# High Level Design Specification (HLDS)

# for

# ASYNCHRONOUS FIFO

**Version 1.0**

**Prepared by Shiva Prasad Balne - 907242984 - sbalne@pdx.edu**

**Praveen Kumar Palli - 926398161 - prapalli@pdx.edu**

**Ram Gopal Kasireddy - 977511305 - ramgopal@pdx.edu**

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

**03/12/2024**

# Table of Contents

# 1. Introduction

## 1.1 Purpose

The High-Level Design (HLD) document for an asynchronous FIFO is a comprehensive blueprint outlining the system's architectural and functional aspects. It provides a high-level overview of the system, outlining its purpose, goals, and functionality. The HLD serves as a foundational document for stakeholders, aiding in implementation, testing, and maintenance of the system. It also outlines the performance optimization strategies employed in the design and any external dependencies or assumptions that impact the system's design.

## 1.2 Document Conventions

**FIFO – First in First Out.**

**wptr – Write pointer**

**rptr – read pointer**

**waddr – write address**

**wfull – write full**

**rempty – read empty**

**w_clk – write clock**

**r_clk – read clock**

**wrst – write reset**

**rrst – read reset**

**g_wptr – gray coded write pointer**

**g_rptr – gray coded read pointer**

**wen – write enable**

## 1.3 Intended Audience and Reading Suggestions

The High-Level Design (HLD) document for the asynchronous FIFO is intended for a diverse audience participating in the development, implementation, testing, and maintenance of the asynchronous FIFO system. This includes quality assurance teams conducting testing and validation, project managers supervising the development process, hardware, and software engineers in charge of designing and coding systems, and end users or clients who want a thorough grasp of the architecture and functionality of the system. For decision-makers, the document is a valuable resource that helps them assess the viability, extent, and complexities of the asynchronous FIFO project.

## 1.4 Product Scope

The design of an asynchronous FIFO involves creating a digital circuit that facilitates orderly data transfer between asynchronous clock domains within a larger digital system. The primary goals include overcoming synchronization challenges, providing a buffering mechanism for seamless data transfer, and ensuring reliable communication between components with varying clock speeds. This design offers benefits such as clock domain synchronization, buffering and flow control, flexibility, and reduced latency. In the broader context of digital system design, asynchronous FIFOs play a crucial role in System-on-Chip (SoC) architectures, CPUs, and network chips, enabling the integration of diverse functionalities without the constraints of a shared clock and contributing to the development of modular, scalable, and efficient digital systems. In System-on-Chip (SoC) designs, asynchronous FIFOs are implemented as key components to facilitate data transfer between different IP blocks or subsystems that operate on independent clock domains. The implementation involves integrating asynchronous FIFOs into the overall SoC architecture to ensure seamless communication and coordination among diverse components.

## 1.5 References

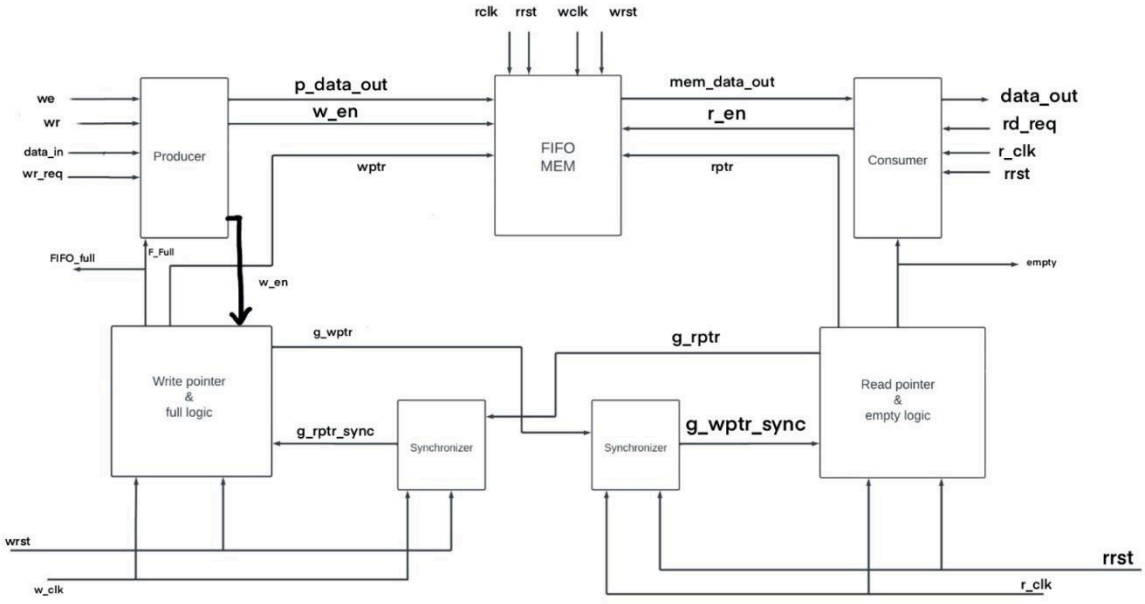

1. **http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf**
2. **http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO2.pdf**
3. **https://hardwaregeeksblog.files.wordpress.com/2016/12/fifodepthcalculationmadeeasy2.pdf**
4. **https://www.youtube.com/watch?v=dCj5HAnaCd8&t=664s**
5. **https://www.youtube.com/watch?v=mGREY8u9ELs&t=1111s**

