High Level Design Specification (HLDS)

for

ECE593 Winter 24 Homework

Version 0.4

Prepared by Masaaki Ishii

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

01-10-24

Table of Contents

Table of Contents ii

Revision History iii

1. Introduction 1

1.1 Purpose 1

1.2 Document Conventions 1

1.3 Intended Audience and Reading Suggestions 2

1.4 Product Scope 2

1.5 References 2

2. Overall Description 3

2.1 Product Perspective 3

2.2 Product Functions 3

2.3 User Classes and Characteristics 4

2.4 Tools and Software 4

2.5 Design and Implementation Constraints 4

2.6 Assumptions and Dependencies 5

3. External Interface Requirements 5

3.1 Hardware Interfaces 5

3.2 Software Interfaces 7

4. Product Features 7

4.1 FIFO Memory Buffer 7

4.2 Synchronizer Module (Read-domain to Write-domain Synchronizer) 7

4.3 Synchronizer Module (Write-domain to Read-domain Synchronizer) 8

4.4 Read Pointer and Empty Generation Logic 8

4.5 Write Pointer and Full Genration Logic 9

4.6 FIFO Depth and Timing 9

5. Logic Design 10

5.1 <Your work Directory Structure> 3

5.2 <Design modules> 3

5.3 <SystemVerilog abstraction Features used> 3

5.4 <Simulation, Tools, Directory Structure> 3

6. Verification 3

6.1 Testbench Style 4

6.2 Testing Strategies 4

6.3 Test case scenarios 4

6.4 Others 4

Summary 4

Appendix A: Glossary 4

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Date** | **Reason For Changes** | **Version** |
| Masaaki Ishii | 1-10-24 | Draft Version. Finished Draft of Section 1.1-1.5, 2.1-2.4, and 2.6 | 0.1 |
| Masaaki Ishii | 1-11-24 | Draft Version. Added Section 4.6  Finished Draft of Section 3.1-3.2 and 4.1-4.6 | 0.2 |
| Masaaki Ishii | 1-13-24 | Draft Version. Revise sections 1, 2, 3, and 4 | 0.3 |
| Masaaki Ishii | 1-14-24 | Draft Version. Revises sections 1, 2, 3, and 4 | 0.4 |
|  |  | **Will not have Version 1.0 until after Homework 3** |  |

# Introduction

## Purpose

The purpose of this High-Level Design Specification (HLDS) is to detail one method that is used to design, synthesize and analyze an Asynchronous FIFO with different clock domains using Gray code pointers. This document can be used as a reference manual on how to implement an Asynchronous FIFO design using Gray code pointers into a system with two different clock domains or as a reference on the operations of an Asynchronous FIFO. All the code, diagrams, and design are from Clifford E. Cummings Asynchronous FIFO design from his paper Simulation and Synthesis Techniques for Asynchronous FIFO Design.

## Document Conventions

* Bullet points rather than numbering will be used.
* Section headers will be:
  + Font: Times
  + Size: 18
  + Bold
  + Align: Left
* Subsection headers will be:
  + Font: Times
  + Size: 14
  + Bold
  + Align: Left
* Subsection body will be:
  + Font: Arial
  + Size: 11
  + Align: Left
* Figures will be:
  + Font: Arial
  + Size: 9
  + Alight: Center
  + Italicized
* Images will be center aligned.
* The paper will use “Asynchronous FIFO” and “FIFO” interchangeably.
* The paper will use “producer” and “consumer” interchangeably as synonyms for “system1” and “system2”.
* Italics identifies:
  + Titles of books, journals, webpages that are references.
  + Also used to indicate key terms or phrases that are accompanied by a definition.
  + General scientific names, species, and varieties.
  + Figure and table captions
* Underline identifies:
  + Hyperlinks
* Bold identifies:
  + The paper’s title, headings, and section/subsection labels
* Highlights identifies:
  + Changes in the most recent revision
* References will be cited using the [reference number] method.
  + Relevant information from reference [#].
  + References will be italicized.

## Intended Audience and Reading Suggestions

This document is meant for design verification engineers that want to implement an Asynchronous FIFO design using Gray code pointers or for individuals interested in the implementation of an Asynchronous FIFO design.

