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

**for**

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

**Version 1.11**

**Prepared by**

**Abram Fouts & Yunus Syed**

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

**January 22th, 2024**

**Table of Contents**

[**I. Revision History 4**](#_v5yla2s1koda)

[**II. Contribution: 5**](#_y66rois005nl)

[i. Abram Fouts Expertise 5](#_rkk09peyhw7l)

[ii. Abram Fouts Contributions 5](#_py22i53dc9q6)

[iii. Yunus Syed 5](#_fkz5bovyb6tb)

[**1. Introduction 6**](#_bgt0pcg9mm4d)

[1.1 Purpose 6](#_3znysh7)

[1.2 Document Conventions 6](#_id80nl44n1fj)

[1.3 Intended Audience and Reading Suggestions 6](#_k0aawen90xoz)

[1.4 Product Scope 6](#_ytvudiw9b6bv)

[1.5 References 6](#_pblmkr2ozavi)

[**2. Overall Description 7**](#_vnseufhxnhqf)

[2.1 Product Perspective 7](#_2s8eyo1)

[2.1.1 Fig. Basic Block Diagram of FIFO 7](#_75hufvtq1owv)

[2.2 Product Functions 7](#_omfecfsmsme9)

[2.2.1 Fig. Major Signals from FIFO with 2 Different Clocks 7](#_y12mh919dz7r)

[2.3 User Classes and Characteristics 8](#_e1yfs9ttq207)

[2.4 Tools and Software 8](#_llfm2b651atz)

[2.5 Design and Implementation Constraints 8](#_52frcg33hz29)

[2.6 Assumptions and Dependencies 8](#_hu2tntakdea6)

[**3. External Interface Requirements 8**](#_ejmyrc4bcjkx)

[3.1 Hardware Interfaces 8](#_44sinio)

[3.1.1 Fig. FIFO Internal Block Diagram 9](#_mtfkcktkzc0p)

[3.1.2 External Device 9](#_jnigibbhnb7h)

[3.1.2.1 External Device Port List 10](#_6pp2oahxu50l)

[3.2 Software Interfaces 10](#_y9pfllokdxhd)

[**4. Product Features 10**](#_imhdqmcen8w5)

[4.1 FIFO Memory Calculations 10](#_6qx3tndwf70p)

[4.2 Memory Pointer Controller 11](#_4r6e4a343rbd)

[4.3 Write/Read Idle Controller 11](#_yy48hjv4z2o3)

[4.4 Write/Read Controller 11](#_xeqivtes1lja)

[4.5 Full and Empty Controller 11](#_z1zfrqy89fko)

[4.5.1 Wrapped Memory 11](#_hc5drd28sluk)

[4.5.2 Full Logic 12](#_dlcr27jhgpac)

[4.5.3 Empty Logic 12](#_7dl0h0mgiy88)

[4.5.4 Reset 12](#_10z82xbtqgkz)

[4.6 Scalable Memory Widths 12](#_nbn0sr46ho8g)

[4.7 Burst Writes 12](#_btowkua5ddx4)

[**5. Logic Design 12**](#_2bn6wsx)

[5.1 Directory Structure 12](#_k8p4renl9iyg)

[5.2 RTL (System Verilog) 13](#_vxxzlrf18k63)

[5.2.1 Async\_fifo.sv 13](#_p9g76r28zacw)

[5.2.2 Code: System Verilog RTL 16](#_ijl3trnkg8j2)

[5.2.3 Table: Asynchronous FIFO Module Definition 17](#_ymup2gxbsv1n)

[5.3 Features 17](#_s3f8mkmsdk6q)

[5.3.1 Write 17](#_f0ilnsktfahb)

[5.3.2 Read 18](#_6fj80bmfo7my)

[5.3.3 Pointer Control 18](#_kz8h984hklpq)

[5.3.4 Reset Control 18](#_tvydz8jklszf)

[5.4 Asynchronous FIFO Waveform 18](#_4xprz72q0w2l)

[5.4.1 Burst Write Simulation Waveform 18](#_nn9b9o2uhm14)

[**6. Verification 18**](#_1timy1i9a4xx)

[6.1 Testbench Style 19](#_147n2zr)

[6.2 Testing Strategies 19](#_ff9nfyo16uy8)