6. https://vlsiverify.com/uvm/

# 2. Overall Description

## 2.1 Product Perspective

In this asynchronous FIFO implementation, it serves as a vital intermediary between a data producer and a data consumer in a digital system. The producer actively sends data to the FIFO as "data_in" and the FIFO, in turn, actively manages the orderly transfer of this data to the consumer as "data_out." This tailored design specifically addresses the unidirectional flow of data, delivering a crucial buffering mechanism to effectively decouple the producer and consumer, especially when operating on different clock domains. As the producer continually sends data, the FIFO actively ensures controlled and synchronized transfer, actively preventing data loss or overflow. This project actively exemplifies the versatility of asynchronous FIFOs in actively facilitating seamless communication within diverse digital systems where efficient data conveyance between asynchronous components is paramount.




**Block Diagram for the Design**

## 2.2 Product Functions

The major functions of an asynchronous FIFO are:

1. Temporarily stores data for orderly transfer between producer and consumer.
2. Handles data transfer between components with different clock domains to ensure synchronization.
3. Maintains a first-in-first-out order for the sequence of data transfer.
4. Implements mechanisms to control the rate of data transfer, preventing data loss or overflow.
5. Bridges asynchronous clock domains, enabling data transfer between components with independent clocks.
6. Supports read operations by the consumer and write operations by the producer while ensuring synchronization.
7. Metastability can be mitigated by using a synchronizer circuit, which consists of 2 flip flops.
8. Provides status signals indicating whether the FIFO is empty or full, crucial for flow control.
9. Includes warning flags whether the FIFO is almost empty or full.
10. Accommodates variable data rates between the producer and consumer for flexibility.
11. Handles data of varying widths, allowing adaptation to different data formats and structures.

## 2.3 User Classes and Characteristics

The asynchronous FIFO is anticipated to be utilized by a diverse range of users in digital system design. Hardware engineers may leverage its low-level details, data width flexibility, and clock domain crossing capabilities, while software engineers would focus on high-level APIs and abstraction for read and write operations. System architects would consider its integration into broader system architecture and flow control mechanisms. Project managers prioritize its robustness and reliability, and verification teams emphasize simulation and testing capabilities. End users seek efficient data transfer with minimal latency, while custom IC designers look for power consumption considerations and adaptability. Network engineers emphasize integration with network chips and support for diverse protocols, and embedded system developers prioritize integration with peripherals and data handling flexibility. The asynchronous FIFO, characterized by its versatility and diverse features, is poised to meet the varied requirements of its users, facilitating seamless integration into a wide array of design scenarios across hardware, software, and system levels.

## 2.4 Tools and Software

The tools and software we will be using are Questa Sim, MobaXterm to verify the design.

## 2.5 Design and Implementation Constraints

In our case the FIFO is operating under the following conditions:

**Producer clock frequency = 750 MHz**

**Consumer Clock frequency = 250 MHz**

**Duty cycle = 50%**

**Maximum write burst size = 500**

**Number of ideal cycles between successive writes = 0**

**Number of ideal cycles between successive reads = 2**

The calculation for FIFO depth is the crucial part of the design as it determines the sizes of other modules. The calculations are given below:

- The no. of idle cycles between two successive writes is 0 clock cycles. It means that, after writing one data, module A is waiting for one clock cycle, to initiate the next write.
- The no. of idle cycles between two successive reads is 2 clock cycles. It means that, after reading one data, module B is waiting for 2 clock cycles, to initiate the next read. So, it can be understood that for every three clock cycles, one data is read.
- Time required to write one data item = 1 * (1/750 MHz) = 1.33 ns
- Time required to write all the data in the burst = 500 * 1.33 = 666.67 ns
- Time required to read one data item = 3 * (1/250 MHz) = 12 ns
- So, for every 12 ns, module B is going to read one data item in the burst.
- So, in a period of 666.67 ns, 500 no. of data items can be written.
- The number of data items can be read in a period of 666.67 ns = (666.67/12) =55.55.

- The remaining number of bytes to be stored in the FIFO = 500-55.55 = 444.45.
- So, the FIFO which must be in this scenario must be capable of storing 445 data items.

**Therefore, the minimum depth of the FIFO should be 445.**

## 2.6 Assumptions and Dependencies

The FIFO is going to utilize all the remaining 67 locations as well because the minimum depth of FIFO is 445 and the number of bits to represent these 445 locations will be 9.

