Designing and Testing Asynchronous FIFO Queues

By: Tommyl404 and elemenopi

Introduction

In this project we will implement an asynchronous FIFO queue, a system that provides an intermediate to solve Clock domain crossing synchronization problems. The asynchronous FIFO is an improvement over a simple FIFO queue, by being able to avoid metastability that may appear in the calculations it does. And show its work through an improved version of an Even-Odd batcher sorter, also providing metastability containing inputs and verifying for metastability containing calculations correctness.

First-In, First-Out (FIFO) Queue

A FIFO queue is a type of data structure used in computing for managing data elements where the order of processing is strictly based on their arrival times; that is, the first element to enter the queue is the first to be processed and removed.

In digital systems, a FIFO queue is used as an intermediate between two systems s.t the first system is the input of the other and both use different clock rates. Because the rates are different, for the data to transfer from one system to enter the other it needs to be synchronized. Therefore, synchronization system is used between the two.

Essentially, the FIFO is a dual port memory with two pointers each used by read and write domain: read pointer and write pointer. Because the pointers themselves are in two different clock domains, two calculation units that communicate between each other are placed in each domain. The communication between the read and write pointer control units is direct, A problem arises:

Metastability in the communication between the read control pointer unit and write control pointer unit.

The solution: Using synchronizers and gray code calculations to decrease the probability of metastability.

our implementation of the FIFO, based on the article : Simulation and Synthesis Techniques for Asynchronous FIFO Design.