[6.2.1 Producer and Consumer 19](#_wmywpfva5odt)

[6.2.2 Interface 19](#_io76c6bqm0tx)

[6.2.3 Mailbox Usage 19](#_oj6ow5o5qg7t)

[6.3 Test case scenarios 20](#_3o7alnk)

[6.3.1 Corner Cases 20](#_1dh1vk10rko)

[6.3.2 Burst Write 20](#_q1agv8216gw6)

[6.3.3 Data 20](#_yzd4oc5bvarn)

[6.3.4 Reset 20](#_1pnwmusx79jy)

[6.4 Coverage 20](#_23ckvvd)

[**7. Results 20**](#_cag7is1i7jkx)

[7.1 Test Results 20](#_jf9lfbm6fu26)

[7.2 Code Misses 21](#_2n3zrs4p26xr)

[7.2.1 Fig. Code Coverage Results 21](#_ibecw77oh98p)

[7.3 Function Coverage 21](#_wt59hw23zeiu)

[7.3.1 Fig. Functional Coverage Results 22](#_pz0sqvjxpi40)

[7.4 Verified by Inspection 22](#_dbd2dfk5w7bj)

[7.4.1 Fig. 500MHz (2ns period) & 225MHz (4.44ns) Clock Verification 23](#_5ptb9v2rakni)

[7.4.2 Fig. Write Idle Inspection (6ns) & Read Idle Inspection (8.88ns) 23](#_ab9uobb20fqq)

[7.5 Issue Tracker / Findings 23](#_5s04nr4nu8sw)

[7.5.1 Timing 23](#_pktteki447n3)

[7.5.2 Reset in middle of transaction 23](#_jrfqauq5eycc)

[7.5.3 Strobing WR/RD Enable 23](#_gw9mebci0pn)

[7.5.3.1 Fig. Strobing WR/RD Enable Back-to-back burst sequences 24](#_ghs1cni648qf)

[7.5.3.2 Fig. Strobing WR/RD Enable Back to back burst sequences transcript 24](#_8mr9agsn0xcs)

[7.5.4 UVM Architecture 24](#_xqsanunc7t7l)

[7.5.5 Parameterization 25](#_je8jixgw3pkb)

[**8. Schedule 25**](#_42m3pa74i2ti)

[**A. Appendix: Glossary 25**](#_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 |
| Yunus Syed | 01/03/24 | UVM block diagram and verification plan | 1.04 |
| Abram Fouts | 02/03/24 | Added verification plan | 1.05 |
| Abram Fouts | 03/03/24 | Added Results and Issue Tracker | 1.06 |
| Abram Fouts | 03/03/24 | Updated Strategies | 1.07 |
| Abram Fouts | 03/04/24 | Reworking RTL Changes | 1.08 |
| Abram Fouts | 03/04/24 | Updated Results and Verification | 1.09 |
| Abram Fouts | 03/05/24 | Updated Issues / Findings | 1.10 |
| Yunus Syed | 03/05/24 | Updated RTL Spec | 1.11 |

# 

# Contribution:

## Abram Fouts Expertise

Working professional since graduating undergrad with a degree in Computer Engineering from Portland State. Worked in Aerospace since graduation working with FPGA design and FPGA verification. Been in verification for the last two years writing system verilog and UVM, and prior to graduation interned at Intel on the PHED team doing board design for 6 months, and then a 6 month internship at Micron writing software remotely for the board design team.

## Abram Fouts Contributions

1. Created the simulation directory environment.
2. Wrote a version of the DUT code.
3. Created outline of the object oriented programming section.
4. Filled in the OOP code.
5. Converted the OOP code to UVM.
6. Created UVM components, agent, env, scoreboard, and tests.
7. Participated in documentation.

## Yunus Syed

1. Verification of DUT code and added some elements.
2. created some sequences, and agent and testing of OOP code.
3. Helped in filling the OOP code and UVM testbench.
4. Updated versions of DUT and tested them.
5. Participated in doing documentation.

