**High-Level Design Specification (HLDS)**

**for**

**Asynchronous First-In-First-Out (FIFO)**

**Version 1.04**

**Prepared by**

**Abram Fouts & Yunus Syed**

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

**January 22th, 2024**

**Table of Contents**

[**Revision History 3**](#_vneadlqakm9y)

[**Contribution: 3**](#_y66rois005nl)

[1. Introduction 1](#)

[1.1 Purpose 1](#_3znysh7)

[1.2 Document Conventions 1](#)

[1.3 Intended Audience and Reading Suggestions 1](#)

[1.4 Product Scope 1](#)

[1.5 References 1](#)

[2. Overall Description 2](#)

[2.1 Product Perspective 2](#_2s8eyo1)

[2.2 Product Functions 2](#)

[2.3 User Classes and Characteristics 3](#)

[2.4 Tools and Software 3](#)

[2.5 Design and Implementation Constraints 3](#)

[2.6 Assumptions and Dependencies 3](#)

[3. External Interface Requirements 3](#)

[3.1 Hardware Interfaces 3](#_44sinio)

[3.2 Software Interfaces 5](#)

[4. Product Features 5](#)

[4.1 FIFO Memory 5](#_6qx3tndwf70p)

[4.2 Flags and Pointers 6](#_4r6e4a343rbd)

[4.3 Synchronizer 6](#_r8a3pc3zqaf)

[4.4 Full and Empty Conditions 8](#_z1zfrqy89fko)

[4.5 Scalable Memory Widths 8](#_nbn0sr46ho8g)

[4.6 Burst Writes 9](#_btowkua5ddx4)

[**5. Logic Design 9**](#_2bn6wsx)

[5.1 Directory Structure 9](#)

[5.2 RTL (System Verilog) 9](#)

[5.2.1 Async\_fifo.sv 9](#_p9g76r28zacw)

[5.3 Features 13](#)

[5.3.1 Write 13](#_f0ilnsktfahb)

[5.3.2 Read 13](#_6fj80bmfo7my)

[5.3.3 Pointer Control 13](#_kz8h984hklpq)

[5.3.4 Reset Control 14](#_tvydz8jklszf)

[5.4 Asynchronous FIFO Waveform 14](#_4xprz72q0w2l)

[6. Verification 14](#)

[6.1 Testbench Style 14](#_147n2zr)

[6.2 Testing Strategies 15](#)

[6.2.1 Producer and Consumer 15](#_wmywpfva5odt)

[6.2.2 Interface 15](#_io76c6bqm0tx)

[6.2.3 Mailbox Usage 15](#_oj6ow5o5qg7t)

[6.3 Test case scenarios 15](#_3o7alnk)

[6.3.1 Corner Cases 15](#_1dh1vk10rko)

[6.3.2 Burst Write 15](#_q1agv8216gw6)

[6.3.3 Data 16](#_yzd4oc5bvarn)

[6.3.4 Reset 16](#_1pnwmusx79jy)

[6.4 Coverage 16](#_23ckvvd)

[**7. Results 16**](#_cag7is1i7jkx)

[7.1 16](#_2n3zrs4p26xr)

[**8. Schedule 16**](#_42m3pa74i2ti)

[**Appendix A: Glossary 16**](#_wfg8akjzx84w)

# Revision History

| **Name** | **Date** | **Reason For Changes** | **Version** |
| --- | --- | --- | --- |
| Abram Fouts | 1/22/24 | Initial Setup | 1.00 |
| Abram Fouts | 1/28/24 | Updated Design Specs, added logic information | 1.01 |
| Yunus Syed | 02/02/24 | Updating HLDS and adding Design spec and extended info | 1.02 |
| Yunus Syed | 03/02/24 | Updating Design RTL | 1.03 |
| Abram Fouts | 02/03/24 | Added verification plan | 1.04 |
|  |  |  |  |

# Contribution:

# **Introduction**

## **Purpose**

The purpose of this HLDS is to show my understanding of the Asynchronous First-In-First-Out and the necessary steps required to fully verify the Asynchronous FIFO. The requirements have been defined in lecture slides and the assignment description. The scope of this verification will be on a single instance of the Asynchronous FIFO where many tests will be performed.

## **Document Conventions**

**Bold** text will be used to bring emphasis to the port connections on the Async FIFIO.

## **Intended Audience and Reading Suggestions**

This HLDS is primarily intended for my professor to review my proficiencies in writing an HLDS also known as a Critical Design Document (CDD), test outline along with detailed test descriptions.

## **Product Scope**

The product that is being designed, tested, and verified is an Asynchronous FIFO. The primary purpose of an Async FIFO is to transfer data between two different clock domains, which is known as clock domain crossing.

## **References**

ZIPCPU. “Crossing Clock Domains with an Asynchronous FIFO.” *Crossing Clock Domains with an Asynchronous FIFO*, 6 July 2018, [zipcpu.com/blog/2018/07/06/afifo.html](https://zipcpu.com/blog/2018/07/06/afifo.html).

Vlsiverify. “Asynchronous FIFO.” *VLSI Verify*, 26 Dec. 2022, [vlsiverify.com/verilog/verilog-codes/asynchronous-fifo/](https://vlsiverify.com/verilog/verilog-codes/asynchronous-fifo).