# 3. External Interface Requirements

## 3.1 Hardware Interfaces

The asynchronous FIFO mentioned in this document has modules like FIFO memory, write pointer and full logic module, read pointer and empty logic module, Binary to Gray conversions, Gray to Binary conversions and Synchronizer module.

**Producer:** The producer is responsible for writing data into the fifo. The producer writes into the fifo by asserting the wr_req signal and write enable(w_en) signal.

**Consumer:** The producer is responsible for reading data from the fifo. The consumer reads from the fifo by asserting the rd_req signal and read enable(r_en) signal.

**FIFO Memory:** This is the FIFO memory buffer that is accessed by both the write and read clock domains. This buffer is most likely instantiated, synchronous dual-port RAM. Other memory styles can be adapted to function as the FIFO buffer.

**Write pointer and full logic module:** This module consists of a write pointer. In this module, the condition for writing full logic determines the FIFO being full. We can also implement almost empty logic and almost full logic based on depth of the FIFO.

**Read pointer and empty logic module:** This module consists of a read pointer. In this module, the condition for reading empty logic determines the FIFO being empty. We can also implement almost empty logic and almost full logic based on depth of the FIFO.

**Synchronizer:** The main intention of using synchronizer is to mitigate the metastability effect on the system. To get full advantage of this feature we need to convert the pointers from binary to gray. The benefits of such conversion can be explained in detail in further sections.

**Binary to Gray conversion:** The conversion from binary to gray of both write and read pointers will be done in their respective modules and are then sent to respective synchronizers.

**The data flow can be explained as follows:**

Whenever the producer initializes sending data, the write pointer starts incrementing on the positive edge of the write clock and negative edge of reset. The write increment starts incrementing the value of the write pointer. Here we have an asynchronous comparison circuit in which we have a comparator, dual n-bit gray counter and quadrant detection circuit to determine the direction of the FIFO going empty or full. In this asynchronous comparison block we are going to compare the asynchronous binary read and write pointer whether full or not, parallelly

we are converting these asynchronous read and write pointers to gray pointers which are incremented until the FIFO is full or empty. These incremented values are further sent to quadrant detection circuitry to determine the direction of FIFO. The logical relation between the result of asynchronous binary read and write pointer and the direction bit gives either of two almost empty or almost full. These signals are then sent to a synchronizer to provide read empty and write full signals respectively. Moreover the FIFO empty and full conditions are determined by binary comparison of Write and read pointers as mentioned in the section 4.

## 3.2 Software Interfaces

The design will be implemented by synthesizable RTL code. The code is then provided with rigorous testing and verification to check the functionality of the asynchronous FIFO. The tools and software we will be using are Questa Sim, MobaXterm to verify the design.

# 4. Product Features

**4.1 Producer:** The producer module is like a user end side module that interfaces with asynchronous FIFO, manages data write operations by giving write requests(wr_req). Further asserts the write enable(w_en) signal in order to perform write operations. By this, the producer effectively controls when data can be written to the FIFO.

**4.2 FIFO Memory :** The FIFO memory module in the context of an Asynchronous FIFO is a crucial component that serves as a shared buffer accessed by both the write and read clock domains. Typically implemented as instantiated, synchronous dual-port RAM, this memory buffer facilitates the orderly transfer of data between asynchronous components within a digital system. The dual-port RAM design allows simultaneous read and write operations, ensuring efficient and synchronized data flow. While synchronous dual-port RAM is commonly used, adaptability is acknowledged, and alternative memory styles can be employed to function as the FIFO buffer based on specific design requirements. This FIFO module is being written by or read from based on the write pointer and read pointer values. This FIFO memory module plays a central role in managing the flow of data, providing a temporary storage solution that aligns with the asynchronous nature of the connected clock domains, thus ensuring seamless communication and preventing data loss or overflow.

**4.3 Write pointer and Full logic:** In the context of an Asynchronous FIFO, understanding how the write pointer and full logic operate is fundamental to grasping the intricacies of the design. The write pointer essentially functions as a guide indicating the specific location in the FIFO memory where the next piece of data should be written. This pointer is initialized to zero during a system reset, designating the initial position for data storage. Upon a FIFO-write operation, the data is written to the memory location pointed to by the write pointer, and thereafter, the pointer is incremented to signify the subsequent location for writing. This sequential movement ensures that data is stored in an organized and orderly manner. Concurrently, full logic plays a crucial role in monitoring the status of the FIFO. It acts like a traffic signal, signaling whether the storage is full or not. If the storage is deemed full, the full logic triggers a signal to halt further write operations, preventing a situation where the system might become overloaded. In essence, the combination of the write pointer and full logic serves to manage the flow of data, guaranteeing controlled and synchronized operations within the Asynchronous FIFO system, thus maintaining its stability and efficiency. The full logic can be implemented as follows: as we know that the depth of our FIFO is 150, so we need 8 bits to address FIFO memory. One method to determine full condition is employing another MSB bit for read and write pointers, so that we can distinguish between read pointer and write

pointer.{f_full = wrst?0:(wptr_gray=={~rptr_gray_sync[9:8], rptr_gray_sync[7:0]})} Here both write and read pointers are of 9 bits each.

```
// Logic to generate fifo full condition.
assign wif.f_full = wif.wrst?0:(wif.wptr_gray=={~wif.rptr_gray_sync[9:8], wif.rptr_gray_sync[7:0]});
```