# 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>

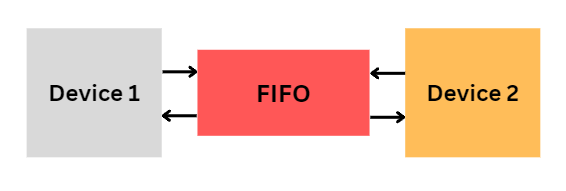
<https://semiconductorclub.com/what-is-fifo-synchronous-fifo-asynchronous-fifo/>

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

# Overall Description

## Product Perspective

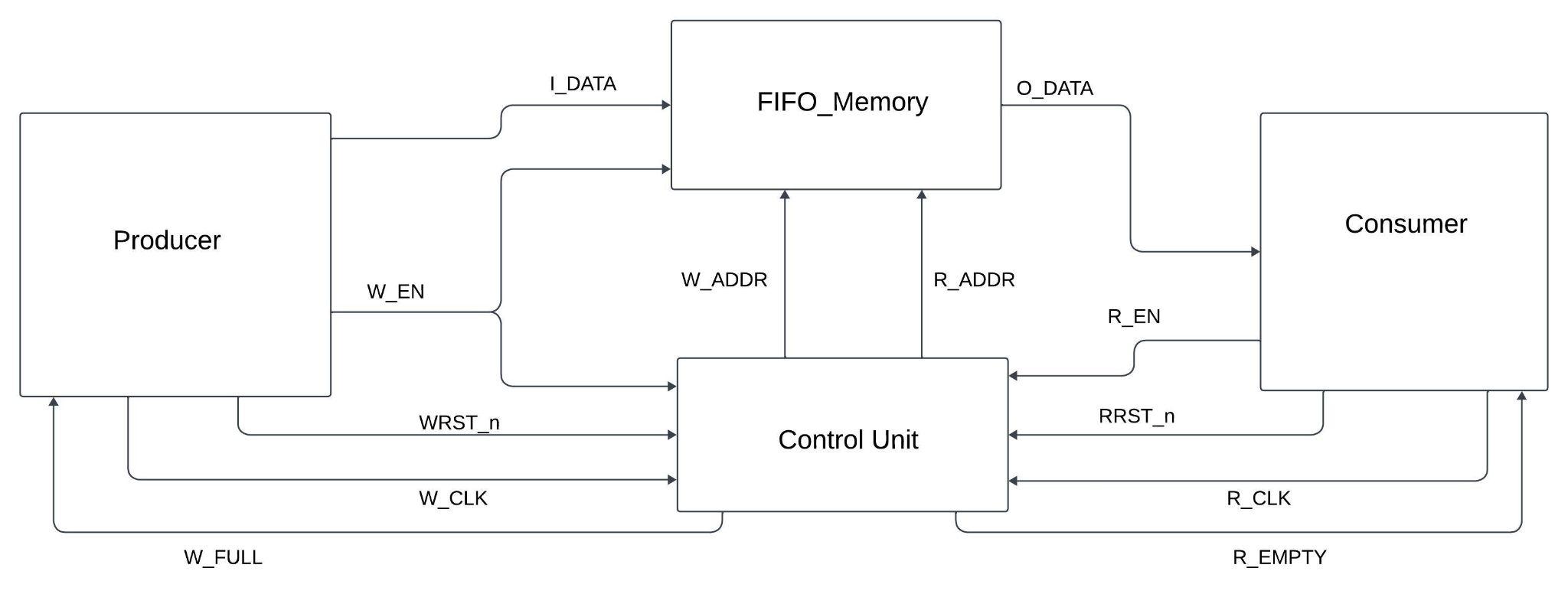
The Asynchronous FIFO is a single component that will be used within systems, chips, SOC’s, and FPGA’s. It’s commonly used in video processing for their clock domain crossings. In the image below the Asynchronous FIFO is in the middle, and there is a producer and consumer connected to the write and read 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. 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. 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 FullFIFO 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

