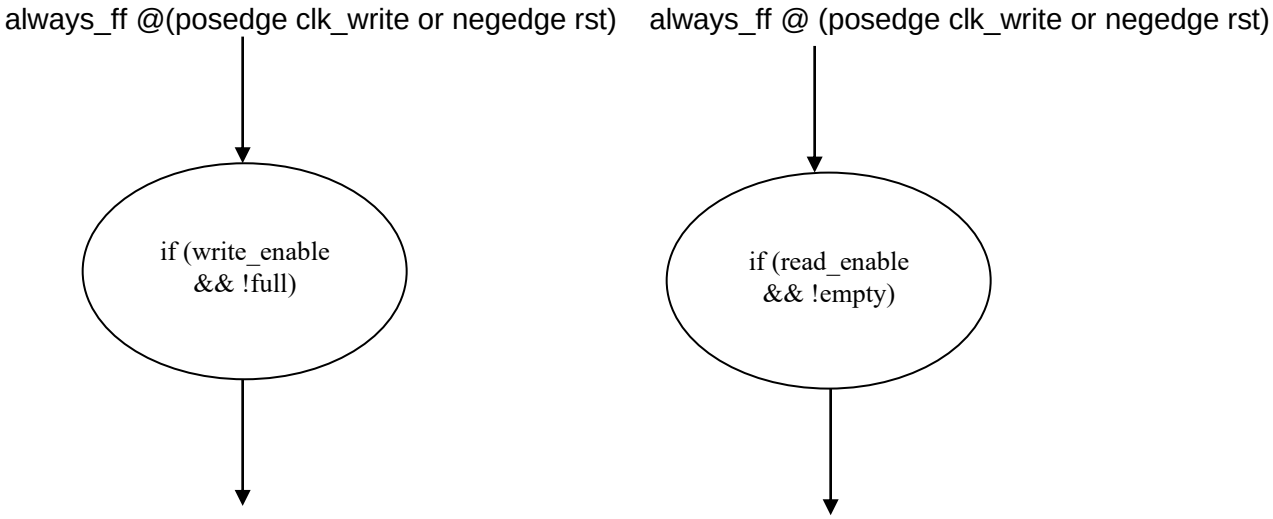
2.1 Product Perspective

This FIFO module is a standalone module, which aims to achieve proper synchronization between different clock domains, with scalable and effective data flow with an ability to integrate into larger SoC subsystems and Microprocessors.



2.2 Product Functions

The main functions are as follows:
Data Write: Writing data to the address inside the FIFO buffer.
Data Read: Reading the data from the memory buffer of the FIFO in the First In First Out fashion.
Full and Empty Condition Status: Keeps updating about whether the Write pointer catching the Read pointer and vice versa.
Almost Full and Almost Empty: Keeps updating about whether the write pointer is about to catch the read point and vice versa



2.3 User Classes and Characteristics

It targets users that work as a hardware engineer such as Logic design engineers and design verification engineers with the help of computer architects can implement and scale this module with complex design. These engineers need to have proper documentation of the design that helps in making design strategies and verification test cases.

2.4 Tools and Software

The Async FIFO module would work in mainstream operating systems such a Linux, Mac, and Windows, and the software applications includes Mentors Questa Sim and Intel Quartus Prime Cadence Xcelium. Any moderate modern hardware setting should be enough for simulation and test cases.

2.5 Design and Implementation Constraints

Accuracy is one of the important constraints for this FIFO module which can become better but with the expense of resources and some serious drawbacks. Also, clock timing amongst different clocks plays a very important role, which could be further explored by doing STA.

2.6 Assumptions and Dependencies

For now, the design will be compatible with any tool that supports simulation of HDLs ad the code will be based on System Verilog/ Verilog. During verification, the whole module will be treated as a black box, and stimulus will be provided to its input while the results of the output will be compared with the intended output.