**Code snippet for fifo full condition**

## 4.4 Read pointer and Empty logic:

In the domain of Asynchronous FIFO design, a pivotal aspect involves understanding the function of the read pointer. This pointer consistently indicates the present word within the FIFO memory that is prepared for reading. Following a system reset, both read and write pointers are set back to zero, indicating an empty FIFO where the read pointer initially references invalid data. Once the initial data word is written, the write pointer advances, the empty flag dissipates, and the read pointer, still pointing to the first memory word, promptly conveys the valid data to the output port for processing by the receiver logic. The significance lies in the fact that the read pointer consistently directs to the subsequent FIFO word for reading, simplifying the procedure for the receiver logic. This negates the necessity for the receiver to increment the read pointer before reading, leading to a more efficient data retrieval process. The FIFO is considered empty when the read and write pointers coincide, a scenario that occurs during a reset or when the read pointer catches up to the write pointer after reading the last word from the FIFO. This ensures a lucid comprehension of the crucial role played by the read pointer in facilitating a streamlined and effective data reading process in Asynchronous FIFO systems. In this design, we can determine the FIFO empty condition whenever the complexity of the write pointer gets caught up with the read pointer. In such cases we determine the empty condition whenever (f_empty = (wptr_gray_sync == g_rptr_next);).

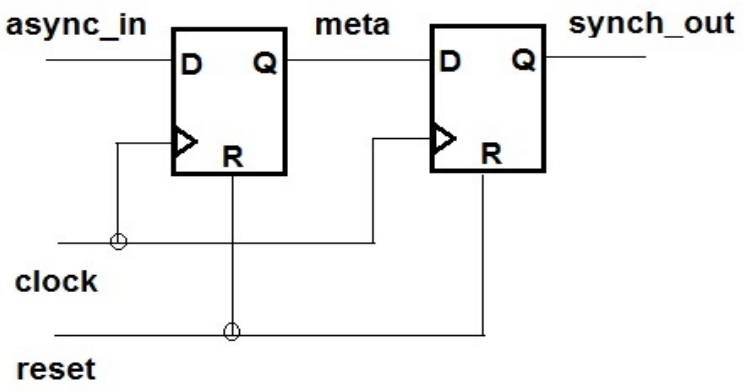
```
assign b_rptr_next = rif.rptr+(rif.r_en & !rif.f_empty);
assign g_rptr_next = (b_rptr_next >>1)^b_rptr_next;
assign rempty = (rif.wptr_gray_sync == g_rptr_next);

always@(posedge rif.r_clk or posedge rif.rrst) begin
  if(rif.rrst) begin
    rif.rptr <= 0;
    rif.rptr_gray <= 0;
  end
  else begin
    rif.rptr <= b_rptr_next;
    rif.rptr_gray <= g_rptr_next;
  end
end

always@(posedge rif.r_clk or posedge rif.rrst) begin
  if(rif.rrst) rif.f_empty <= 1;
  else         rif.f_empty <= rempty;
end
```

**Code snippet for fifo empty condition**

**4.5 Synchronizer:** A synchronizer is essential in an asynchronous FIFO to address challenges associated with clock domain crossing. Asynchronous FIFOs involve data transfer between different clock domains, and the potential misalignment of clocks can lead to metastability issues. Metastability occurs when data changes near the edge of the receiving clock, causing uncertainty and potential errors. In this design we are employing a 2 flip flop synchronizer. A 2-flip-flop synchronizer is a simple but effective design employed in asynchronous FIFOs to mitigate metastability issues during clock domain crossing. In this configuration, two flip-flops are sequentially arranged to capture the incoming signal. The first flip-flop samples the asynchronous signal, and the output is then fed into a second flip-flop, creating a two-stage structure. This arrangement increases the likelihood of capturing the signal in a stable state, reducing the risk of metastability. The combination of these flip-flops acts as a buffer, providing a more robust mechanism for transferring data between asynchronous clock




domains in an asynchronous FIFO. The synchronizer mitigates this risk by introducing a multi-stage structure, typically using flip-flops, to reduce the likelihood of capturing data in an unstable state. This ensures reliable sampling of data between clock domains, minimizing the impact of clock misalignment and preserving the integrity of data transfer within the asynchronous FIFO system.

```
always_ff@(posedge clk or posedge rst) begin
    if(rst) begin
        q1 <= 0;
        d_out <= 0;
    end
    else begin
        q1 <= d_in;
        d_out <= q1;
    end
end
```