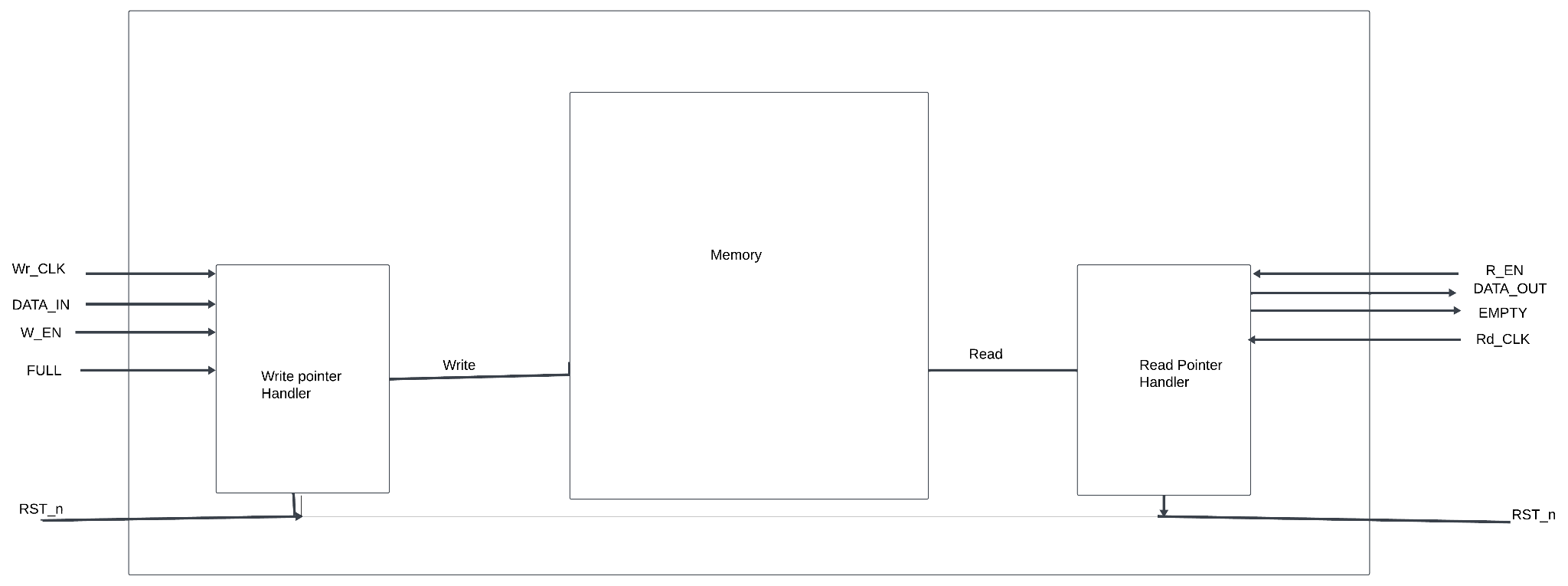
It is assumed that the user will have a basic understanding of SystemVerilog and Questasim to properly run this module. It is assumed that the external devices will work properly and that they have already been verified.

# 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. FIFO Internal Block Diagram

### 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 |

#### 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 Calculations

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:

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 reader is allowed enough time to read the remaining entries before the writer sends more data. The depth will be an unpacked array of size 333 of packed type logic 8 bits wide.

## Memory Pointer Controller

The memory pointers start at 0. Every time the module writes or reads it increments the respective memory pointer by one. If there is a reset the points are reset to the starting location of memory.

## Write/Read Idle Controller

A counter keeps track of the idle cycles. The write counter and read counter are separate and only increment at the positive edge of their respective clocks. For writes there are two idle cycles, so on the third clock cycle the internal write signal is asserted to indicate a write. If the memory is full the write signal will not be asserted and the idle counter is reset. For reads there is only one idle cycle, so on the second clock cycle the internal read signal is asserted. If memory is empty the idle counter is restarted at zero and must be idle for another cycle until attempting to read again. On reset the idle cycle clock counters are reset to zero.

## Write/Read Controller

At the positive edge of the previously discussed internal write signal, it checks to see if the device is not in reset mode, and it verifies that write, write enable are asserted and full is deasserted before writing to the current write pointer location in memory. On a successful write, the write counter is incremented by one. At the positive edge of the previously discussed internal read signal, it checks to see if the device is not in reset mode, then checks if read and read enable are asserted and empty is deasserted. If all of these conditions are met, the device will place the current read pointer data of memory on the data out port. On reset, the entirety of memory is reset to all zeroes, and the write and read pointers are reset to zero.