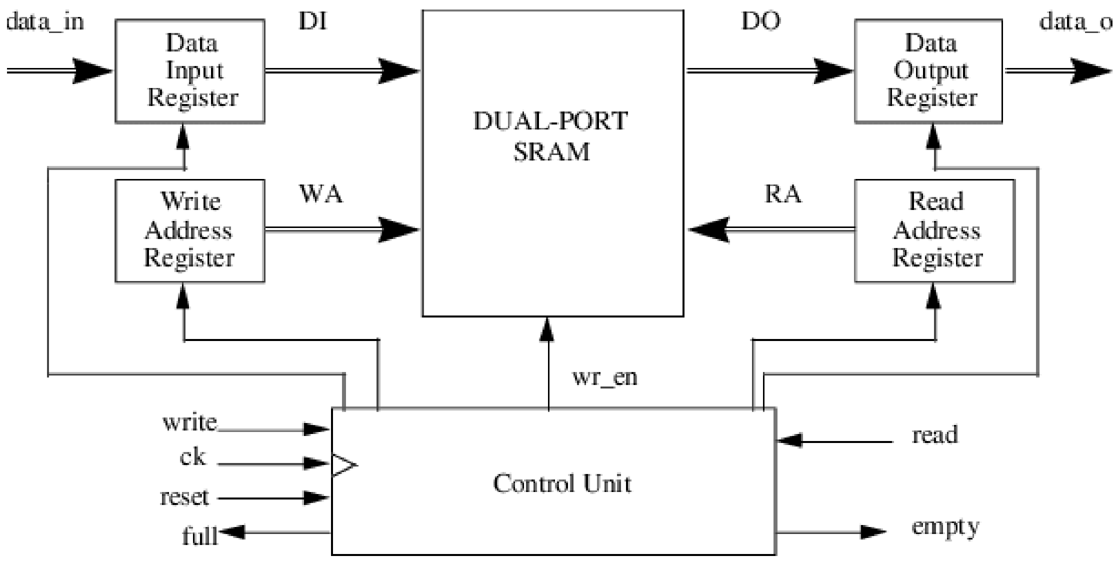

Fig 1 - Basic FIFO synchronizer block diagram

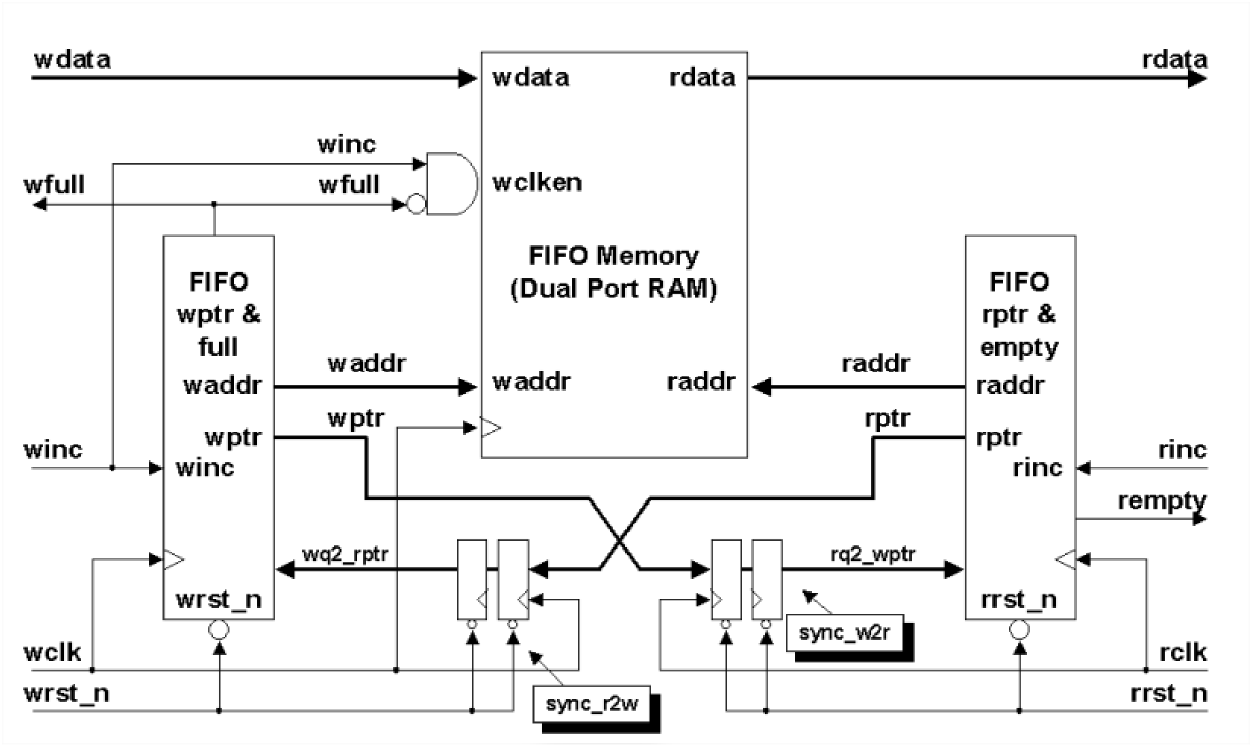

Fig 2 - Asynchronous FIFO synchronizer block diagram.

Metastability in Digital Circuits

Metastability is a critical phenomenon in digital circuits that occurs when a bistable element (like a flip-flop) receives an input signal near the transitioning threshold during its decision window. This state of uncertainty can lead to the element failing to resolve to a stable logic state (either high or low) within the required time frame, potentially causing unpredictable behavior in digital systems. Metastability is particularly problematic in asynchronous systems where timing differences can lead to signals arriving at critical moments. Expressing and handling metastability involves ensuring that the system can tolerate or resolve these states without leading to system-wide errors or failures.

In addition to timing violations, environmental factors such as temperature variations, power supply fluctuations, and electromagnetic interference can exacerbate the risks of metastability by affecting the physical properties of the electronic components and thereby influencing their timing characteristics.

The implications of metastability extend beyond the transient performance issues. In high-speed digital systems, even a single metastable event can propagate through the system, leading to widespread errors and malfunctions. For example, in a processor, a metastable state in a control signal could misdirect data flow, leading to incorrect operations or system crashes. In communication systems, it could result in the loss or corruption of critical data packets, severely impacting system reliability and performance.

Asynchronous FIFO Queues and Metastability

Asynchronous FIFO queues are specialized FIFOs designed to operate between two independent clock domains—areas of a digital system that operate under different timing constraints. These queues play a pivotal role in buffering and transferring data safely across these domains. The key challenge in designing such queues lies in their ability to handle metastability effectively. To address this, asynchronous FIFOs incorporate various techniques such as employing metastability-hardened flip-flops and designing robust control circuits that manage the read and write operations in a manner that minimizes the likelihood of metastable occurrences. This ensures data integrity and system reliability, facilitating smooth and stable operations across asynchronous interfaces.

Design of Asynchronous FIFO Queues

The design of an asynchronous FIFO queue focuses on the robust transfer of data between producer and consumer blocks operating under separate clock domains. This is achieved through a careful architectural setup that includes a memory array and read/write control logic, which operates independently under different clock regimes:

1. **Memory Structure:** The FIFO memory typically consists of a circular buffer that efficiently utilizes space to store data elements. The management of this memory involves pointers or counters that track the positions of the 'head' (write pointer) and 'tail' (read pointer) of the queue.
2. **Control Logic:** The control logic for asynchronous FIFOs includes write and read pointers that are incremented in their respective clock domains. This logic must ensure that the write operations do not overwrite unread data and read operations do not attempt to access data that has not yet been written.
   For the correct operation of the producer system and the consumer system, we need to provide them a full flag and an empty flag indicating when the FIFO is full or empty since we don't want to write to a full memory or read from an empty one. Here is how this works:

   There is a read pointer that increases on read and the write pointer increases on write, using binary code. When the conditions in *Fig 3* are used the flags are turned on.

   We will explain how it works through an example:

   An 8 bit address FIFO will be accessed through 3 bits and one extra bit is used for each pointer.

   Both pointers start at 0000. After two writes : write ptr will be at 0010 and read at 0000, after two reads both pointers will be at 0010 and the condition : wptr == rptr is met. now that the FIFO is empty we continue to fill it for 8 more writes until we reach wptr = 1010, and the condition: [not(msb),wptr[wptrlen-2:0]] == rptr is met, we can no longer write.