## Product Scope

The HLDS documentation presents the structure of the Asynchronous FIFO and how it fits into systems where data is needed to pass from one clock domain to another system with a different clock domain. Asynchronous FIFO helps to synchronize data flow between two systems (SoC, CPUs, Network chips, etc.) working on different clocks with different data rates. For example, communication network bridges, switches and routers used in computer networks.

## References

[1] Clifford E. Cummings, “Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs,” SNUG 2001 (Synopsys Users Group Conference, San Jose, CA, 2001) User Papers, March 2001, Section MC1, 3rd paper.

[2] Clifford E. Cummings and Peter Alfke, “Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons,” SNUG 2002 (Synopsys Users Group Conference, San Jose, CA, 2002) User Papers, March 2002, Section TB2, 3rd paper

[3] “FIFO (Computing and Electronics).” Wikipedia, Wikimedia Foundation, 2 Sept. 2023, en.wikipedia.org/wiki/FIFO\_(computing\_and\_electronics).

[4] Vlsiverify. (2022, December 26). Asynchronous FIFO. VLSI Verify.https://vlsiverify.com/verilog/verilog-codes/asynchronous-fifo/

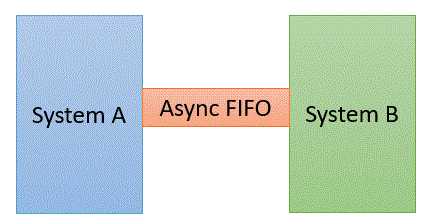
[5] https://research.ijcaonline.org/volume86/number11/pxc3893347.pdf

[6] https://hardwaregeeksblog.files.wordpress.com/2016/12/fifodepthcalculationmadeeasy2. pdf

[7] Ramesh, G. (n.d.). Asynchronous FIFO Design with Gray code Pointer for High Speed AMBA AHB Compliant Memory controller. IOSR Journal of VLSI and Signal Processing (IOSR-JVSP), 1(3), 32–37. https://doi.org/www.iosrjournals.org

# Overall Description

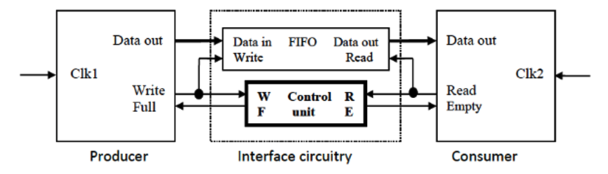
## Product Perspective



## *Figure 1. Simple Block Diagram of Asynchronous FIFO Connecting Two Systems Working on Different Clocks*

In computing and systems theory, First-In-First-Out is a method for organizing the manipulation of a data structure where the oldest (first) entry is processed first [3]. Asynchronous First-In-First-Out (Asynchronous FIFO) refers to a FIFO design where the data values are written to a FIFO buffer from one clock domain (System A) and the data values are read from the same FIFO buffer from another clock doman (System B), where the domains are asynchronous to each other which can introduce metastability issues. FIFOs are used to safely pass data from one clock domain to another asynchronous clock domain, or in simple terms, passing data between two systems with different data rates (clocks) [1]. Using a FIFO to pass data from one clock domain to another clock domain requires multi-asynchronous clock design techniques. This specification will detail one method that is used to design, synthesize and analyze a FIFO with different clock domains using Gray code pointers.

## Product Functions



*Figure 2. Top-level Block Diagram of the Asynchronous FIFO and Two Systems*

* FIFO Block
  + Asynchronous FIFO Pointers
    - Write address pointer (Data in Write)
    - Read address pointer (Data out Read)
* Control Unit
  + Write Pointer Handler
    - Write Pointer:
      * Dual N-bit Gray code counter
        + N-bit Gray code counter pointer (wptr)
        + (N-1)-bit Gray code counter pointer (waddr) used to address FIFO buffer
    - FIFO full: When the write address pointer reaches the read address pointer.
    - FIFO almost full: When multiple entries are written.
  + Read Pointer Handler
    - Read Pointer:
      * Dual N-bit Gray code counter
        + N-bit Gray code counter pointer (rptr)
        + (N-1)-bit Gray code counter pointer (raddr) used to address FIFO buffer
    - FIFO empty: When the read address pointer reaches the write address pointer. Or when the FIFO is initialized or reset is asserted and both the read and write address pointers are at the first memory location.
    - FIFO almost empty: When multiple entries are read.
* Write Enable Signal: Write is enabled.
* Read Enable Signal: Read is enabled.
* Clock: data will be read/written at the positive edge (posedge) of the clocks. (wclk and rclk)
* Reset (wrst\_n and rrst\_n): De-assert all active signals. Clear FIFO contents. Assert FIFO empty. Both the read and write address pointers are at the first memory location.