## Full and Empty Controller

### Wrapped Memory

There is an internal flag that keeps track of if there is wrapped memory. This bit is the decider if the memory is empty or if it is full. The read point should never surpass the write pointer, so the first pointer to reach the end of memory is the write point. Once it reaches the end and attempts to write (assuming memory is NOT full) the point must wrap to the beginning of memory. During this scenario, the wrapped memory bit is asserted. When the read pointer wraps memory and is now pointing at the 0th location of memory, the wrapped memory flag is deasserted.

### Full Logic

To assert the full logic the write pointer and read pointer must be pointing to the same location in memory, but the wrapped memory flag must be asserted.

### Empty Logic

To assert the empty logic signal the write pointer and the read pointer must be pointing to the same location in memory, but the wrapped memory flag must be deasserted.

### Reset

On reset the memory is reset and the pointers will go back to their original position at the start of memory location 0, meaning the empty signal will be asserted on and after reset until the FIFO receives reads.

## 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 system verilog code, and UVM files.

## 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 PROD\_CLK,  input CON\_CLK,  input logic RST\_n,  input logic [P\_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; bit wrapped\_mem = 1'b0;  *// Memory* logic [P\_DATA\_WIDTH-1:0] memory [P\_MEM\_DEPTH-1:0];  *// Write/Read Logic* integer wr\_clk\_cnt = 0; integer rd\_clk\_cnt = 0; logic write = 1'b0; logic read = 1'b0;  always @(posedge PROD\_CLK or negedge RST\_n) begin  if (RST\_n == 1'b0) begin  wr\_clk\_cnt <= 0;  end   else if(W\_EN) begin  wr\_clk\_cnt = ((wr\_clk\_cnt+1) % 3);  end  else begin  wr\_clk\_cnt <= 0;  end end  always @(posedge CON\_CLK or negedge RST\_n) begin  if (RST\_n == 1'b0) begin  rd\_clk\_cnt <= 0;  end   else if (R\_EN) begin  rd\_clk\_cnt = ((rd\_clk\_cnt + 1) % 2);  end  else begin  rd\_clk\_cnt <= 0;  end end  always @(posedge PROD\_CLK or negedge RST\_n) begin  if (RST\_n == 1'b0) begin  write <= 0;  end   if (wr\_clk\_cnt == 2) begin  if (!FULL) begin  if (W\_EN) begin  write <= 1'b1;  end  end  end   else begin  write <= 1'b0;  end end  always @(posedge CON\_CLK or negedge RST\_n) begin  if (RST\_n == 1'b0) begin  read <= 0;  end   if (rd\_clk\_cnt == 1) begin  if (!EMPTY) begin  if (R\_EN) begin  read <= 1'b1;  end  end  end   else begin  read <= 1'b0;  end  end   *//---------------------------*  *// Always block for writes*  *//---------------------------*  always @(posedge write 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 (write && 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 @(posedge read or negedge RST\_n) begin   if (RST\_n == 1'b0) begin  DATA\_OUT = 'z;  rd\_ptr = '0;  end  else begin  if (read && 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 |
| --- |

### Code: System Verilog RTL

| **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 | Clk | Producer Clock |
| **CON\_CLK** | In | 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 |