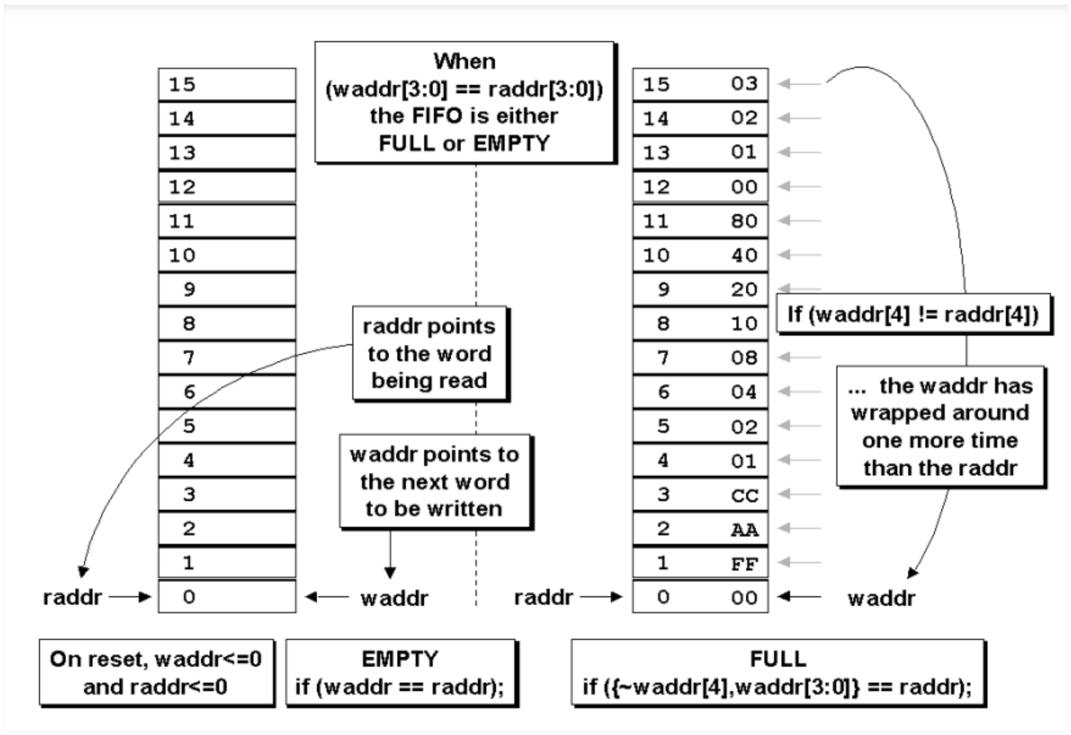
When (waddr[3:0] == raddr[3:0]) the FIFO is either FULL or EMPTY
raddr points to the word being read
waddr points to the next word to be written
If (waddr[4] != raddr[4])
... the waddr has wrapped around one more time than the raddr
raddr
waddr
On reset, waddr<=0 and raddr<=0
EMPTY if (waddr == raddr);
FULL if ({~waddr[4],waddr[3:0]} == raddr);

Fig 3 – diagram of how the “full” and “empty” flags work

Handling Metastability

Asynchronous FIFO queues are inherently susceptible to metastability due to the crossing of data and control signals between different clock domains. The design and implementation of these queues, therefore, incorporate several strategies specifically aimed at minimizing and managing metastability:

1. **Pointer Synchronization:** Since the read and write pointers are managed by different clocks, their values must be safely transferred across clock domains. This is typically done using synchronization registers or flip-flops, which help to mitigate the risk of metastability by providing a buffer period for the signals to stabilize. (add FIG)
2. **Gray Coding:** Gray coding is often used for encoding the read and write pointers before they are transferred across clock domains. Gray code ensures that only one bit changes at a time, reducing the likelihood of errors during the synchronization of pointer values and thus minimizing the risk of metastable states affecting the system.
3. **Depth and Status Flags**: The FIFO includes mechanisms to track its status—whether it is full or empty. This tracking helps prevent data corruption scenarios where writes occur to a full FIFO or reads from an empty FIFO. Such conditions, if not managed, could exacerbate metastability issues.

For the purpose of simplifying the complex system that is going to be presented in this paper, our system is going to consist of 3 players:

- The manufacturer

- The mediator

- The consumer

in this paper we are going to present a system that consists of three systems that we designed:

1. Sequence Generator - This system produces 8 bit data to be written into the Asynchronous FIFO Queue. The generator has its own clock. Our generator will be able to give outputs with or without metastable bits. This system is “The manufacturer” we mentioned earlier. It is important to mention that our generator is not really a real "random-generator". it emits series of bits that we wrote to it beforehand in its ROM memory. By also designing all these systems in VHDL, we are given the possibility to introduce metastability into the outputs of this machine with the help of "X" which is considered a metastable bit.