<http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf.><http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO2.pdf>

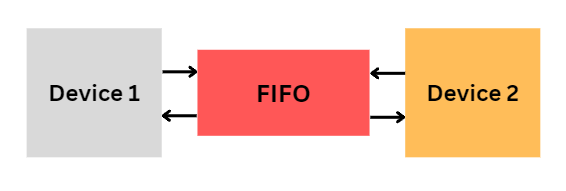
<https://semiconductorclub.com/what-is-fifo-synchronous-fifo-asynchronous-fifo/>

[*https://www.youtube.com/watch?v=mGREY8u9ELs*](https://www.youtube.com/watch?v=mGREY8u9ELs)

# **Overall Description**

## **Product Perspective**

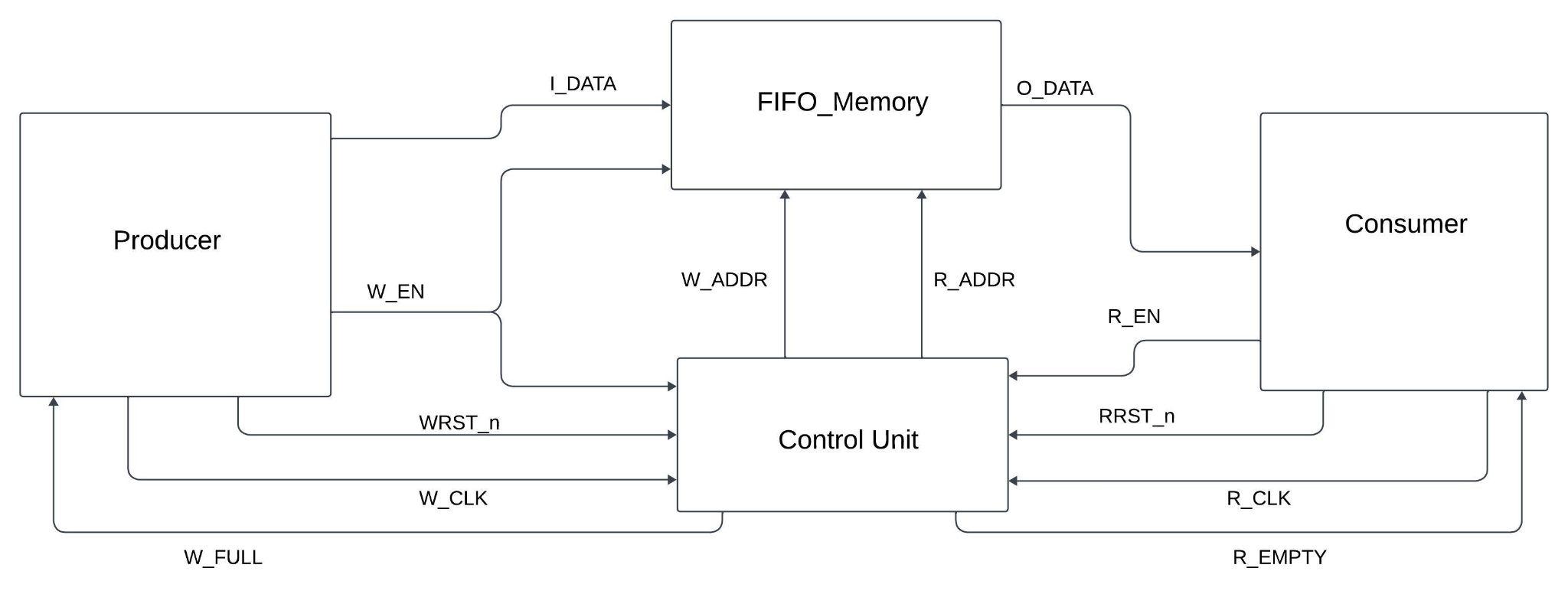
The Asynchronous FIFO is a single component that will be used within systems, chips, SOC’s, and FPGA’s. It’s commonly used in video processing for their clock domain crossings. In the image below the Asynchronous FIFO is in the middle, and there is a producer and consumer connected to the write and read interfaces respectively. The external devices may or may not share a clock of the same frequency, but in the case that they do not, they cannot access the same data safely at the same time, this is where the Asynchronous FIFO comes in.



**Fig 2.1. Basic Block Diagram of FIFO.**

## **Product Functions**

The Asynchronous FIFO is going to contain the common features of a FIFO with the addition to additional clocked interfaces.



**Fig.2.2. Major Signals from FIFO with 2 different Clocks.**

1. Data Write as I\_DATA
2. Data Read as O\_DATA
3. A FIFO Control Unit
4. FIFO Full
5. FIFO Empty

## **User Classes and Characteristics**

Two classes and an interface are intended to be used during this project.

1. Asynchronous FIFO Class
2. External module connection
3. Asynchronous FIFO interface to share data between the test and module.

## **Tools and Software**

The environment in which the design and testing will take place is Mentors Questa Sim 2021.3. The repository where the code, verification and all supporting simulation files will be found here, <https://github.com/abfouts/ece_593_asynchronous_fifo>.

## **Design and Implementation Constraints**

When signals traverse clock domains, there is a risk of metastability, which can cause unpredictable behavior. To mitigate the risk, proper synchronization measures must be employed. For the majority of synchronizing applications, a two-flip synchronizer is sufficient to eliminate metastability. However, in some high-speed applications, a three-flip-flop synchronizer is required. A two-flip-flop synchronizer is all that is required to create an Asynchronous FIFO.

## **Assumptions and Dependencies**

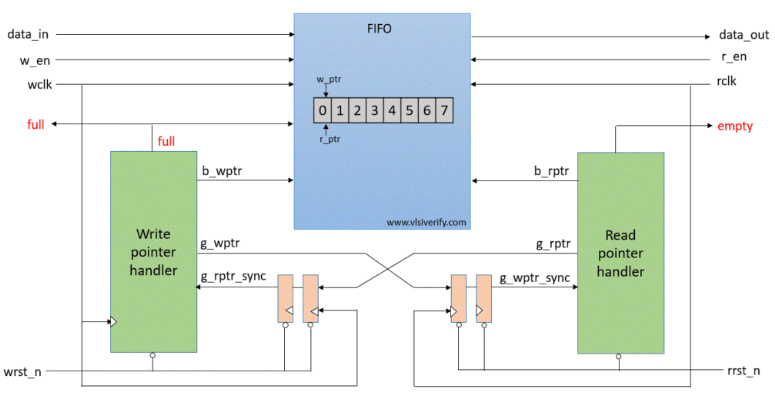
It is assumed that the user will have a basic understanding of SystemVerilog and Questasim to properly run this module. It is assumed that the external devices will work properly and that they have already been verified.