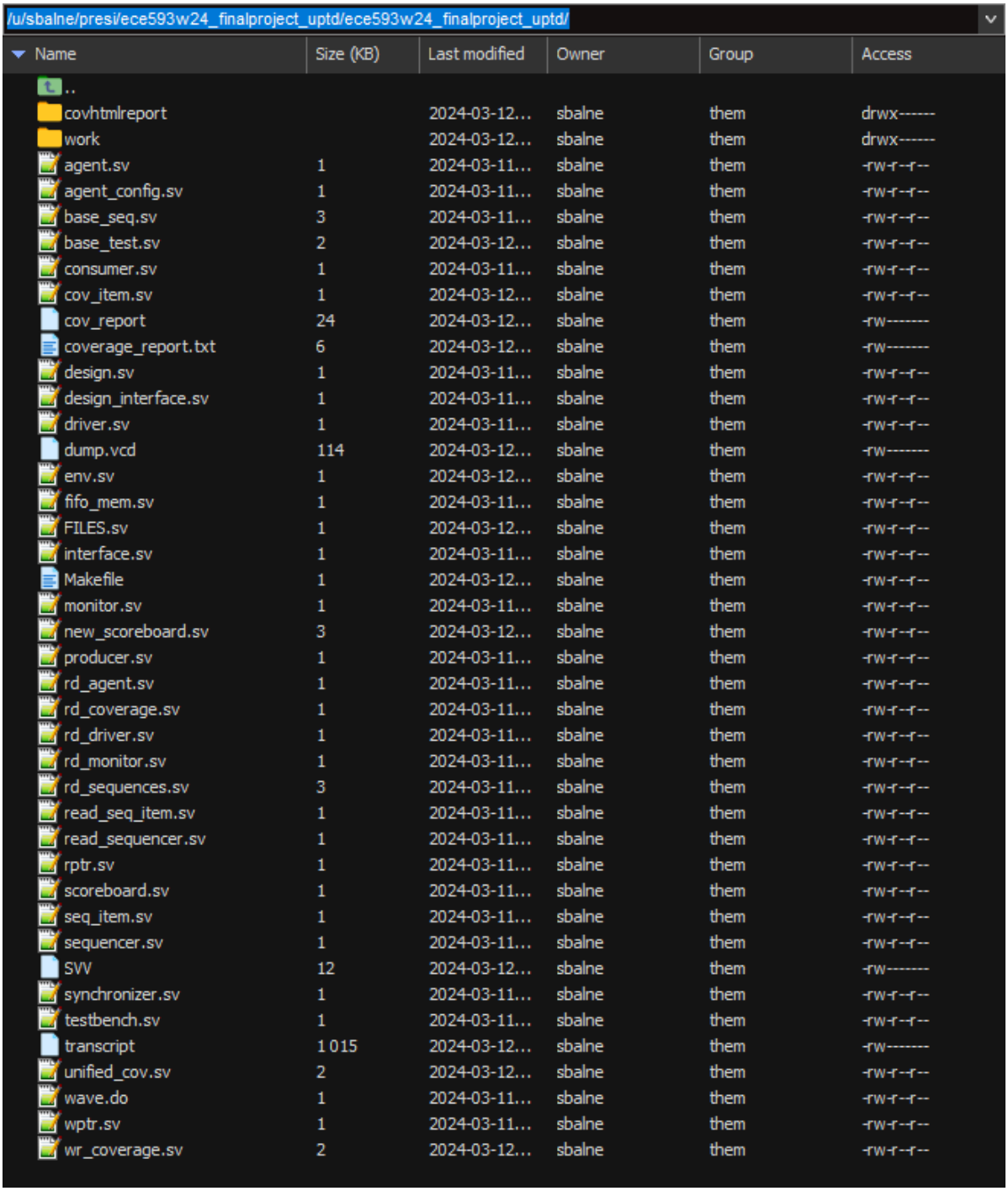
**Code snippet for synchronizer**

**4.6 Consumer:** The consumer module is like a user end receiving side that interfaces with asynchronous FIFO, manages data read operations by giving read request (rd_req) which further asserts a read enable (r_en) signal in order to read from the fifo. By this the consumer effectively controls when data can be read from the FIFO.