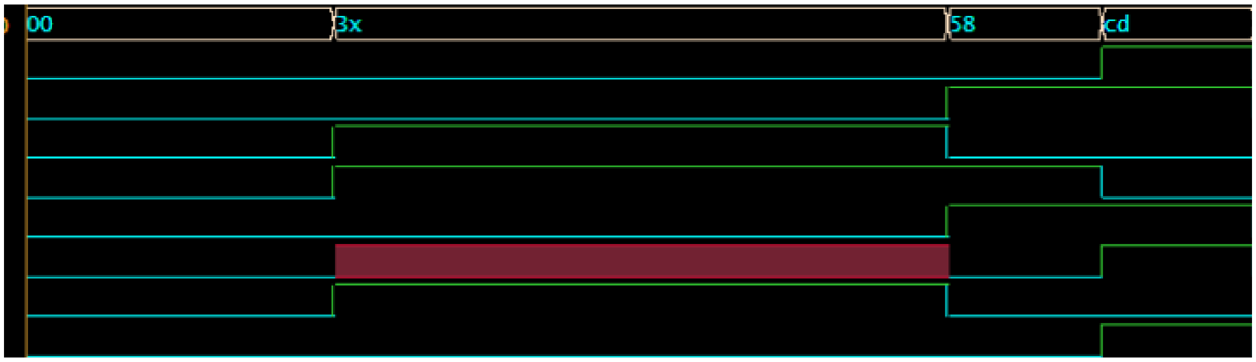

Fig 4 – how a metastable bit looks like in VHDL. (the red thick line)

2. Asynchronous FIFO Queue - The most important system for this paper. Its role will be to take the data that is fed to it, save it within the system, handle metastable bits that may be fed to it and give access to read from the memory for another system that will read the information and use it for its needs. This system is “The mediator” we mentioned earlier.
3. Batcher sorting network - Bitonic sorting network, this system has its own clock which is different from the first system we mentioned. The system reads the information from the queue in the form of bits as input and returns it as a sorted output as we saw in this year's lectures. This system is “The consumer” we mentioned earlier.

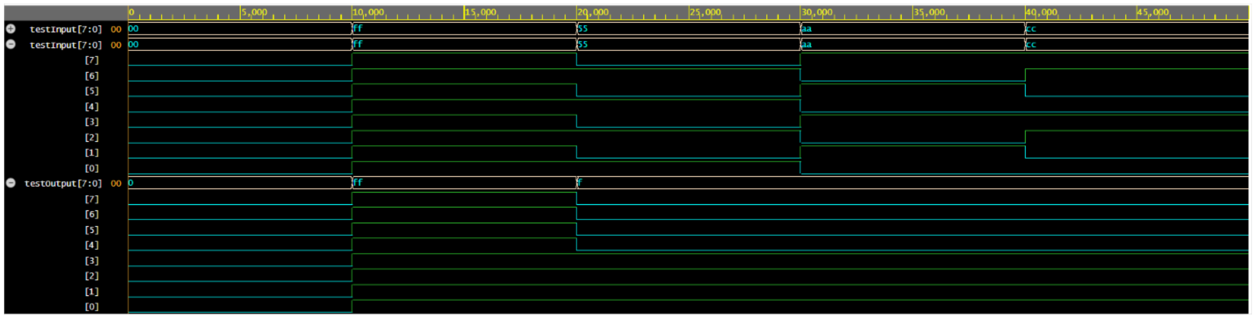

Fig 5 - Screenshot taken while simulating the sort operation.

To simulate Consumer for the system we decided to use a pipelined version of the batcher sorter from the lectures. The stages are divided to 6 concurrent calculations of minimum and maximum as presented in Fig 6.

The sorter sorted the inputs correctly even if they were metastable (metastability containing), this is allowed because of the monotonic functionality of the max and min modules which operate by simple OR and AND gates.