# **External Interface Requirements**

## **Hardware Interfaces**

The Detailed block diagram of FIFO and its internal signals are shown in fig:3.1 , the below modules are involved in the design:

* Flip Flop Synchronizers - Synchronize gray write and read pointers.
* Write Pointer Handler - Generates FIFO using gray coded read pointer. Full and almost-full circumstances
* Read Pointer Handler - Accepts a gray-coded write pointer and generates FIFO empty state.(furthermore, flags going to be added to the design)
* FIFO Memory - Uses binary-coded write and read pointers to store and retrieve data.
* Design module - Creates and connects all the components indicated above.



**Fig:3.1. FIFO Individual modules**

* + 1. **External Device**

The external modules will contain the same interfaces, so they will have data in, data out, read, write, full and empty ports, but they will be connected to the Async FIFO in a way where they will be treated as an individual writer and individual writer. The external writing interface has a clock frequency of 500MHz with a duty cycle of 50%, and the external reading interface will have a clock frequency of 225MHz with a duty cycle of 50%. The writing external interface will have burst write capability lengths of 1024 data writes. The writer has an idle period of 2 cycles, so every 3rd respective clock cycle the device will write. The reader has an idle period of 1 cycle, so every 2nd respective clock cycle the device will read.

| **External Device Port List** | | | |
| --- | --- | --- | --- |
| **Port** | **Direction** | **Type** | **Description** |
| **CLK** | In | Logic | Clock provided to the module |
| **RSTn** | In | Logic | Active low reset |
| **DATA\_IN** | In | Logic [7:0] | Data byte in from the FIFO |
| **DATA\_OUT** | Out | Logic [7:0] | Data byte out to the FIFO |
| **WR** | Out | Logic | Write to enable a FIFO write |
| **RD** | Out | Logic | Read to request a byte from the FIFO |
| **FULL** | In | Logic | FIFO is full do not write |
| **EMPTY** | In | Logic | FIFO is empty do not read |

* + - 1. **External Device Port List**

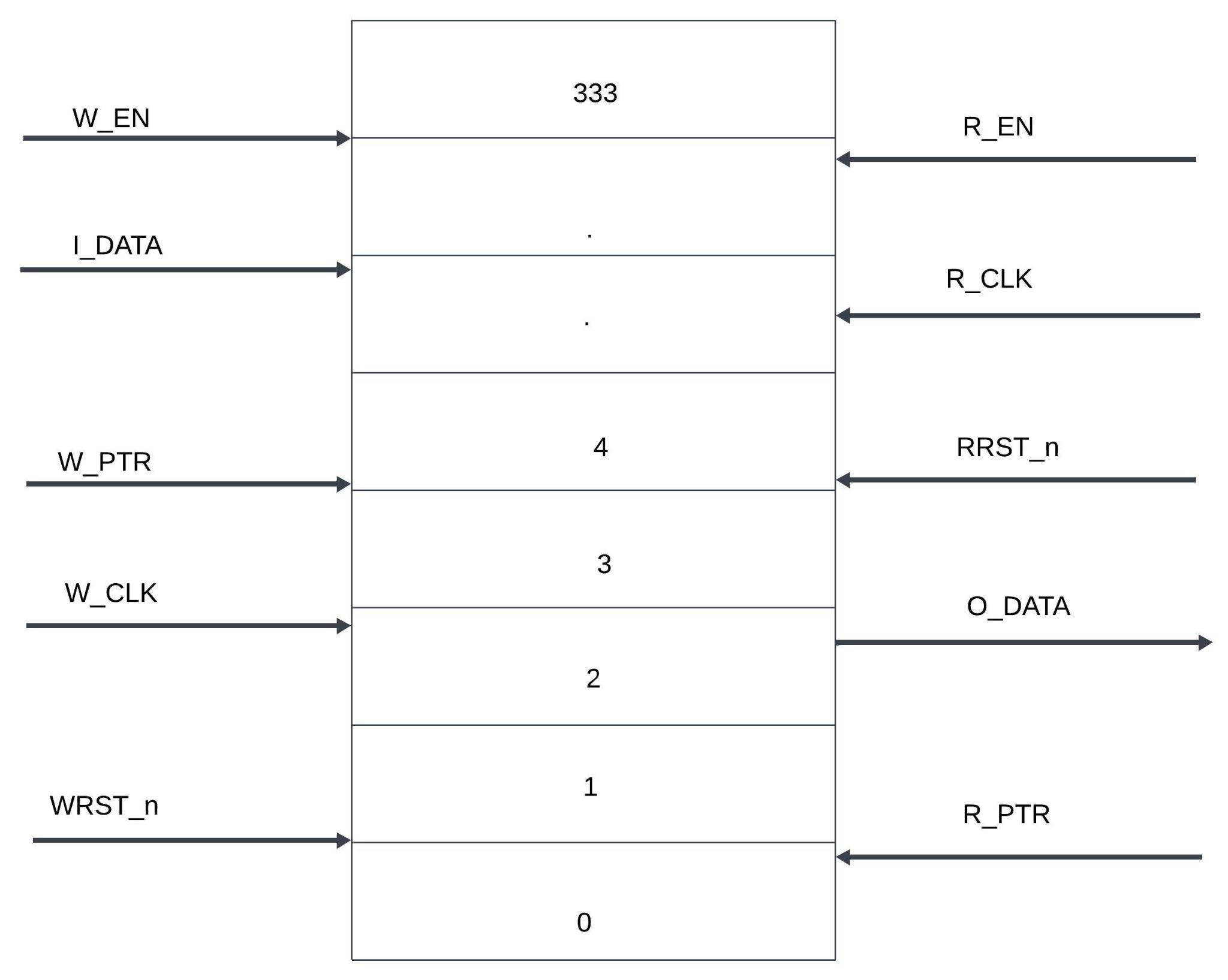
## **Software Interfaces**

This design does not make use of any particular software APIs. Using any EDA tool or HDL simulator, such as Mentor Questa or Synopsys VCS, this design can be synthesized and simulated.