### Table: 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.

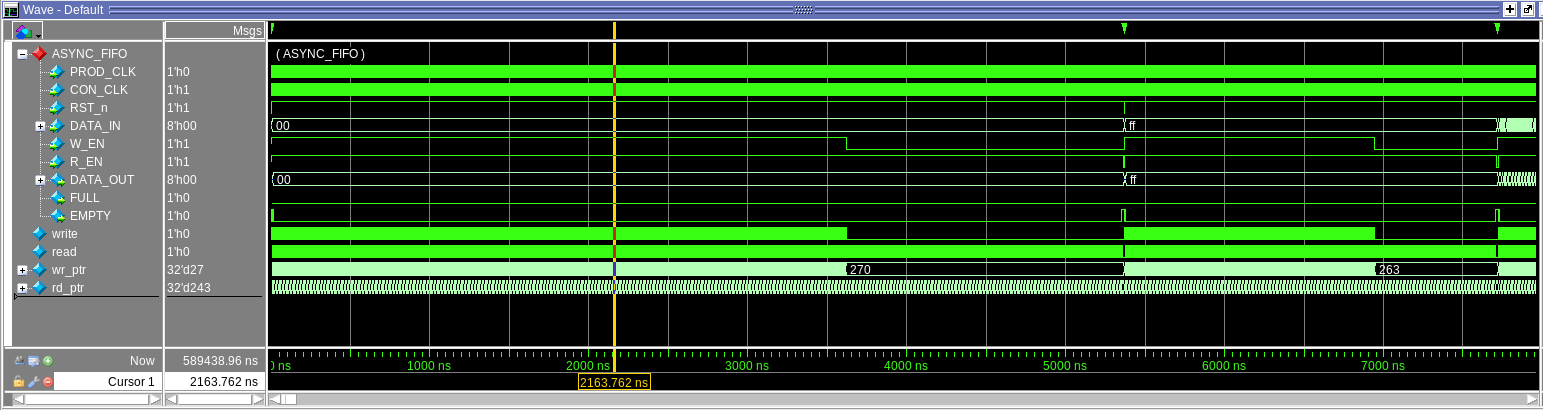
### Pointer Control

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



### 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

Every comparison has passed and the log file can be found in the sim/ directory of the project files. The tag log is named *ccd\_tag\_log.log* and shows the results from cdd\_test. The transcript shows the UVM information between the interconnects and the number of burst items sent and detected. The only missing piece is the scoreboard as this generates log data, so the sequence item write function does not state it received and item, but it does generate a log file to prove it is working properly.

## Code 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\_code\_coverage.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% |
| --- |

#### Fig. Code Coverage Results

## Function Coverage

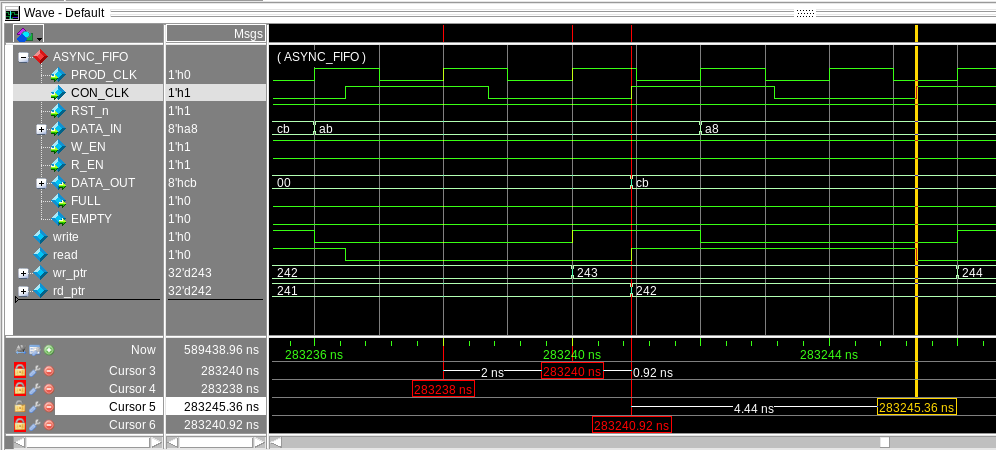
100% function coverage was achieved by monitoring each interface signal and measuring every value. The data in and data out signals were broken into three categories, all zeros, all ones and others. The names refer to the value of the bits on the interface bus. The remaining interface signals were single 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 *final.txt*.