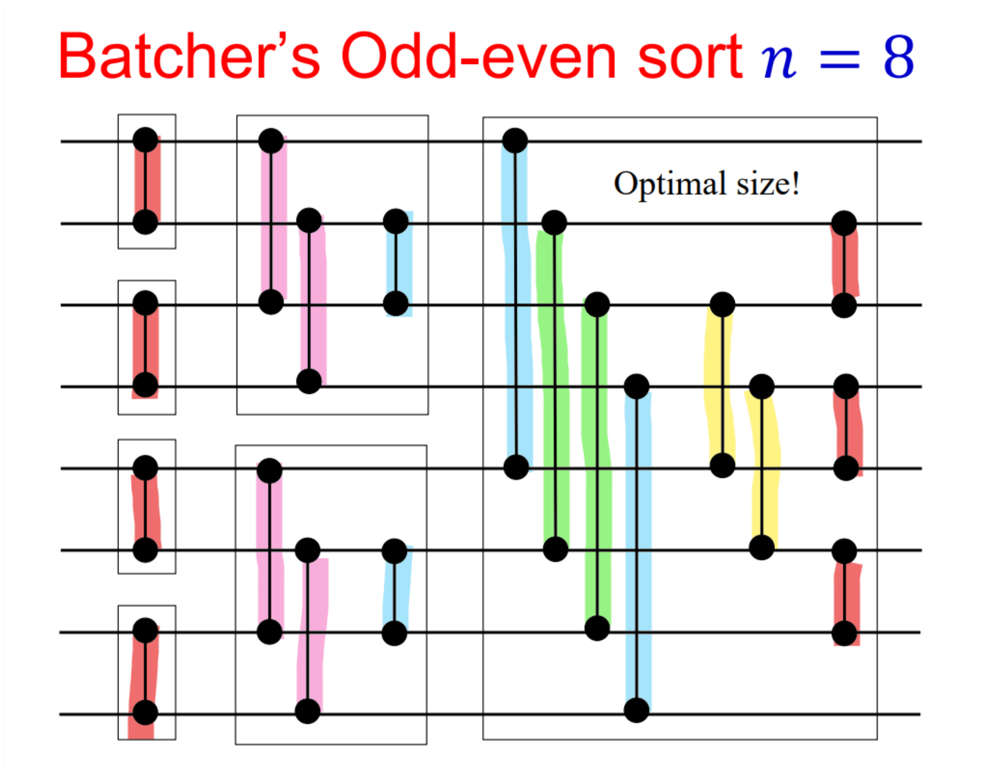

Fig 6 – diagram of the batcher's odd-even sort network and how we separated it to stages.

Before we proceed to the experiment itself, it is important to emphasize that the consumer and the manufacturer can work directly with each other if they were to work on the same clock frequency. (Fig 7)

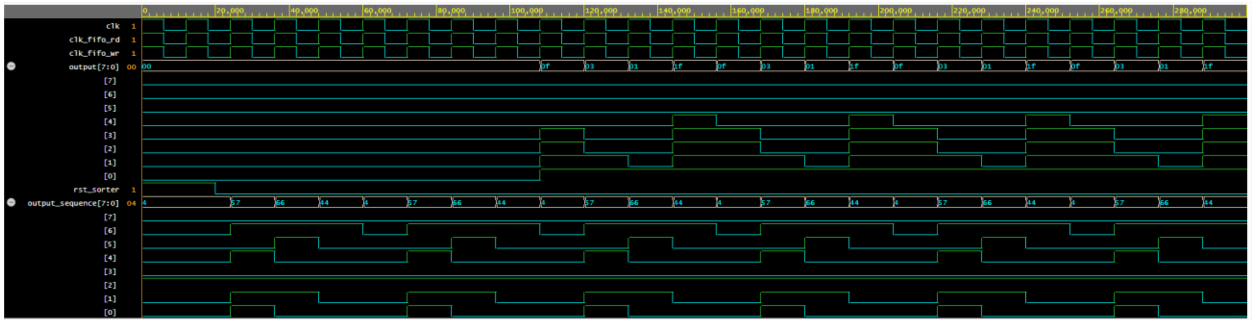

Fig 7 – The manufacturer and consumer at work when the clocks are at sync.

However, when we talk about two clocks with different clock cycles, we encounter synchronization problems that arise from CDC (clock domain crossing). Something that destroys the activity of the normal system that we expect that we expect to get (Fig 8)

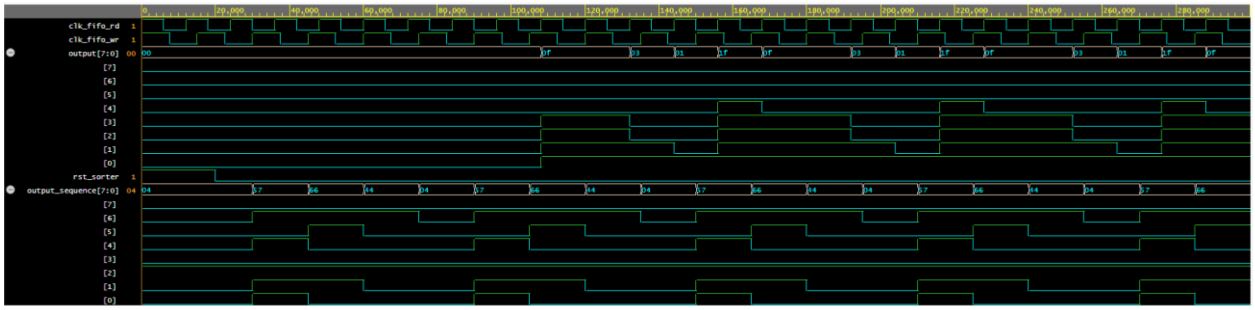

Fig 8 - The manufacturer and consumer at work when the clocks are not at sync.