## User Classes and Characteristics

Any individual that is attempting to synchronize multiple changing signals from one clock domain into a new clock domain and ensuring that all the changing signals will be synchronized to the same clock cycle in the new clock domain would be interested in using this Asynchronous FIFO implementation.

## Tools and Software

* All the programming will be done in SystemVerilog on QuestaSim using Portland State University’s remote computer lab.
* The processor on the provided remote computers is an Intel(R) Xeon(R) CPU E3-1241 v3 @ 3.50 GHz 3.50 GHz with 16 GB of installed RAM.
* The operating system running is Windows 10 Enterprise version 22H2 and is a 64-bit operating system (x64-based processor)
* Questa Sim is a multi-language environment by Siemens (previously developed by Mentor Graphics) for simulation of hardware description languages such as VHDL, Verilog and SystemVerilog.
* The version of Questa Sim that is on the remote computers provided by Portland State University is Questa Sim-64 2023.3. Revision 2023.07 Date: Jul 17 2023.

## Design and Implementation Constraints

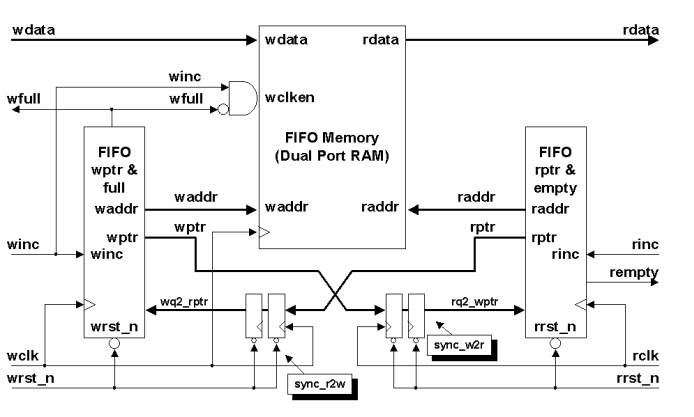
The FIFO implementation that is described in the HLDS is meant for a design using two different systems and is not meant for a design using any more than two or any less than two. The Gray code counter that is used in the HLDS must have a power-of-2 count in the sequence and cannot be of odd-length count. Users should also be aware that this design has also only been implemented on the hardware/software described in section 2.4 Tools and Software and is not verified to work on other hardware/software.

## Assumptions and Dependencies

* The FIFO operates under the following conditions:
  + Producer clock frequency: 1 GHz
  + Consumer clock frequency: 500 MHz
  + Duty-cycle of 50%
  + Max write burst size: 200
  + Number of idle cycles between successive write: 2
  + Number of idle cycles between successive reads: 4
* The Gray code counter must have power-of-2 counts in the sequence.
* The Gray code counter cannot be of odd length count
* The FIFO will be 2^n deep.
* Reset and Initialization of the FIFO should trigger FIFO empty signal.
* Write entry to FIFO de-asserts FIFO empty signal.
* Read entry when the FIFO is full de-asserts the FIFO full signal.
* Write multiple entries asserts the FIFO almost full or FIFO full signal.
* Clock: data will be read/written at the positive edge (posedge) of the clocks. (clk1 and clk2)
* Reset: De-assert all active signals. Clear FIFO contents. Assert FIFO empty. Both the read and write address pointers are at the first memory location.