3. External Interface Requirements

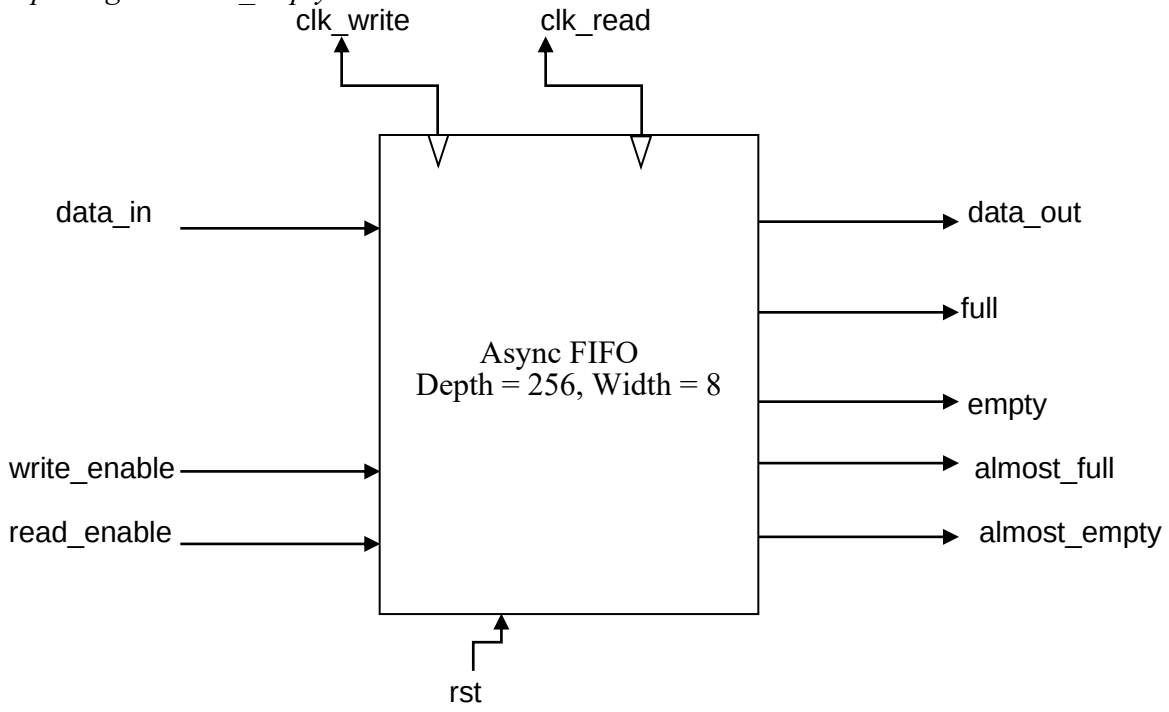
3.1 Hardware Interfaces

Logical elements in the design include memory blocks, synchronizers, and gray code counters. Following are the modules:

A top level design that will interface different modules.

```
Interface fifo_if(input clk_write, clk_read);
logic rst;
logic write_enable, read_enable;
logic full, empty;
logic [7:0]data_in;
logic [7:0]data_out;
logic almost_empty;

module AsyncFIFO (
    input logic clk_write,
    input logic clk_read,
    input logic rst,
    input logic [7:0] data_in,
    input logic write_enable,
    input logic read_enable,
    output logic [7:0] data_out,
    output logic full,
    output logic empty,
    output logic almost_full,
    output logic almost_empty
);
```



Other than this, there will be a memory buffer that will function as a memory between different domains, two synchronizers, that will synchronize read domain to write domain and vice versa, read pointer with empty generation logic to ensure that the necessary action during the empty condition and write pointer with full generation logic to do the same with its condition.

3.2 Software Interfaces

This production can be used in various applications which could use software like MATLAB interfaced with Intel Quartus Prime.