The experiment we are going to show in this paper will show what happens when the read/write addresses of the FIFO contain metastable bits (which is a real threat to the functioning of the entire system) and how the Gray coding in the controllers solves these problems for us. That is, this paper will show how the gray code affects the activity of the controllers and thus also the activity of the entire system. (Refer to Fig 2)

Below is a block diagram of the system in question that we built

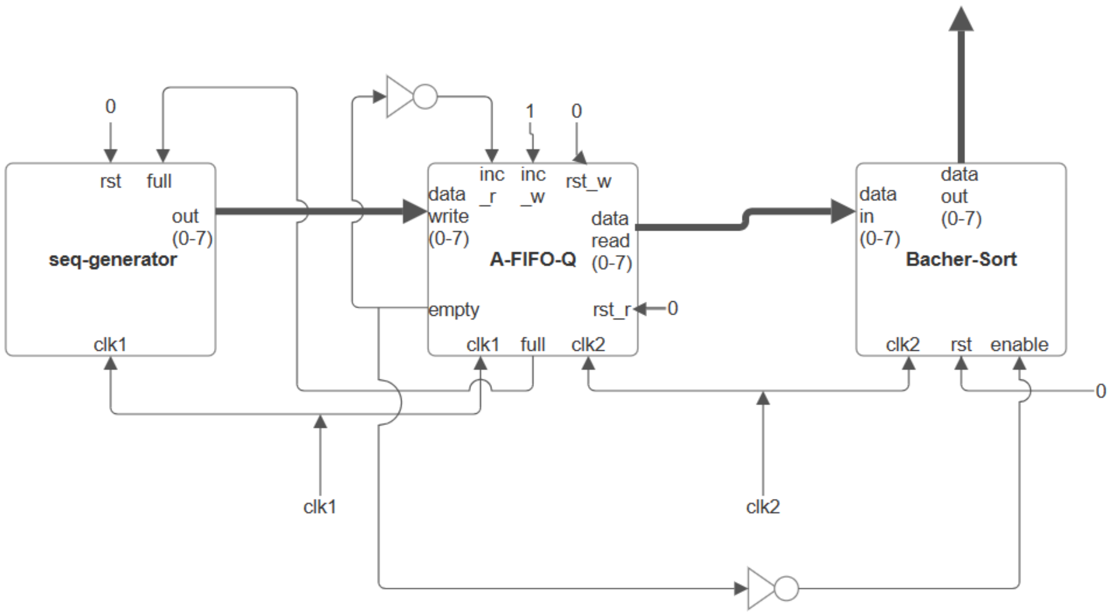

Fig 9 – Block diagram of the system that we built.

During this paper, we will see the activity of the system when the generator produces for us outputs that contain metastability and outputs that do not contain metastability.

<u>Testing of the system – without metastability inputs from the generator</u>

In Fig 10, you can see how we expect our system to behave in its standard way (when the controllers work with gray coding properly)

The system takes the output of our generator, and after a couple of clock cycles, it feeds the output from the FIFO memory into our consumer, which gives us bitonically sorted output at the end.

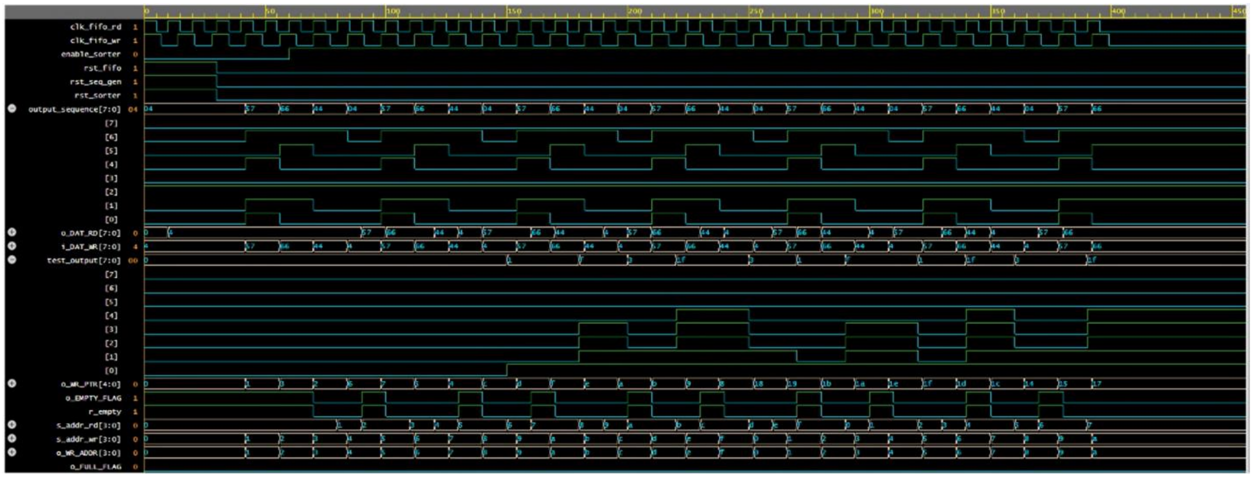

Fig 10 – The system at work with the gray coding as it should.