# External Interface Requirements

## Hardware Interfaces



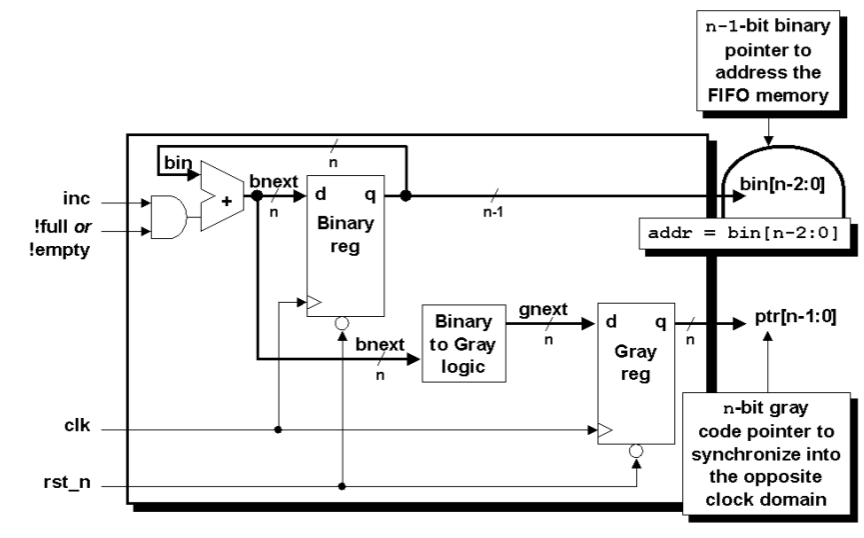
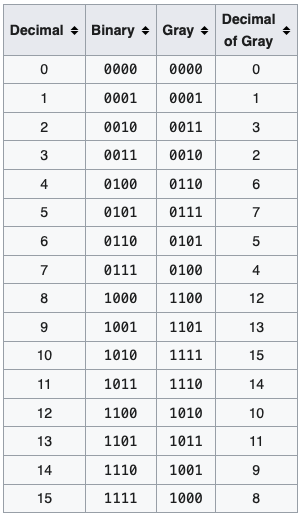
*Figure 3. Detailed Block Diagram of the Asynchronous FIFO that is Used to Connect Two Systems [1]*

Using the Asynchronous FIFO designed by Clifford E. Cummings there are six SystemVerilog modules that will be used in the design described in this HLDS.

* fifo1.sv: This module is the top-level module that includes both the clock domains from the producer and consumer systems. The top-level module is used to instantiate all of the other FIFO modules that are used in the design [1].
* fifomem.sv (FIFO Memory from Figure 3): This module is the FIFO memory buffer that is used by both the producer (system1) and consumer (system2) clock domains [1].
* sync\_r2w.sv (sync\_r2w from Figure 3): Is the synchronizer module that is used to synchronize the read pointer into the write-clock domain. In this module the read pointer that is synchronized will be used in the wptr\_full.sv module to generate the FIFO full condition. This module only contains flip-flops that are synchronized to the write clock (producer) [1].
* sync\_w2r.sv (sync\_w2r from Figure 3): Is the synchronizer module that is used to synchronize the write pointer into the read-clock domain. In this module the write pointer that is synchronized will be used in the rptr\_empty.sv module to generate the FIFO empty condition. This module only contains flip-flops that are synchronized to the read clock (consumer) [1].
* rptr\_empty.sv (FIFO rptr & empty block from Figure 3): Is the module that contains the FIFO read pointer, empty-flag and almost empty-flag logic [1].
* wptr\_full.sv (FIFO wptr & full block from Figure 3): Is the module that contains the FIFO write pointer, full-flag, and almost full-flag logic [1].

There is also a reset function that simultaneously and asynchronously resets the wclk-domain (producer domain) and the rclk-domain (consumer domain)

The Gray code counter style that is used in this design is based on Cummings [1] design and will be referred to as a “dual n-bit Gray code counter” which comprises of a n-bit Gray code counter and an (n-1)-bit Gray code counter. The Gray counter is what enables the interactions to occur between the two systems with different clock domains. The code distance between any two adjacent words is just 1, only one bit can change from one Gray count to the next. The Gray code counter must have a power-of-2 counts in the sequence and there are no odd-count-length Gray code sequences. The Gray code counter used in the FIFO will be 2^n deep. The left-hand side of figure 4. shows the workings of a 4-bit gray code counter and how the bits in a Gray ccode counter change. While the right-hand side of figure 4. shows a Gray code counter style that uses two sets of registers, one binary and the other one to for a binary-to-Gray converted value [1]. This Gray code counter utilizes the binary carry structure and simplifies the Gray-to-binary conversion, reducing the combinational logic required. The Gray code counter is used for both the read pointer and empty generation logic as well as the write pointer and full generation logic blocks.