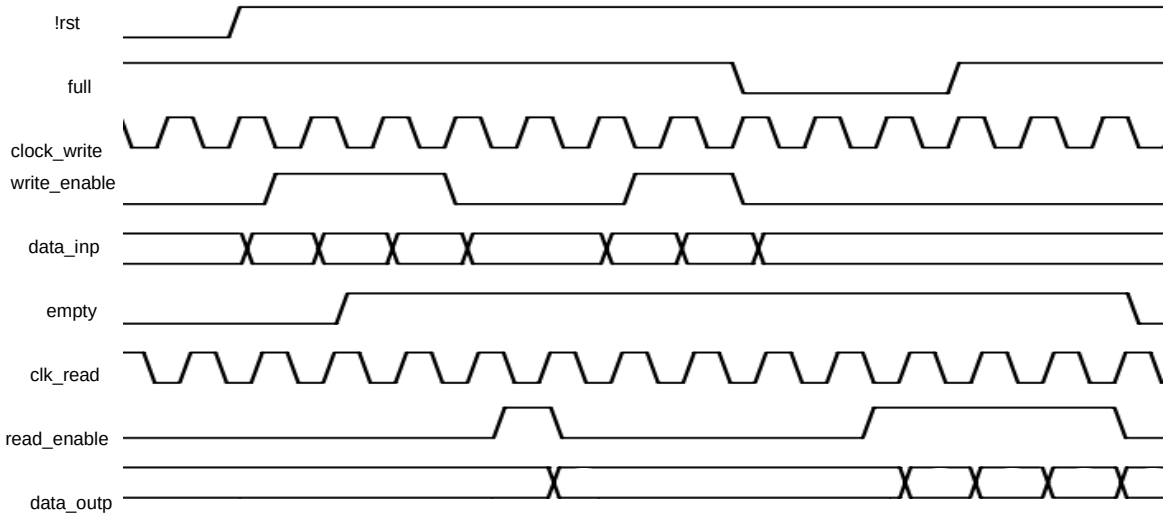
4. Product Features

There are various features in these products.

4.1 FIFO Memory

The product consists of a FIFO memory buffer which could be parameterized as per the design needs. This portion is a dual port synchronous RAM which will be accessed by both differ clock domains. The data will be written from one of the domains and will be read to another domain in a FIFO fashion. Its depth and width can be changed as it is parameterized. The design's flexibility allows it to handle different needs for data width and memory capacities. The memory can perform read and write operations based on the read and write pointers that are provided. When a write operation is carried out, data is written to the memory at the given address. In contrast, a read operation retrieves data from memory by using the read address.

Timing Diagram



4.2 Binary Counter

We have decided to implement binary counter for the initial stage of our design which simply increments or decrements as per the iterations of reads and write pointers.