Now to illustrate what would happen when the system does not work with gray code momentarily - this is to illustrate how an untreated metastable error can destroy the entire function of the system - We simulate such a case by, in some clock cycle, directly inserting a metastable bit into the calculation of the next address in a way that bypasses the treatment that Gray Code provides inside the controllers.

We will see the result of this operation for our 2 controllers - READ and WRITE controllers.

For the READ controller, it can be seen that there is a direct effect on the "empty" flag that comes out as an output from the FIFO and the “rptr” (Fig 11)

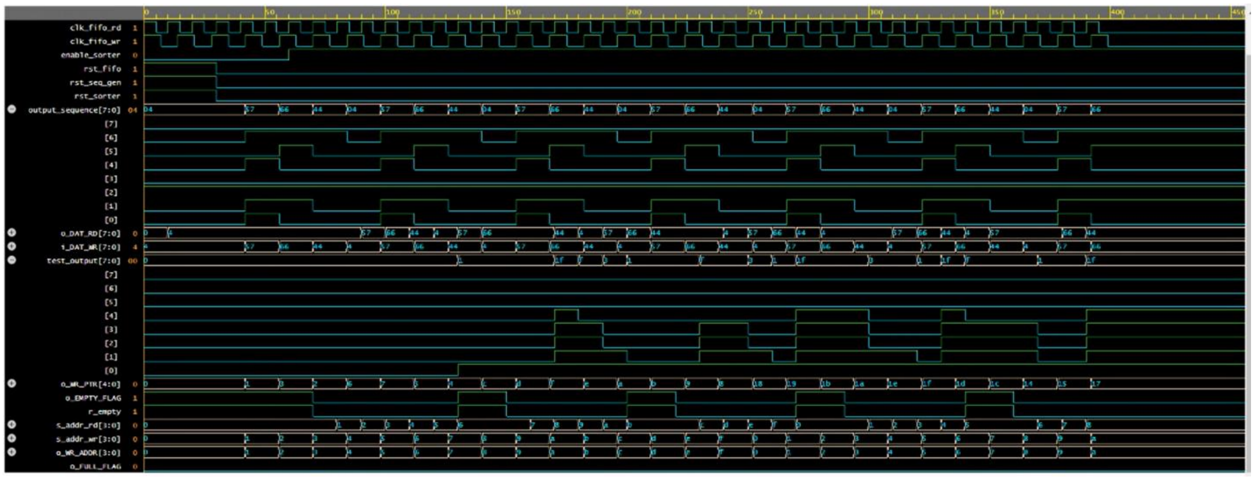

Fig 11 – Bypassing the gray code in a single cycle inside the READ controller.

It can indeed be seen in Fig. 11 that the "Empty" flag is greatly affected by this even when the metastability was for a single clock cycle. as a result - the function of the “empty” flag output goes completely wrong. This also directly affects the output we receive from our consumer (which is completely different from what we expected get under normal functioning).

For the WRITE controller, it can be seen that there is a direct effect on the “wptr” that comes out as an output from the controller. (Fig 12)

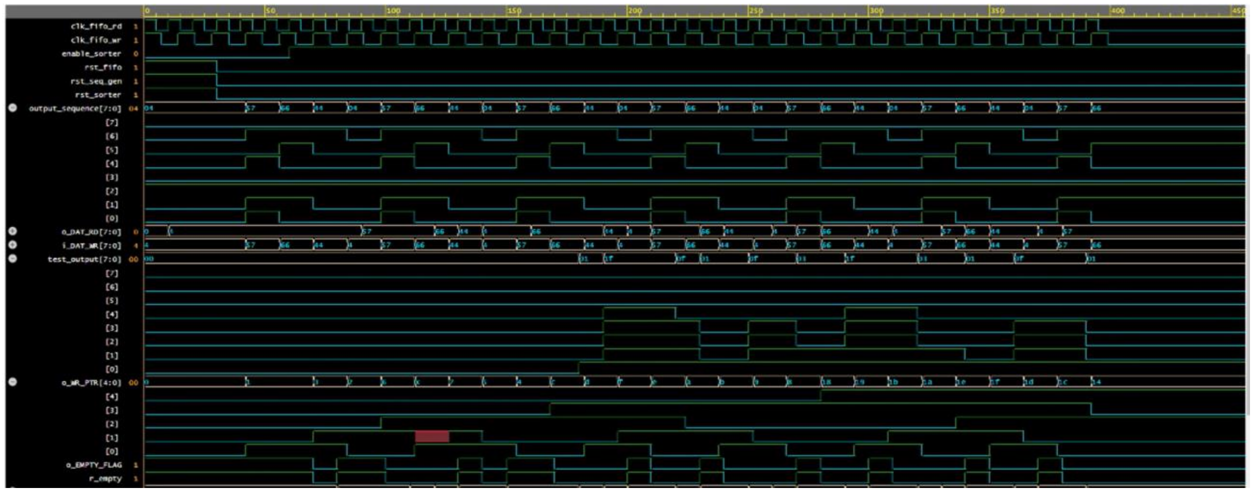

Fig 12 – Bypassing the gray code in a single cycle inside the WRITE controller.