*Figure 4. Gray Code Conversion and Dual n-bit Gray Code Counter Block Diagram*

## Software Interfaces

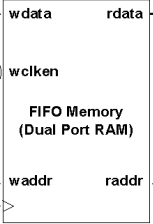
* All the programming will be done in SystemVerilog on QuestaSim using Portland State University’s remote computer lab
* The processor on the provided remote computers is an Intel(R) Xeon(R) CPU E3-1241 v3 @ 3.50 GHz 3.50 GHz with 16 GB of installed RAM.
* The operating system running is Windows 10 Enterprise version 22H2 and is a 64-bit operating system (x64-based processor)
* Questa Sim is a multi-language environment by Siemens (previously developed by Mentor Graphics) for simulation of hardware description languages such as VHDL, Verilog and SystemVerilog.
* The version of Questa Sim that is on the remote computers provided by Portland State University is Questa Sim-64 2023.3. Revision 2023.07 Date: Jul 17 2023.
* Included Libraries on Questa Sim: See Appendix for full library contents.



*Figure 4. Questa Sim-64 2023.3 Included Libraries*

# Product Features

## FIFO Memory Buffer

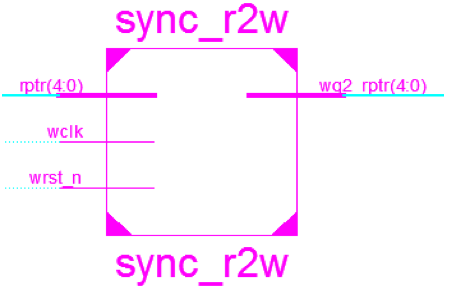
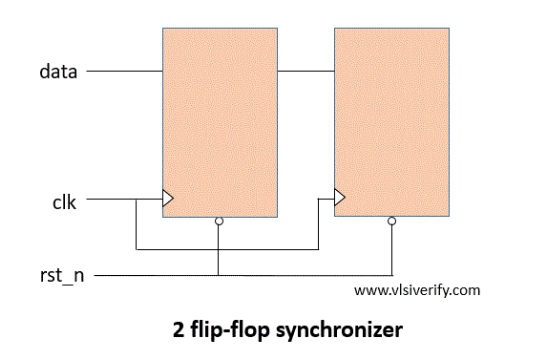


This is the memory buffer that is accessed by both the write and read clock domains. The FIFO memory buffer has a set data size (memory data word width) and address size (number of memory address bits). The inputs to the FIFO memory buffer is the data to be written (wdata), the address to be written (waddr), the address to be read (raddr), the clock enable signal from the producer (wclken), the write buffer is full signal (wfull), and the clock signal from the producer (wclk). The output of the FIFO memory buffer is the data to be read (rdata). The output rdata is assigned the value from raddr and at the positive edge of the producer clock, the waddr is assigned the value from wdata [1].

*Figure 5. FIFO Memory*

## Synthesizer Module (Read-domain to Write-domain Synchronizer)

Synchronizer module that is used to synchronize the read pointer into the write clock domain through a pair of registers that are clocked by the write-clock. The synchronized read pointer (rptr) will be used by the wptr\_full module to generate the FIFO full condition This module only contains flip-flops that are synchronized to the write clock. No other logic is included in this module [1]. The read pointer uses the Gray code counter logic discussed in *Section 3.1.*



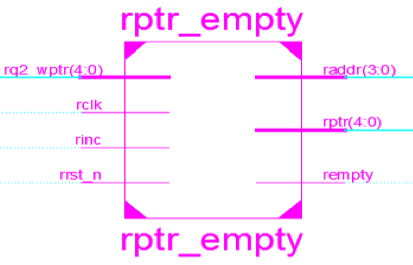
*Figure 6. 2 Flip-flop Synchronizer [5]*

## Synthesizer Module (Write-domain to Read-domain Synchronizer)

There is another synchronizer module that is used to synchronize the write pointer into the read clock domain through a pair of registers that are clocked by read-clock. The synchronized write pointer will be used by the rptr\_empty module to generate the FIFO empty condition. This module contains flip-flops that are synchronized with the read clock. No other logic is used in this module [1]. The write pointer uses Gray code counter logic discussed in Section 3.1. The same 2 flip-flop synchronizer is used for the Write-domain to Read-domain Synchronizer as the Read-domain to Write-domain Synchronizer used in section 4.3, Figure 6, except that the input write/read will be reversed. For example, rptr = wptr, wclk = rclk, wq2\_rptr = rq2\_wptr, etc.