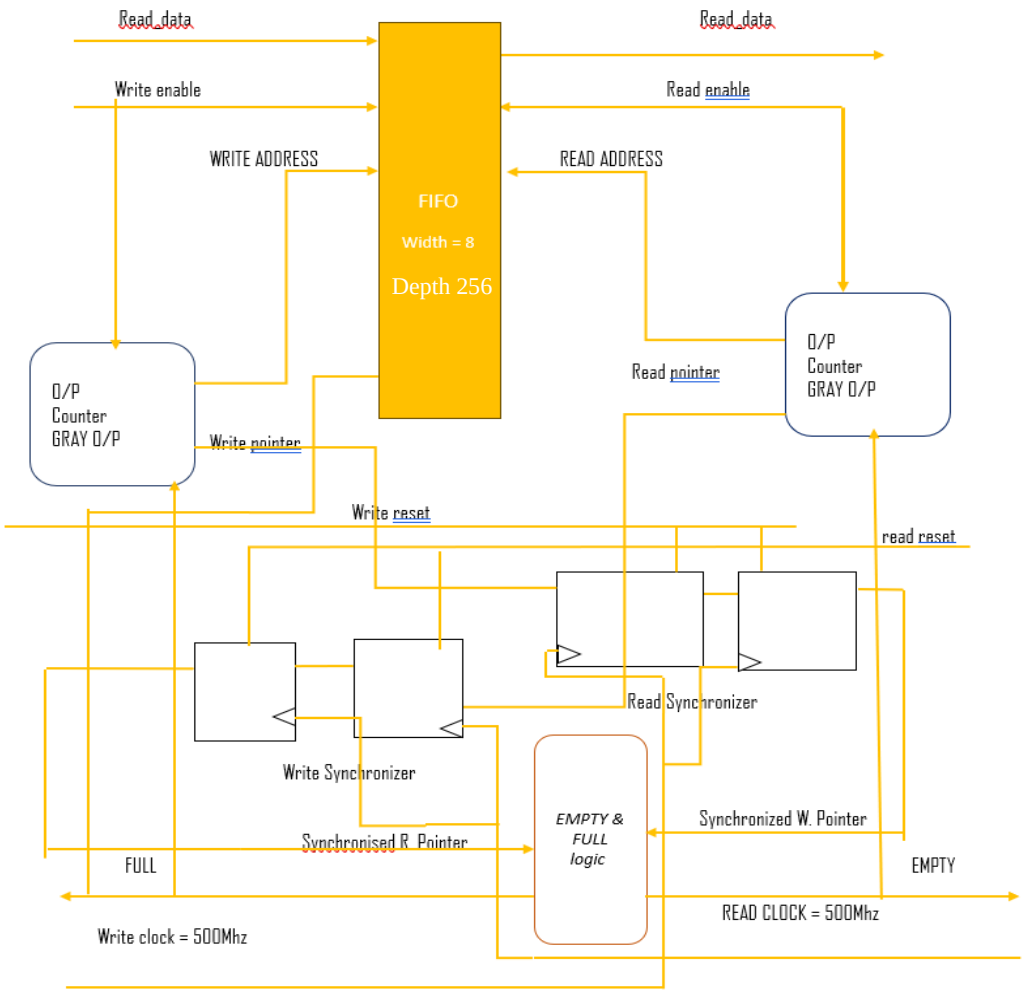
4.3 Synchronizers

This product may include read pointer to write-domain and write pointer to read-domain synchronizers to ensure proper data transfer between clock domains. More specifically, two flip-flop synchronizers are arranged to reduce hazards and provide reliable data transfer. For the first one, the synchronized read pointer is used and compared with to determine the FULL condition. Similarly, for the second one, the synchronized write pointer is used and compared with to determine the EMPTY condition. Their Flipflops are synchronized to write clock and read clock respectively. Comparison of the pointer bits helps in determining the empty and full condition, i.e., if every bit of both the pointers are equal, it is an empty condition or only the MSBs are not equal, then it's the full condition. To determine the empty flag, the synchronized read pointer and the synchronized write pointer are compared.

4.4 Empty and Full flag condition

There will be two logics to assert the Empty and Full status flags. The empty logic functions with the read clock domain and contains the FIFO reads pointer. Similarly, the full logic functions with the write clock domain and contains the FIFO write pointer. These two logics are the most crucial portions as the transmission of correct data from one clock domain to another clock domain relies on it. Suppose if the write pointer catches up the read pointer and the full flag is not set, the very first location will be corrupted then and the very same location with corrupted data will be accessed by the read clock domain. The complete full flag is obtained by comparing the modified write pointer with the synchronized and modified read pointer. Their modules utilize a straightforward block that concatenates the two registers together for shifting and reset. A concise and uncomplicated design is supported by this purposeful simplicity. So after the writing, the FIFO de-assert the empty signals, and also after the read, the FIFO de-assert the write signal.

In both the conditions, if resets are not asserted, then the value of flags depends upon the synchronized pointer value.



4.5 Almost Empty and Almost Full condition

Almost full refers to a situation where the total number of storage spaces occupied is approaching, but not quite reaching, the maximum capacity. Usually, the maximum FIFO depth falls below a preset threshold due to the difference between the write and read pointers. The system's goal is to notify users when the FIFO is almost full so they can take the appropriate action before it really fills. It could act as a warning to prevent data overload or loss. An almost empty FIFO is one that has almost 0 stored entries but is not quite empty. Usually, this condition is triggered when there is a significant increase over a predefined threshold in the difference between the write and read locations. To provide an early warning when the FIFO is about to run out of data, an almost empty signal is used. It can be used to initiate operations so that the flow never ceases, such as obtaining fresh data.