# 5. Logic Design:

## 5.1 Directory structure:

- **Github Link:** https://github.com/shivaprasadbalne/PRE_SI/branches



/u/sbalne/presi/ece593w24_finalproject_uptd/ece593w24_finalproject_uptd/

| Name | Size (KB) | Last modified | Owner | Group | Access |
|---|---|---|---|---|---|
| .. | | | | | |
| covhtmlreport | | 2024-03-12... | sbalne | them | drwx------ |
| work | | 2024-03-12... | sbalne | them | drwx------ |
| agent.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| agent_config.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| base_seq.sv | 3 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| base_test.sv | 2 | 2024-03-12... | sbalne | them | -rw-r--r-- |
| consumer.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| cov_item.sv | 1 | 2024-03-12... | sbalne | them | -rw-r--r-- |
| cov_report | 24 | 2024-03-12... | sbalne | them | -rw------- |
| coverage_report.txt | 6 | 2024-03-12... | sbalne | them | -rw------- |
| design.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| design_interface.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| driver.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| dump.vcd | 114 | 2024-03-12... | sbalne | them | -rw------- |
| env.sv | 1 | 2024-03-12... | sbalne | them | -rw-r--r-- |
| fifo_mem.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| FILES.sv | 1 | 2024-03-12... | sbalne | them | -rw-r--r-- |
| interface.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| Makefile | 1 | 2024-03-12... | sbalne | them | -rw-r--r-- |
| monitor.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| new_scoreboard.sv | 3 | 2024-03-12... | sbalne | them | -rw-r--r-- |
| producer.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| rd_agent.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| rd_coverage.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| rd_driver.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| rd_monitor.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| rd_sequences.sv | 3 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| read_seq_item.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| read_sequencer.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| rptr.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| scoreboard.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| seq_item.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| sequencer.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| SVV | 12 | 2024-03-12... | sbalne | them | -rw------- |
| synchronizer.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| testbench.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| transcript | 1 015 | 2024-03-12... | sbalne | them | -rw------- |
| unified_cov.sv | 2 | 2024-03-12... | sbalne | them | -rw-r--r-- |
| wave.do | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| wptr.sv | 1 | 2024-03-11... | sbalne | them | -rw-r--r-- |
| wr_coverage.sv | 2 | 2024-03-12... | sbalne | them | -rw-r--r-- |

Both the design and verification environment files are enclosed in the directory named ece593w24_final_project. In addition a Makefile is created to automate the execution of the verification environment and generate a coverage report. We have created a github repository in which we have included design files, system verilog class based approach for the environment creation and along with it, we have created a UVM environment for the verification of the design.

## 5.2 Design Modules:

The top design module takes inputs as write clock, read clock, write and read resets, read and write requests, input data to be written to FIFO and gives outputs as fifo empty, fifo full and output data. The internal modules consist of a producer, fifo memory, write pointer for determining the fifo full condition, read pointer to determine the fifo empty condition and a consumer to control read operations from the fifo.

Within the verification environment we have modules for write and read sequences, write and read drivers, write and read monitors, write and read agents. A scoreboard is created to validate the data integrity and the performance of the design in capturing the written data. A coverage report is also generated to check the amount of stimulus being used for the verification of the design.

## 5.3 System Verilog abstraction Features used:

In the development of verification environment and design we have used a set of System Verilog abstraction features, which include packages, continuous assignments and procedural blocks, Interfaces, modports, associative arrays, queues, constraints, classes and Objects, parameterized modules that enhance the design's readability, reusability, and verification.

## 5.4 Simulation, Tools, Directory Structure:

The simulation environment and tools used for verifying the design. It would include:

- **Simulation Tools:** ModelSim, QuestaSim, used for running testbenches and simulations.
- **Directory Structure for Simulation:** A makefile is created to automate the execution of design.

- **Verification Methodology:** Employed UVM and class based environment to verify the design.

# 6. Verification:

## 6.1 Objective of the verification plan:

The objective of the verification for asynchronous FIFO is to thoroughly test its functionality and performance across various conditions. The main goal is to make sure that the FIFO operates according to the specification and meets the functional requirements that are outlined in the design and also clock domain crossing verification, metastability handling to ensure proper operation across different clock domains and under metastable conditions. By detecting bugs in the design and through comprehensive coverage analysis we try to present a thoroughly verified asynchronous FIFO design.

## 6.2 Specifications for the design:

- The async FIFO shall support standard FIFO operations including write, read, reset and status monitoring.
- The depth of this async FIFO is 445 data entries with data width of 32 bits long.
- The FIFO shall maintain data ordering and ensure that data transferred should be retrieved in the order that was written.
- The FIFO shall write data upon a write enable signal (1 bit) and shall read data upon a read enable signal (1 bit).
- It shall have two status signals depicting FIFO memory status full/empty (1 bit each).
- The FIFO shall operate with a write frequency of 750 Mhz and read frequency of 250 Mhz with no idle cycles for write and 2 idle cycles for read having a burst length of 500 data items.
- When there is a reset, the FIFO shall clear all the internal storage elements and return to an empty state.
- Gray counters and synchronizers are employed in the design to eliminate metastability.

## 6.3 Level of Verification:

**Unit Level Testing:** This level involves testing of individual modules or components of the design to ensure the correct functionality of the FIFO as specified. It also includes the checking of functionalities like writing and reading of the data items, memory full/empty detection.