## Read Pointer, Empty Generation Logic, and Almost Empty Logic

This module is completely synchronous to the read clock domain and contains the FIFO read pointer and empty flag logic [1]. The read operation is controlled by the read pointer which points to the location that the next data is read from. This module uses the dual n-bit Gray code counter discussed in *Section 3.1* to generate the read pointer and read address values. When the FIFO read operation takes place and the rptr signal is incremented, the FIFO pointers are no longer equal t o each other so the full signal is de-asserted. Because the two empty and full signal changes occur on two different clock domains, the 2 flip-flop synchronizer described in *Section 4.2, Figure. 6,* is needed to remove the metastability that could be generated [7]. The FIFO design in this paper assumes that the empty flag will be generated in the read-clock domain to ensure that the empty flag is detected immediately when the FIFO buffer is empty, that is, the instant that the read pointer catches up to the write pointer (including the pointer MSBs) [1]. The empty signal is asserted when the synchronized read and write pointer are equal. To do the empty comparison, pointers that are one bit larger than needed for the FIFO memory buffer are used, and if the n+1 bit of the read and write pointers are equal to each other, that means that the pointers have wrapped around the FIFO buffer the same amount of times and a comparison of the rest of the pointer’s bits are done to see if they are equal. If the bits are all equal the FIFO is assumed to be empty [1]. The almost empty signal is asserted when the read pointer is incremented multiple times without new writes.

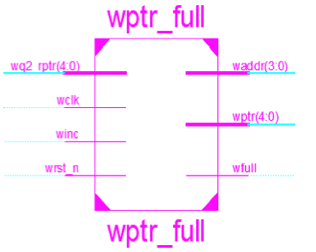


*Figure 7. Read Pointer and Empty Generation Logic [5]*

## Write Pointer, Full Generation Logic, and Almost Full Logic

This module is completely synchronous to the write clock domain and contains the FIFO write pointer and full flag logic [1]. The write operation is controlled by the write pointer which points to the memory location where the next data is written to. This module uses the dual n-bit Gray code counter discussed in *Section 3.1* to generate the write pointer and write address values. When the FIFO write operation takes place and the wptr signal is incremented, the FIFO pointers are no longer equal to each other so the empty signal is de-asserted. Because the two empty and full signal changes occur on two different clock domains, the 2 flip-flop synchronizer described in *Section 4.2, Figure. 6,* is needed to remove the metastability that could be generated [7]. The FIFO design in this paper assumes that the full flag will be generated in the write-clock domain to insure that the full flag is detected immediately when the FIFO buffer is full, that is, the instant that the write pointer catches up to the read pointer (except for different pointer MSBs) [1]. The full signal is asserted by running a comparison between the write and read pointers after they have been synchronized. To ensure that there are no issues, a comparison is made after synchronizing the read pointer into the write clock domain. After that, three comparisons are made. The first is if the write pointer and the read pointer’s most significant bit (MSB) are not equal. Second, comparing the write pointer and read pointer’s 2nd MSB are not equal. Finally, checking if all the other write pointer and read pointer bits are equal [1]. The almost full signal will be asserted when the write pointer has been incremented multiple times in a row.

*Figure 8. Write Pointer and Empty Generation Logic [5]*



## FIFO Depth and Timing

The FIFO depth and timing information was calculated using information from reference 6, *Calculation of FIFO Depth Made Easy by Putta Satish.* Since the FIFO is working with two modules of different clock domains, we need to calculate the depth that is required of the FIFO since the producer system is faster at writing to the buffer than the consumer system is at reading data from the buffer. The depth of the FIFO is determined in a way, so that the FIFO can store all the data which cannot be read by the slower consumer module [6].