# **Product Features**

## **FIFO Memory**

The Async FIFO will contain all of the same interfaces as the external modules but the directions will be reversed e.g. write is an input and not an output. The FIFO memory will be determined by a generic to allow scalability. The FIFO will only accept single-byte writes for simplicity, but this can be scaled by a set of generics. To prevent any race conditions the device reading data from the FIFO will have priority over the other device unless the FIFO is empty. The memory depth of the FIFO will be determined by the following calculations:

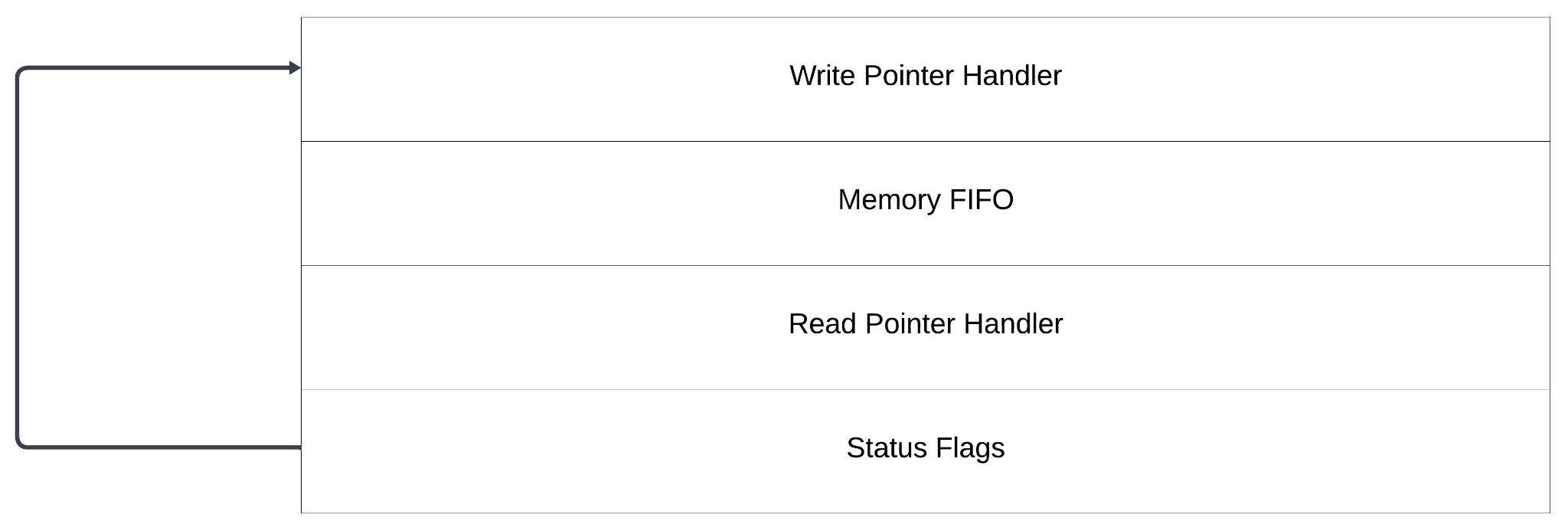
****

**Fig:4.2. FIFO Memory**

Based on the calculations above the FIFO must have at least enough room for 333 data entries to not run into any overflow issues. This is assumed that after a burst write the read is allowed enough time to read the remaining entries before the writer sends more data. Depth can be calculated by log base 2 of Depth which is 9.

## **Flags and Pointers**

Pointer difference, write pointer, and read pointer are all initialized to zero. The initial values of all FIFO status flags are set to default. Resetting binary counters resets the read and write pointers, the pointer difference, and all status flags because reset signals are linked to binary counters.