## 6.4 Functions Verified:

- **Write**: The write operation in an asynchronous FIFO is synchronized with a write enable signal. Upon assertion of the write enable signal, the FIFO captures the incoming data and stores it sequentially into its internal memory. And a status flag is used to reflect the FIFO memory status whether it's FULL or EMPTY.
- **Read**: The read operation in an asynchronous FIFO is initiated by asserting a read enable signal, which prompts the FIFO to output data through a dedicated output port. And a status flag is used to reflect the FIFO memory status whether it's FULL or EMPTY.
- **FIFO FULL(fifo_full):** The FIFO full flag will be high if the FIFO storage is full unless it'll remain low. Write and read pointers are compared to check this condition to determine if there is more space available for data items to be written to memory.
- **FIFO EMPTY(fifo_empty):** The FIFO empty flag will be high if the FIFO storage is empty unless it'll remain low. Write and read pointers are compared to check this condition to determine if there are more data items available to read.
- **Reset:** Upon a reset FIFO should clear all its internal storage elements i.e; both the write and read pointers have to point to the initial starting storage location.

## 6.5 Testbench Architecture:

**System Verilog class based environment:**

**Generator:** We have created a class named packet with three inputs wr_req, rd_req and data_in and randomized it in the generator.

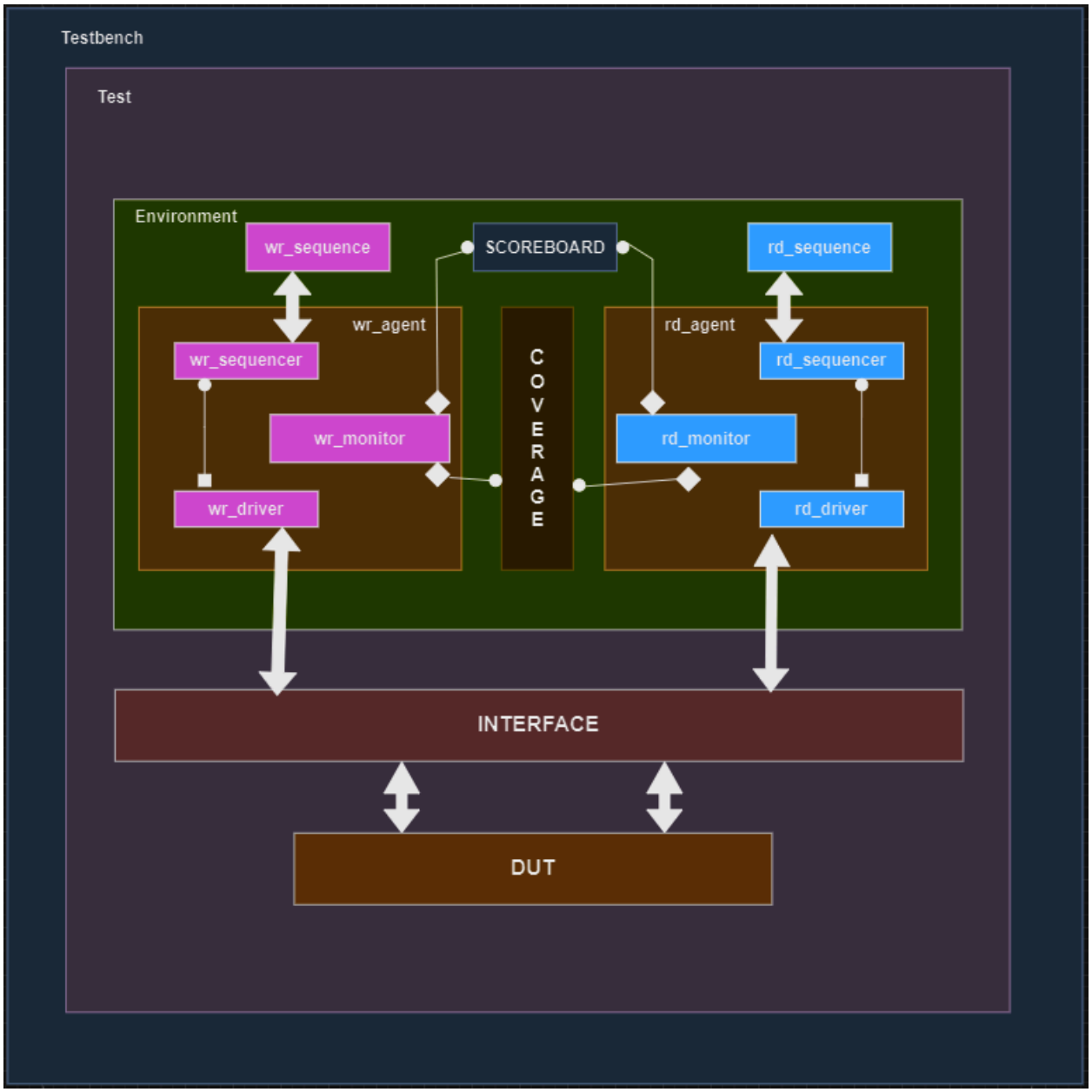
**Driver:** The generator and driver interact by means of a mailbox. We have considered a test with write and read operation to test DUT. Created a virtual interface to be able to interact with the driver and monitor.