4.6 Parameterization

The flexible parameterization of the design allows for customization of the data size, address size, and other relevant aspects. This characteristic increases the design's flexibility and adaptability, making it appropriate for a variety of uses. The Data Size parameter specifies the Data Bus Width (in bits). Users can choose the size of the data that will be stored in and retrieved from the FIFO, supporting various data widths, based on the demands of the application. The address size parameter indicates how many bits are necessary to reach memory locations inside the FIFO. Customers can use this option to scale for varying storage capacities by matching the required size of the FIFO memory.

4.7 Data Writing

The Data is written on the positive edge, or posedge, of the write clock. This ensures that data is entered into the FIFO consistently and synchronously. A write enable signal, which determines when data is accepted, controls the writing process.

4.8 Interaction Between Clocks

Since the architecture supports two asynchronous clocks, suitable synchronization methods are needed. Synchronizers that safely transfer data and control signals between the read and write clock domains include dual flip-flop designs.

4.9 Clear or Reset Operation

A reset mechanism is incorporated to initialize the FIFO. By activating the FIFO almost empty and/or empty conditions, it guarantees a precisely defined beginning state.

```
if (rst)
      begin
         front_index <= 0;
         rear_index <= 0;
         count <= 0;
         data_out <= 0;
       end
```

4.10 Protocol for FIFO Status

FIFO Empty: Indicates an empty FIFO and is asserted when the read and write pointers are equal.
FIFO Full: A full FIFO is indicated when the difference between the read and write pointers is equal to the FIFO depth.
Almost-Empty/Almost-Full: These states enable sophisticated status reporting, and they are established by binary pointer computations.

5. Logic Design

5.1 Our design and repository can be easily accessed by https://github.com/wasey299/Team9_ECE593W24_Project.git

5.2 Design module

The AsyncFIFO module uses two main processes, one triggered on the positive edge of clk_write and the other on clk_read, to manage data entries and exits. Data is written to the FIFO when write_enable is active and the FIFO is not full, and read from it when read_enable is active and it is not emptyThe indices and count (the total number of elements in the FIFO as of right now) are initialized to zero at reset (rst). Data (data_in) is written to the place indicated by rear_index during operation if the FIFO is not full and a write is enabled (write_enable). This location is then incremented and wraps around if it reaches the depth of the FIFO. On the other hand, data is read from the spot indicated by front_index, which is likewise incremented with wrap-around, if the FIFO is not empty and a read is enabled (read_enable).

The interface fifo_if and its associated clocking blocks (wr_drv, wr_mon, rd_drv, rd_mon) define the protocol for signal timing and data exchange between the FIFO and external entities, separating concerns between driving and monitoring operations for both write and read functionalities. This structured approach allows for clear delineation of functionality, supporting robust and flexible FIFO design suitable for asynchronous data transfer between different parts of a digital system, addressing common challenges in synchronization across clock domains.

Interfacing signals:

For this version, we are using interfacing signals shown below and there will beonly one module for the design.

```
Interface fifo_if(input clk_write, clk_read);
logic rst;
logic write_enable, read_enable;
logic full, empty;
logic [7:0]data_in;
logic [7:0]data_out;
logic almost_empty;
```

5.3 For the simulation process we are using Questa-sim to check the functionality of the design.

We are using VSCode which is mapped to our git repository, from that we can push and pull changes made by the team members.

In the transcript you will be able to see 30 randomized bursts, which are coming from the connection between sequence and driver.

6. Verification

6.1 We've planned to verify our design according to the below table:-

Milestone	Task Description	Deadline
Milestone 1	In this milestone we've tested a basic functionality of a FIFO.	03/02/2024
Milestone 2	In this milestone we've developed a testbench environment in UVM and the whole infrastructure, to ensure the proper communication between the sequence and driver, to generate bursts from the sequence and drive them.	11/02/2024
Coverage Analysis	Assess code and functional coverage reports	DD/MM/YYYY
Debugging stage	Debus TB and collaborating with RTL Designer.	DD/MM/YYYY
Verification Report	Final report with results	DD/MM/YYYY
Sign-off	Final check with DV Manager and sign-off	DD/MM/YYYY

6.2 Our testbench is in UVM environment with all the components in it as:-

- Scoreboard
- Agent
- Sequencer
- Driver
- Sequence
- Sequence_item
- Package (This has all `included files)
- Module top
- test

6.3 Testing Strategies

We've used Dynamic Simulation strategy. We've used UVM methodology, and mechanism we've implemented to ensure that the sequence items (representing bursts of transactions) are accurately transmitted to the DUT and correctly logged for which we've made a connection between Sequence and Driver for burst Transaction. Our checking methodology is designed to ensure the FIFO's response to sequence bursts is as expected, focusing on the integrity and order of data, as well as the FIFO's control signals (e.g. full_empty, almost _full, almost_empty signals).