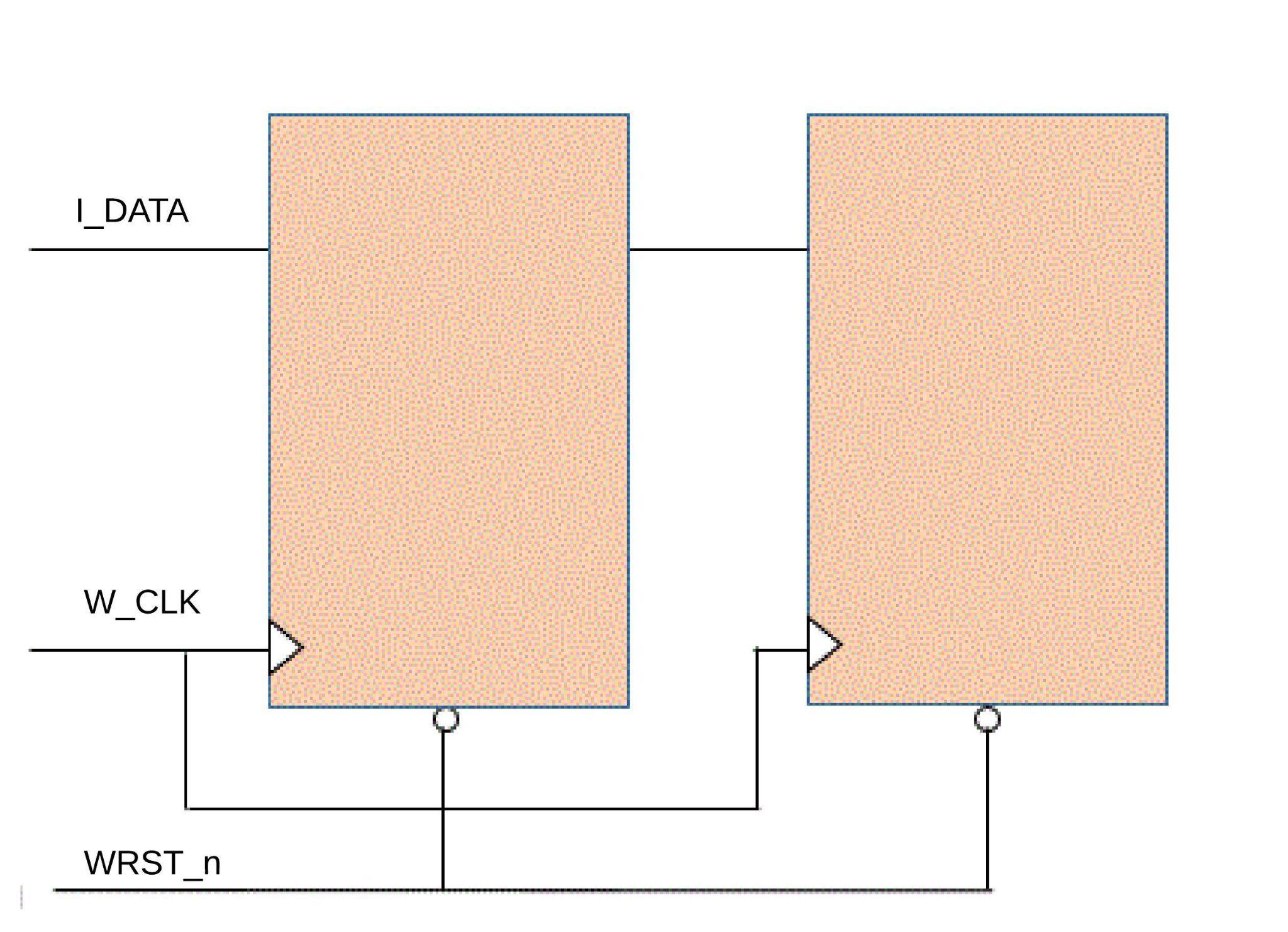


**Fig:4.3. Flags and Pointers**

## **Synchronizer**

It is not necessary for the FIFO design's gray code counters to capture every valid value from the opposing clock domain. However, they must sample the data required to provide the FIFO's Full and Empty states. Therefore, synchronization of the CDC into the receiving clock domain will be required in some way. Figure 4.5 depicts a two-flip-flop synchronizer, which is the most basic and widely used type of synchronizer.

After receiving the out-of-phase signal, the first flip-flop waits one clock cycle to ensure that any instability has subsided. The same clock then feeds the signal from the first flip-flop into a second flip-flop. The goal is to ensure that the second flip-flop's signal is accurate and stable so that it may be applied to the new clock domain.



**Fig:4.5. 2\_Flip-Flop Synchronizer**.

In detail, The FIFO control unit will properly manage any race condition that can happen and will control the output signals of the Asynchronous FIFO. The control unit will be accepting writes from external device 1. Device 1 will be providing writes on a positive edge of a clock when it has data to write. Device 2 will be performing reads on the positive edge of its respective clock. The control unit can read and write at the same time as long as there is enough memory to read or room to write. The control unit will have a parameter that controls the depth of the FIFO memory, and it will control a read pointer and write pointer.

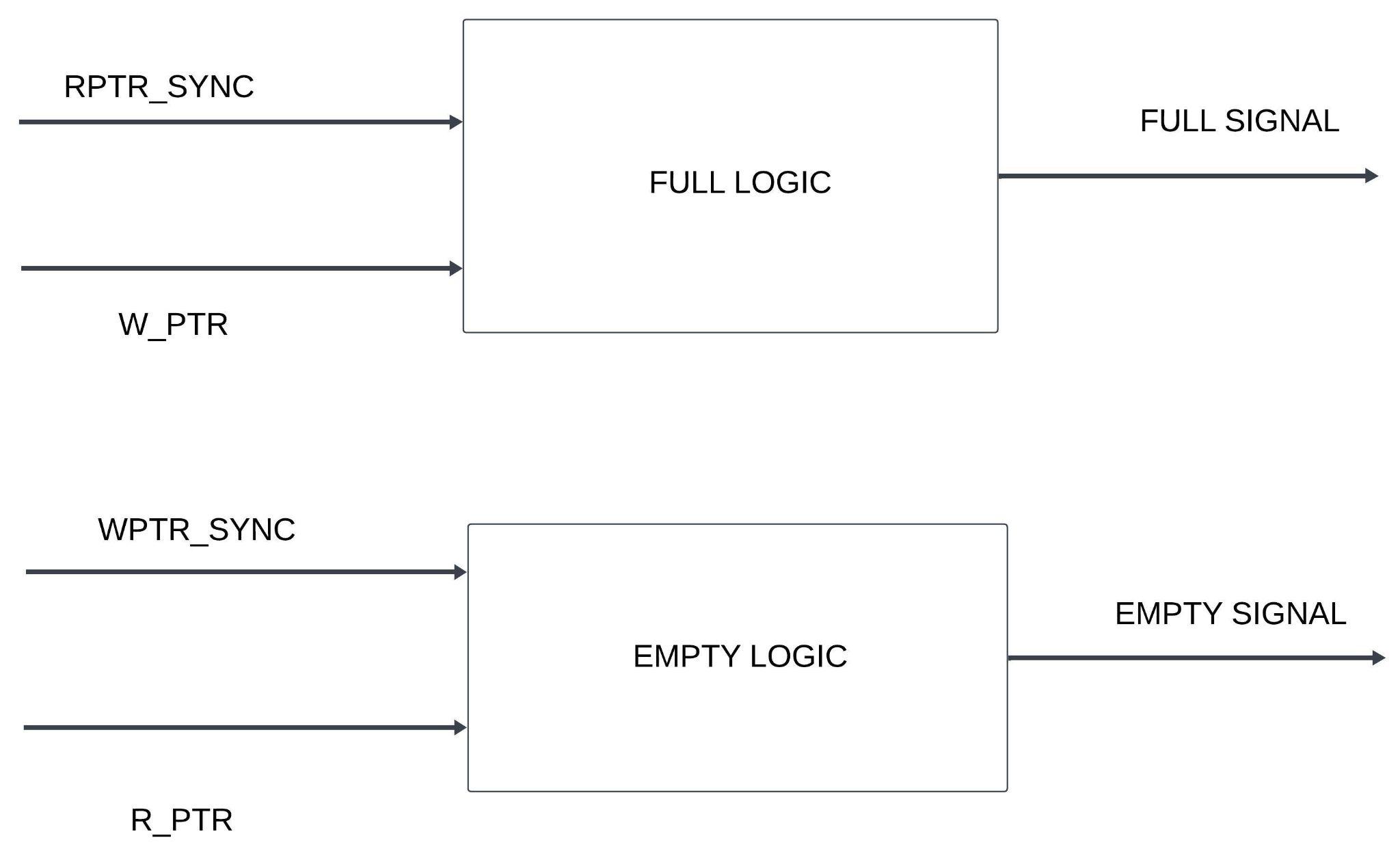
On a write or a read, the pointers will increment and loop back to 0 once reaching the memory depth. To prevent any metastability a flag will be kept to keep track of the write pointer to verify it has or has not looped back to the beginning of memory. This is asserted when the write pointer wraps and the read pointer has not wrapped. Once the read pointer wraps it will deassert. If the pointers are equal and the wrapped signal is asserted the FIFO is full, and if the wrapped signal is deasserted when the pointers are equal the FIFO is empty.