**Monitor:** Developed a monitor class and instantiated virtual interface to provide communication between monitor and DUT.

**Scoreboard:** The Scoreboard consists of a reference model. The reference model we have considered to check design functionality by developing a queue with depth equal to fifo and it validates fifo full and empty conditions as well. The queue will be pushed front for every write and the queue will be popped back to implement fifo functionality for every read request from the consumer. The scoreboard checks the condition which compares the design output with queue data out for every read request.

**UVM verification environment:**

**Testbench Hierarchy:**




**Top level Test Bench Components:**

**top_tb:** This top_tb is responsible for coordinating our verification environment.

**Test:** Test is responsible for configuring which kind of stimulus should be driven into DUT by the driver.

**Environment:** Environment is where we'll build, connect and run our agents and components to verify the DUT.

**Agent Architecture:**

In our project, we are configuring two agents. One is with the producer side responsible for writing and the other is with the consumer side responsible for reading.

**Sequencer:** It is basically where the sequence items are sent sequentially to the driver at transaction level.

**Driver:** In general, it is responsible for translating transaction-level objects into pin-level signals and driving them to DUT.

- It is connected to the sequencer via seq_item_port and receives transaction objects from the sequencer and drives them to DUT.
- It utilizes virtual interface(vif) to interact with DUT's interface signals.
- During build phase it initializes its configuration including virtual interface from uvm_config_db and in run phase it drives the corresponding signals to DUT.
- It provides information messages of the signals using UVM messaging.

**Monitor**: Basically, it is responsible for converting pin/signal level stimulus into transaction level objects for further analysis and verification.

- During the build phase it initializes its configuration including a virtual interface from uvm_config_db.
- In the run phase it captures the corresponding signals behavior and populates a transaction item(mon_item) and converts it into a transaction level object and reports the observed behavior using UVM messaging.
- With an analysis port(item_collect_port) it communicates with other components such as scoreboard.

**Scoreboard:** Created a scoreboard to compare the packets being sent to DUT through a virtual interface by the driver and the outputs from the DUT. The checking is done for every posedge of respective clocks in order to capture and compare appropriate written and read data accordingly.

# 6.6 Implementation Strategy:

**Environment Setup:** Developed a modular environment architecture and configured required interfaces between the testbench and DUT.

**Transaction & Sequence Generation:** Defined transaction objects and implemented sequences and covered various scenarios (including FIFO full, empty, overflow & underflow) and randomized sequence item data to cover all input scenarios.

**Driver and Monitor Implementation:** Developed the driver to convert the generated transaction objects into corresponding pin/signal level stimulus to drive to DUT and also develop the monitor to observe the DUT's output signals.

**Scoreboard and Functional Coverage:** Develop the scoreboard to compare the expected and actual outputs. Define coverage goals and implement coverage to track the functional coverage metrics.

**Test development:** Create test scenarios to verify various scenarios like FIFO full, empty, overflow & underflow conditions and randomizing test inputs to improve test coverage.

**Test Scenarios:** Below listed are some of the test scenarios we are going to verify, only few of the below were verified for now and remaining will be completed by the next milestone.

| Test Name / Number | Test Description/ Features |
|---|---|
| Read | Checked basic read operation |
| Write | Checked basic write operation |
| successive writes | Checked FIFO behavior for successive writes |
| successive reads | Checked FIFO behavior for 2 successive reads with idle cycles |
| Read before Write | Checked FIFO behavior for reading before writing any data into memory. |
| Reset | Checked FIFO behavior upon a reset |
| Read after successive writes | |
| Write after successive reads | |

| | |
|---|---|
| Overflow | Checked overflow behavior of FIFO |
| successive reads | Checked FIFO behavior for 3 successive reads with idle cycles |
| Concurrent Read and Write | Simultaneous read and write |
| FIFO full | Checked for FIFO full condition |
| FIFO empty | Checked for FIFO empty condition |

# Summary:

This HLDS document outlines the design and development approach for the "Asynchronous FIFO" project, aimed at implementing a First-In, First-Out (FIFO) queue with asynchronous read and write operations. The document details the directory structure, including locations for source code, testbenches, and documentation, ensuring a well-organized and accessible project layout. Key design modules, such as the FIFO buffer, control logic, and status flag generators, are thoroughly described, highlighting their roles, interfaces, and interactions within the system. The project leverages advanced SystemVerilog features, including interfaces for clean module communication, parameterized modules for design flexibility, and classes for sophisticated testbench creation. While assertion-based verification is omitted, the specification incorporates generics, packages, and continuous assignments for enhanced design clarity, reusability, and verification. The document also specifies the simulation tools and directory structure for efficient and effective verification. By defining clear interfaces and specifications for each design level, the project ensures controllability and observability throughout the development process, facilitating a robust and reliable asynchronous FIFO implementation.