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

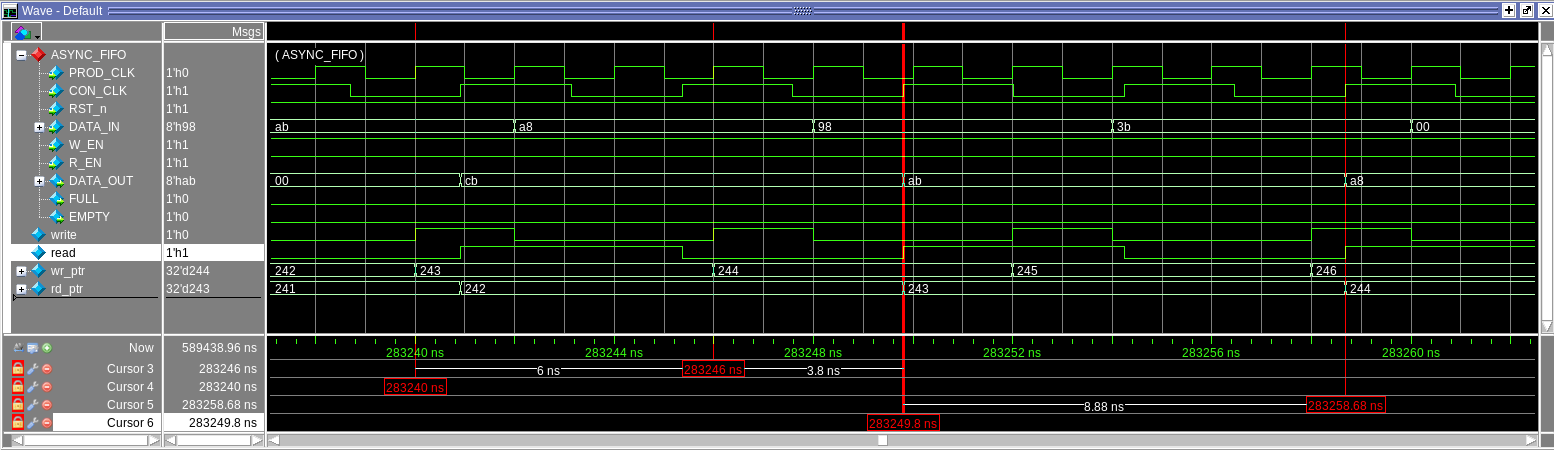
#### Fig. Functional Coverage Results

## Verified by Inspection

Due to the lack of time and members working on the project some specifications never received any official test tags, but if we were to add more tags it would be to include checks on the producer and consumer clocks verifying their correct frequencies, and that in nominal read and write situations it the reads and writes would wait the idle cycle time frame. The below figures describe the manual inspection using the VSIM waveform and by adding cursors between write and read assertions.



#### Fig. 500MHz (2ns period) & 225MHz (4.44ns) Clock Verification



#### Fig. Write Idle Inspection (6ns) & Read Idle Inspection (8.88ns)

## Issue Tracker / Findings

### Timing

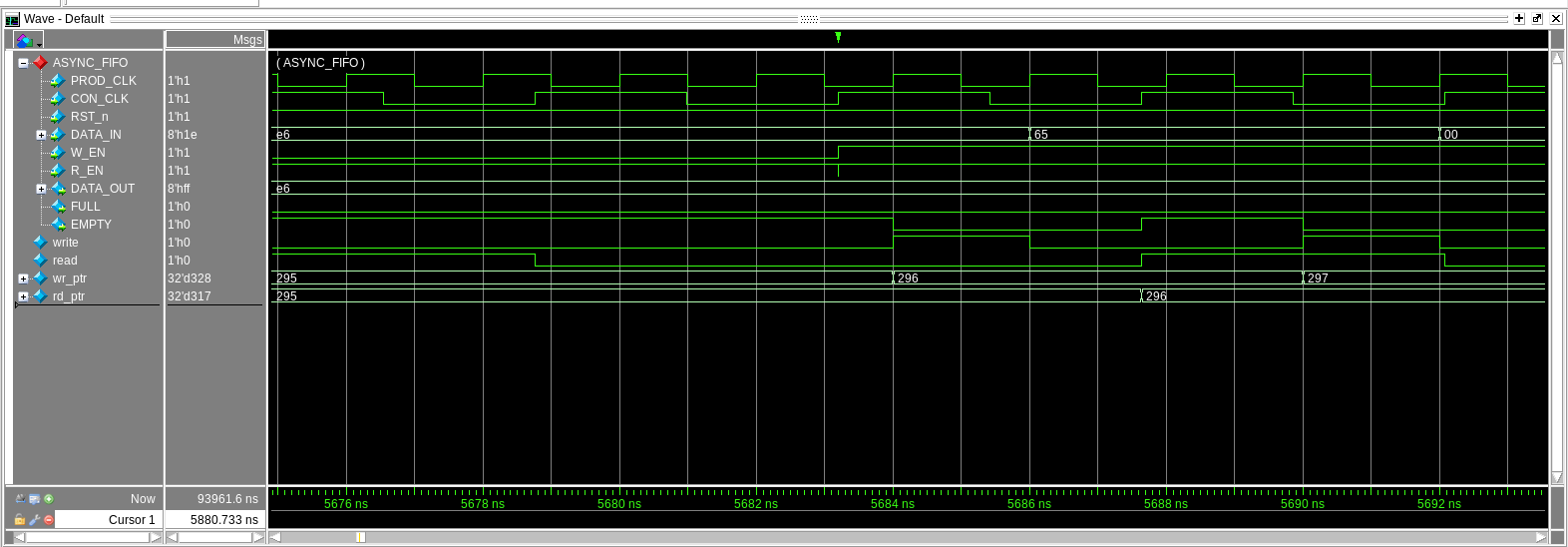
A massive amount of time was spent getting the perfect timing between the monitor and the RTL. It was much more challenging after rewriting the RTL to hold the enable lines high for the entirety of the burst. Initially, it was expected that the design would wait when there was an empty or full memory and the device was attempting to read or write, but that proved to be more difficult to code, so the design decision to change the RTL to reset the idle cycles if the device could not accept a read or a write was implemented. This simplified the nested loops that took over the UVM driver and monitor. Since the enable signals were controlled by simulation and both consumer and producers had different clock speeds, the reset deassertion and enable assertions were not perfectly timed, and if they occurred between a write and read clock cycle the device would be ahead of the writes right away. Having it wait for data caused issues, so resetting the counter for idle cycles felt necessary. It appeared to work perfectly fine in RTL, but the UVM monitor was the biggest problem.