## **Full and Empty Conditions**

We can declare the buffer empty when the write and read pointers are at the same location and the read pointer is trailing the write pointer. In a similar vein, we can declare the FIFO empty if the write pointer follows the read pointer at the same location.

**FIFO Full Condition**: The most significant bit (MSB) of the read and write pointers is XORed to determine the wrap\_around variable. This determines whether the pointers have encircled the buffer. When wrap\_around is true and the remaining bits of the read and write pointers are equal, the FIFO is considered full. This indicates that the write pointer has overtaken the read pointer.

**FIFO Empty Condition**: In the same manner, When the next value of the read pointer equals the binary format of the synchronized write pointer, the FIFO is said to be empty. This signifies that the write pointer and the read pointer have caught up.



**Fig:4.6 Fill and Empty Logics**

## **Scalable Memory Widths**

The memory size for depth and width will be determined by a generic value to allow reuse and scalability. It will be tested and designed around single-byte writes reads and burst writes up to 1024 bursts. The memory will be designed as a queue for simplicity of following the FIFO operations. The memory depth has been calculated to hold 333 data entries. This is manually calculated based on the idle write and read times paired with the writer and reader's respective clock frequencies.

## **Burst Writes**

The external write will have the capability of writing 1024 data bursts.

# **Logic Design**

## **Directory Structure**

The top level directory for this project is structured as follows:

1. **README.md:** Lists the directories and directory descriptions along with how to run the simulation.
2. **docs/:** Any supporting documentation will be stored here for the project e.g. HLDS.
3. **sim/:** Contains simulation files.
   1. work/ directory for simulation files.
   2. run.do file.
   3. wave.do file containing the tests signals for the waveform.
   4. makefile to run the simulation
   5. manifest.f, a RTL and simulation files used in the project.
4. **src/:** Contains the RTL files.
5. **verify/:** Contains the testbench and device model systemverilog code.

## **RTL (System Verilog)**

### Async\_fifo.sv

The following code will need to be updated to reflect the design RTL, but it currently works for simulation as metastability is a non-issue in simulation and this will not be tested physically on an FPGA.

| module async\_fifo #(  P\_DATA\_WIDTH = 8,  P\_MEM\_DEPTH = 333 )(  *// Inputs*  input logic RST\_n,  input logic [DATA\_WIDTH-1:0] DATA\_IN,  input W\_EN,   input R\_EN,   *// Outputs*  output logic [P\_DATA\_WIDTH-1:0] DATA\_OUT,  output logic FULL = 0,  output logic EMPTY = 1 );  integer wr\_ptr = 0; integer rd\_ptr = 0; *//logic [$clog2(MEM\_DEPTH)] wr\_ptr = '0;* *//logic [$clog2(MEM\_DEPTH)] rd\_ptr = '0;* bit wrapped\_mem = 1'b0;  *// Memory* logic [P\_DATA\_WIDTH-1:0] memory [P\_MEM\_DEPTH-1:0];   *//---------------------------*  *// Always block for writes*  *//---------------------------*  *//always\_comb begin*  always @(posedge W\_EN or negedge RST\_n) begin  if (RST\_n == 1'b0) begin  foreach (memory[i]) begin  memory[i] = 8'h00;  end   wr\_ptr = '0;   end  else begin  if (W\_EN && !FULL) begin  memory[wr\_ptr] = DATA\_IN;  wr\_ptr = (wr\_ptr + 1'b1) % P\_MEM\_DEPTH;  end  end  end   *//---------------------------*  *// Always block for reads*  *//---------------------------*  *//` always\_comb begin*  always @(posedge R\_EN or negedge RST\_n) begin   if (RST\_n == 1'b0) begin  DATA\_OUT = 'z;  rd\_ptr = '0;  end  else begin  if (R\_EN && !EMPTY) begin  DATA\_OUT = memory[rd\_ptr];  rd\_ptr = (rd\_ptr + 1'b1) % P\_MEM\_DEPTH;  end   end  end   *//---------------------------*  *// Always block for wrapped control*   *//---------------------------*  always\_latch begin  if (RST\_n == 1'b0) begin  wrapped\_mem = 1'b0;  end   else begin  if (wr\_ptr == 0) begin  wrapped\_mem = 1'b1;  end   if (rd\_ptr == 0) begin  wrapped\_mem = 1'b0;  end  end  end    *//---------------------------*  *// FULL / EMPTY Handles*  *//---------------------------*  always\_comb begin  if (RST\_n == 1'b0) begin  FULL = 1'b0;  EMPTY = 1'b1;  end  else begin  if (rd\_ptr == wr\_ptr) begin  if (wrapped\_mem) begin  FULL = 1'b1;  EMPTY = 1'b0;  end   else begin  FULL = 1'b0;  EMPTY = 1'b1;  end  end  else begin  FULL = 1'b0;  EMPTY = 1'b0;  end   end  end endmodule |
| --- |