Scoreboard: - we've implemented a scoreboard in our testbench. Which basically compares the expected outcomes (based on the sequence items sent to the FIFO) against the actual outputs observed by the monitor.

Data Integrity and order: - The scoreboard checks for data integrity and orders, ensuring that each item in a bursts sequence is correctly stored and retrieved from the FIFO.

6.4 Testcase Scenarios

Basic tests

Test Name / Number	Test Description/ Features
1.1.1Read_and_write_test	This test will check basic read and write operation.
1.1.2Reset_test	This test will checks for reset.
1.1.3Basic_seq_transaction_test	This test will validate basic sequence to driver communication and correct FIFO operation for simple write and read transactions. Ensure data integrity and correct signal processing.

Complex tests

Test Name / Number	Test Description/ Features
1.2.1Concurrent_R/W_transaction_test	Concurrent events (R+W) Conditions: fifo_full/fifo_empty/always_full/always empty etc.
1.2.2 randomized_transaction_test	This test will Assess the resilience of the FIFO and driver during stress situations by conducting randomized sequence transactions, which include testing of boundary conditions and implementing random resets.

Regression Tests

Test Name / Number	Test Description/Features
1.3.1 Clock_domain_crossing_test	This test will check the clock domain by focusing on data integrity across clock borders, ensure proper data transmission between various clock domains through driver-sequence interactions.
1.3.2	

Special or Corner cases testcase

Test Name / Number	Test Description
1.4.1data_integrity_check_test	To guarantee strong FIFO functioning, carry out extensive data integrity checks for scenarios requiring complicated sequence transactions, encompassing a range of data patterns and operating circumstances.
1.4.2	

Summary

To summarize, First-In-First-Out (FIFO) memory buffers are crucial components of digital systems because they facilitate the orderly transmission of data across asynchronous clock domains. When data needs to be delayed over several clock regions, FIFOs are often used to ensure synchronized transmission and prevent data loss. The design of this document may be updated in the future.

Appendix A: Depth Calculation

Writing frequency = f_A = 500MHz.

Reading Frequency = f_B = 500MHz.

Duty cycle = 50%

Burst Length = No. of data items to be transferred = 300.

No. of idle cycles between two successive writes is = 2.

No. of idle cycles between two successive reads is = 5.

The no. of idle cycles between two successive writes is 2 clock cycle. It means that, after writing one data, module A is waiting for one clock cycle, to initiate the next write and duty cycle is 50%. So, it can be understood that for every six clock cycles, one data is written.

The no. of idle cycles between two successive reads is 5 clock cycles. It means that, after reading one data, module B is waiting for 5 clock cycles, to initiate the next read and duty cycle is 50%. So, it can be understood that for every 12 clock cycles, one data is read.

Time required to write one data item = (6 * 1)/500Mhz = 12ns
Time required to write into 300 data items = 300 * 12ns = 3600ns
Time required to read one data item = (12 * 1)/500Mhz = 24ns
So, for every 24ns, module B is going to read one data item in the burst.
So, in a period of 3600ns, 300 number of data items can be written.
The no. of data item can be read in a period of 3600 ns = 3600/24 = 150
So, the remaining number of bytes stored in the FIFO = 300 – 150 = 150
Depth = Burst length – 150 = 300 -150 = 150

So, the FIFO which must be in this scenario must be capable of storing 150 data items.

The calculated depth is 150 bits, and the width is 8 bits for this design.
By increasing by 10-15 percent margin, and taking the next 2^n number, we decided to take a depth of 256.
[7:0] memory [0:255]