### 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.

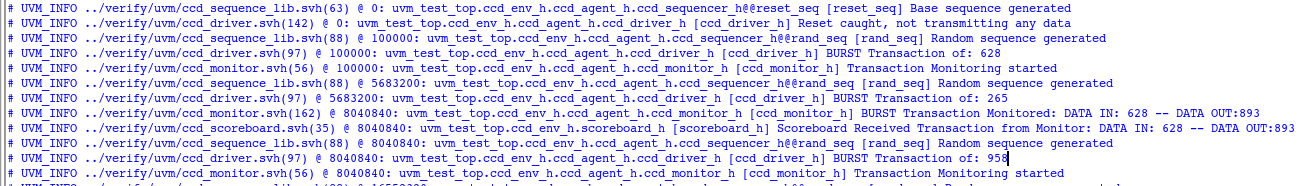
### Strobing WR/RD Enable

The original RTL design was to strobe write and read enable to indicate the asynchronous FIFO to write or read. It was designed to be truly asynchronous without providing any clocks to it. This was successfully done, and the external devices used the respective clocks to strobe the enable signals according to the specifications. After the UVM architecture was completed it was learned that this is not the behavior of the asynchronous FIFO and that the enable signals should be asserted for as long as the burst needs then be deasserted. The issue that was occurring was that back to back bursts were not supported because the UVM driver needed needed special logic to know when the burst has completed as the enable signals strobed, so it waited until an expected enable strobe and if it did not occur it would assert a flag and disable reading or writing. When both flags were asserted the scoreboard would receive the sequence item for comparisons.



#### Fig. Strobing WR/RD Enable Back-to-back burst sequences

The back-to-back sequences would not allow enough time for the monitor to register it had finished a transaction and the driver would start again and it would combine the sequences.



#### Fig. Strobing WR/RD Enable Back to back burst sequences transcript

### UVM Architecture

The issues when converting the OOP code to a UVM test environment was more challenging than anticipated. Deciding which UVM component/object to start with was challenging, as many of them instantiate or are connected to each other. At this point it was necessary to attempt to fill out each, so the compiler can direct us to the most present compiling error. An agent configuration was attempted that would use the UVM config database to get the only virtual interface. This would limit the amount of manual configuration database gets for the virtual interface, but this presented a challenge, and the route that was decided was to assign the virtual interface in the agent component.

### Parameterization

The RTL and the OOP code was designed to be parameterized for the data width and the memory depth, but this was beyond the scope of what has been implemented in the UVM code. The environment, top, environment and agent all have hard coded values modeled for the expected parameter value used in the RTL of a data width of 8 bits.

# 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: 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