| **Asynchronous FIFO** | | | |
| --- | --- | --- | --- |
| **Generic / Parameter** | | **Description** | |
| **P\_MEM\_DEPTH** | | Number of memory entries (depth of the memory) | |
| **P\_DATA\_WIDTH** | | Input and output data width of each memory entry | |
| **Port** | **Direction** | **Type** | **Description** |
| **PROD\_CLK** | In | Wire / Clk | Producer Clock |
| **CON\_CLK** | In | Wire / Clk | Consumer Clock |
| **RST\_n** | In | Logic | Active low reset, clears FIFO |
| **DATA\_IN** | In | Logic [7:0] | Data byte in from the |
| **DATA\_OUT** | Out | Logic [7:0] | Data byte out to the FIFO |
| **W\_EN** | In | Logic | Write byte to memory |
| **R\_EN** | In | Logic | Read to request a byte from the FIFO |
| **FULL** | Out | Logic | FIFO is full do not write |
| **EMPTY** | Out | Logic | FIFO is empty do not read |

* + 1. **Asynchronous FIFO Module Definition**

The asynchronous fifo model will all be contained within the async\_fifo.sv file. The entire design will be built with multiple combinational blocks.

We haven't done any hardcoding in our design, 4 major modules are connected using Design.sv nodules we have used `include to connect all the modules and to interact with each other

## **Features**

The asynchronous FIFO module will parameterize the data widths to and from memory and the depth of the FIFO memory. These will allow the scalability of the module to take on much larger transactions. We have used non-blocking and assigned in our design.

### Write

Writing to the FIFO can be achieved by placing data on the data in port and by strobing the write enable signal. The RTL detects the positive edge of the write enable signal and checks to see if the FIFO is full, and if it is not full it writes data to the write pointer location and increments the write pointer. When the pointer is incremented it is checked by doing a modulus memory depth that makes it wrap once the end of memory is found.

### Read

Reading from the FIFO can be achieved by strobing the read enable signal. The read pointer memories location will be placed on the data output port allowing for the external interface to capture it. This is done on the positive edge of the read enable, and the read pointer is incremented after memory is placed on the data output port. The read point is incremented and wraps to the beginning of memory once it reaches the end.

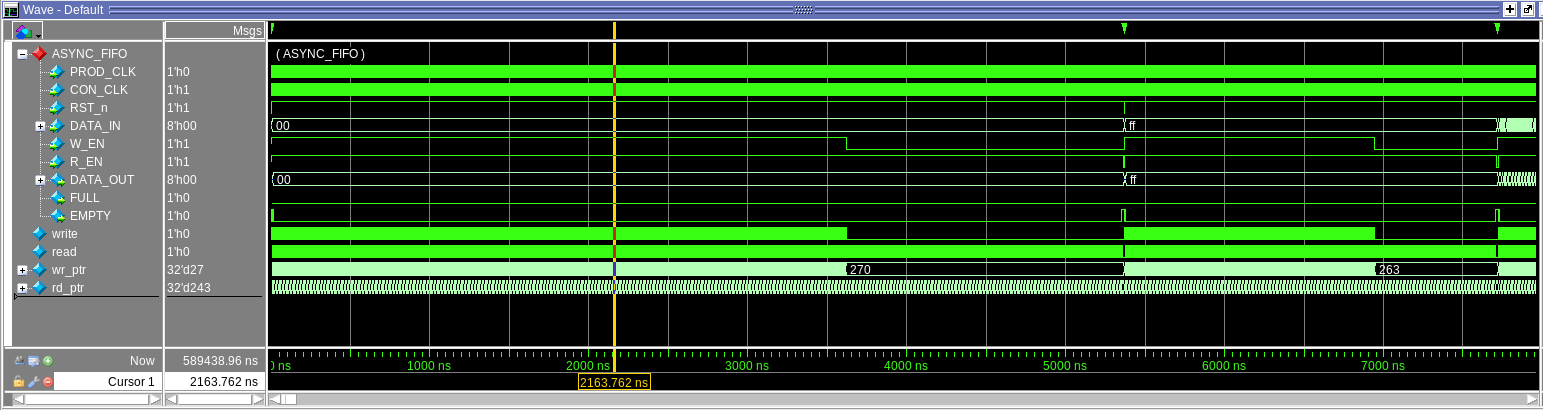
### Pointer Control

The pointer is incremented at the end of a read or write operation, this has been described above. There is a wrapped memory bit flag that lets the FIFO know that the write pointer has wrapped memory. This is used to assert the empty and full signals. The wrapped bit flag is asserted when the write pointer is wrapped to the beginning of memory, and is deasserted when the read pointer wraps to the beginning of memory. The FIFO memory is full when the read and write pointers are at the same location and the wrapped flag bit is asserted. The FIFO memory is empty when the read and write pointer are at the same location and the wrapped flag bit is deasserted.

### Reset Control

There is an asynchronous reset that will reset all logic to a default value of high impedance or to all zeroes.

## **Asynchronous FIFO Waveform**



* + 1. **Burst Write Simulation Waveform**

# Verification

The following sections will describe how the asynchronous FIFO will be verified using strategies learned from ECE-593. The goal of this section is to describe what type of testbench style will be used, what verification strategies will be implemented, and what type of stimulus will be used. A more thorough verification strategy is documented deeply in the *Team 6 Verification Plan* document.