FIFO operating conditions:

* Producer clock frequency = 500 MHz
* Consumer clock frequency = 225 MHz
* Duty-cycle of 50%
* Max Write Burst Size = 1024
* Number of idle cycles between successive writes = 2
* Number of idle cycles between successive reads = 1

The number of idle cycles between two successive writes is 2 clock cycles, which means that after writing one data, the producer module is waiting for 2 clock cycles to initiate the next write. Thus, it can be understood that for every 3 clock cycles, one data is written.

The number of idle cycles between two successive reads is 1 clock cycles, which means that after reading one data, the consumer module is waiting for 1 clock cycles to initiate the next read. Thus, it can be understood that for every 2 clock cycles, one data is read.

Time required to write one data item = 3 \* (1/500MHz) = 6 ns.

Time required to write all the data in the burst = 1024 \* 6ns = 6144 ns.

Time required to read one data item = 2 \* (1/225MHz) = 8.89 ns.

For every 8.89 ns, the consumer module is going to read one data in the burst.   
  
In a period of 6144 ns, 1024 number of data items can be written.

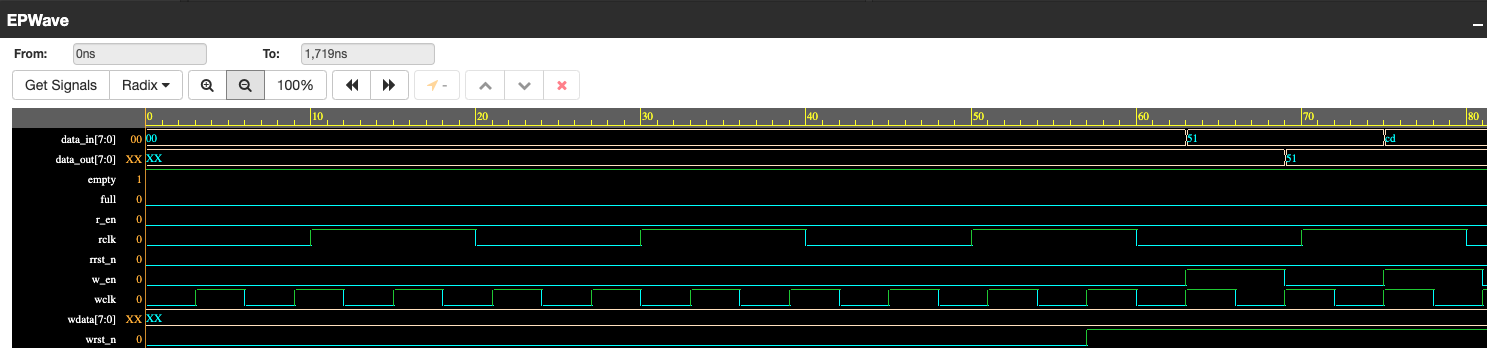
The number of data items that can be read in a period of 6144 ns = (6144ns/8.89ns) = 691.2

The remaining number of bytes to be stored in the FIFO = 1024 – 691 = 333.

The FIFO must be capable of storing 333 data items, so the minimum depth of the FIFO should be 333.

In Figure 9. we have the timing diagram that was made on *EDA Playground,* that uses the timing for the two system clocks that was calculated above in *Section 4.6.* The timing diagram will be verified in the next segments of the assignments.

*Figure 9. Timing Diagram for FIFO Operation Simulated with EDA Playground. https://edaplayground.com/x/iQ5c*



# Logic Design

## <Your work Directory Structure>

*<How is your design and simulation repository, where to find what?>*

## <Design modules>

<what will be your hardware coding style, how many modules, how many, how they will be connected to each other, how will they interact with each other and to the outer world>

## <SystemVerilog abstraction Features used>

<List all the SystemVerilog features you are going to use to take advantage of abstraction, parametrization, scaling your design, etc.>

## <Simulation, Tools, Directory Structure>

<What simulation tools used, what aspects will be shown transcripts, waveforms etc.>

# Verification

<Describe how you plan to verify your design.>

## Testbench Style

## Testing Strategies

## Test case scenarios

## Others

Summary

Appendix A: Glossary