## Testbench Style

The testbench style being used for verification will be a traditional verification style wrapped up with OOP. Tasks will be written to control when the transaction starts, and the number of writes for a burst transaction will be an argument available to the user. This will allow for a simple concise testbench. The idea of this approach will be to use UVM methodology by creating classes to have a driver, pseudo scoreboard, and agent. They will not be official UVM methods, but the common setup structure is used. After a working OOP testbench style is completed, the code will be converted to a UVM testbench style with an agent that controls the stimulus driving and monitoring to and from the DUT interface. Using a sequence item, and multiple sequences this will provide necessary stimulus.

## **Testing Strategies**

### Producer and Consumer

Model the external (producer and consumer) in a class that can be instantiated into the test environment. The model will contain tasks that will perform the necessary verification tasks to prove the asynchronous FIFO is working correctly. This will include all of the requirements and will have a displayed check to show that they are running as expected. A task to start a transaction with arguments that allows the user to choose how many writes they would like to perform, and another argument to provide the tests data to the producer.

### Interface

An interface will be written that connects the external interface models to the asynchronous FIFO DUT. The benefit of treating this as an interface is that inside of the test, and inside of the external model the signals can be snooped and monitored.

### Mailbox Usage

Mailboxes will be used to verify the inputs and outputs of the asynchronous FIFO. Since the external device model contains both producer and consumer, the write mailbox will capture the write data, and the read mailbox will capture the read data. Another thread will take single items out of each mailbox and compare them and make a statement whether or not the data is the same. This will work because using the get() function on the mailbox, it will wait until data occurs before it continues. Since the mailbox acts as a queue each item should be associated together, first write and first read should be the first items in both mailboxes. When changing to UVM mailboxes will not be used, but instead a sequence item will contain dynamic arrays for both input and output data, and the arrays will grow to provide the entirety of the transaction to the scoreboard to compare each written and read values.

## **Test case scenarios**

### Corner Cases

Testing corner cases such as writing no values to the asynchronous FIFO, writing one value, writing max values, and writing beyond max values will be tested and considered as corner cases.

### Burst Write

Burst writes of 1024 will be tested as it is a maximum burst write requirement. This will also be tested through robustness and many different sizes will be tested. Every single size can be tested as it is a very quick simulation, but in the future if the tests become longer, less tests in the middle of 1 to 1024 range will be used less.

### Data

Random data will primarily be used to verify the writing and reading capabilities of this project, but there will be some key important data sets, such as incrementing values, random, all 1’s, and all 0’s.

### Reset

Reset will be tested at many different points. A reset will occur before, during and after any tests. Tests such as writing half a burst, reset and writing the remaining burst data will also prove reset is working properly.

## **Coverage**

There is an expectation to achieve 100% code coverage, except in areas where parameters will not allow the code to occur.

# Results

## Test Results

## Code Coverage Misses

The few code coverage misses occurred due to a flaw in the RTL where there will be redundant branches resulting in the nested else branch never being taken because the top level branch is the same conditional. A full report can be found in the *dut\_coverage\_report.txt*.

Below will show the branch misses, for the remaining expression, conditional, and statement misses please review the *dut\_coverage\_report.txt*.

| *------------------------------------IF Branch------------------------------------*  73 66188 Count coming in to IF  73 1 66188 if (!FULL) begin  \*\*\*0\*\*\* All False Count Branch totals: 1 hit of 2 branches = 50.00%  *------------------------------------IF Branch------------------------------------*  74 66188 Count coming in to IF  74 1 66188 if (W\_EN) begin  \*\*\*0\*\*\* All False Count Branch totals: 1 hit of 2 branches = 50.00%  *------------------------------------IF Branch------------------------------------*  112 66188 Count coming in to IF  112 1 66188 if (write && W\_EN && !FULL) begin  \*\*\*0\*\*\* All False Count Branch totals: 1 hit of 2 branches = 50.00%  *------------------------------------IF Branch------------------------------------*  128 66188 Count coming in to IF  128 1 66188 if (read && R\_EN && !EMPTY) begin  \*\*\*0\*\*\* All False Count Branch totals: 1 hit of 2 branches = 50.00% |
| --- |

## Function Coverage

100% function coverage was achieved by monitoring each interface signal and measuring every value. The data in and data out signals were broken into three categories, all zeros, all ones and others. The names refer to the value of the bits on the interface bus. The remaining interface signals were single bit so there are only two potential values for each one to be, 1 or 0. After testing the device with the agent function coverage is reported to be 100%. The functional coverage report can be found at *fcover\_report.txt*.

| TOTAL COVERGROUP COVERAGE: 100.00% COVERGROUP TYPES: 3  Total Coverage By Instance (filtered view): 100.00% |
| --- |

## Issue Tracker

### Reset in middle of transaction

This was not able to be performed without deeply researching how to intersect an active sequence and override the current sequence in the driver to a new one. The test attempted was to send a random sequence, wait some time then start the reset sequence, but it would only wait until the first transacter had finished then send the reset sequence.

### Strobing WR/RD Enable

The original RTL design is to

# Schedule

| Milestone | Description | Due |
| --- | --- | --- |
| Milestone 1 | Initial design as specified in HLDS. Initial test bench compiling. Initial verification plan. | 02/03/2024 |
| Milestone 2 | Testing using OOP. Refined verification plan. | 02/11/2024 |
| Milestone 3 | OOP Coverage and working RTL | 02/17/2024 |
| Milestone 4 | UVM Conversion | 02/25/24 |
| Milestone 5 | Finalized UVM agent, env, coverage, scoreboard, tests | 03/03/24 |
| Final | Finalize documentation any code changes, ready for turn in | 03/05/24 |

# Appendix A: Glossary

SV - Systemverilog

UVM - Universal Verification Methodology

FIFO - First-In-First-Out

Async - Asynchronous

IF - Interface

Generic - Parameters determined on the modules

OOP - Object Oriented Programming