From Fig 12 it is already trivial to see that a change in the wptr calculation creates a serious catastrophe for us in the output that was supposed to be received from our consumer.

Thus, we saw how sensitive the system is to metastable bits and the importance of the Gray code within the controllers.

Testing of the system – with metastability inputs from the generator

As before, we will do the exact same thing, only now our generator also brings us metastable bits as output.

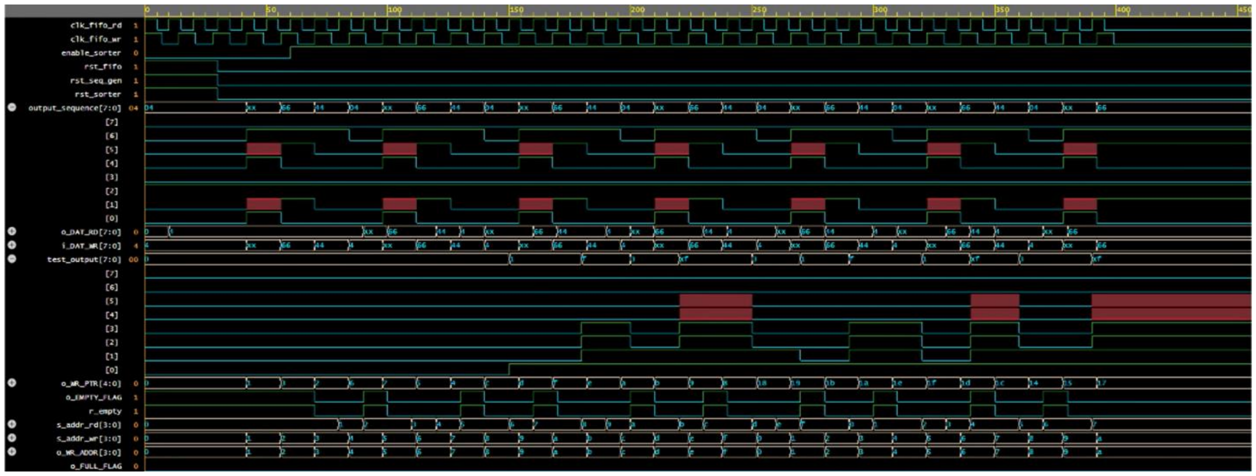

Fig 13 - The system at work with the gray coding as it should. (with metastability inputs)

For the READ controller, it can be seen that there is a direct effect on the "empty" flag that comes out as an output from the FIFO and the “rptr” (Fig 14)

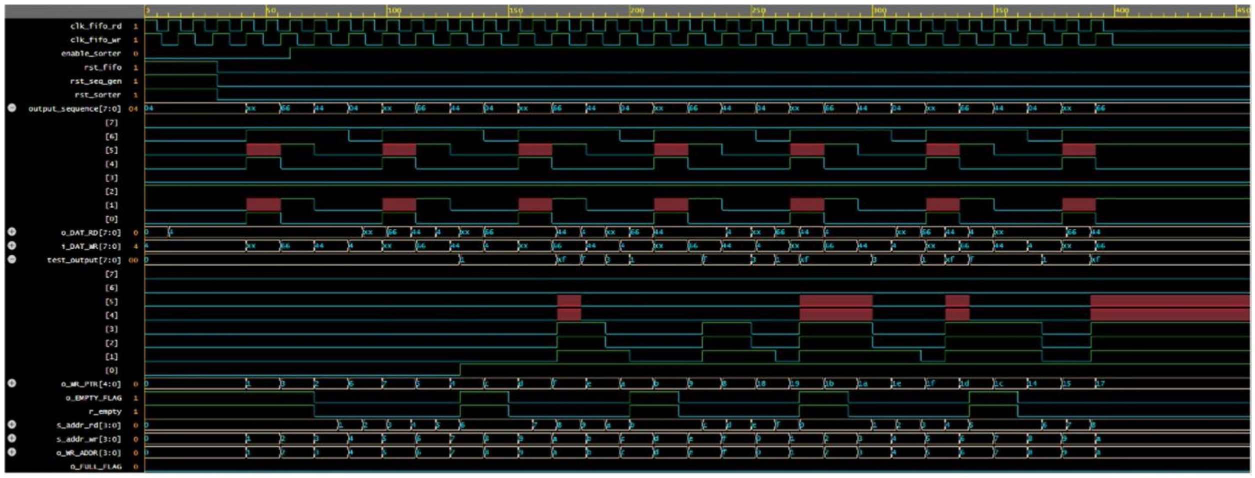

Fig 14 – Bypassing the gray code in a single cycle inside the READ controller. (with metastability inputs)

For the WRITE controller, it can be seen that there is a direct effect on the “wptr” that comes out as an output from the controller. (Fig 15)

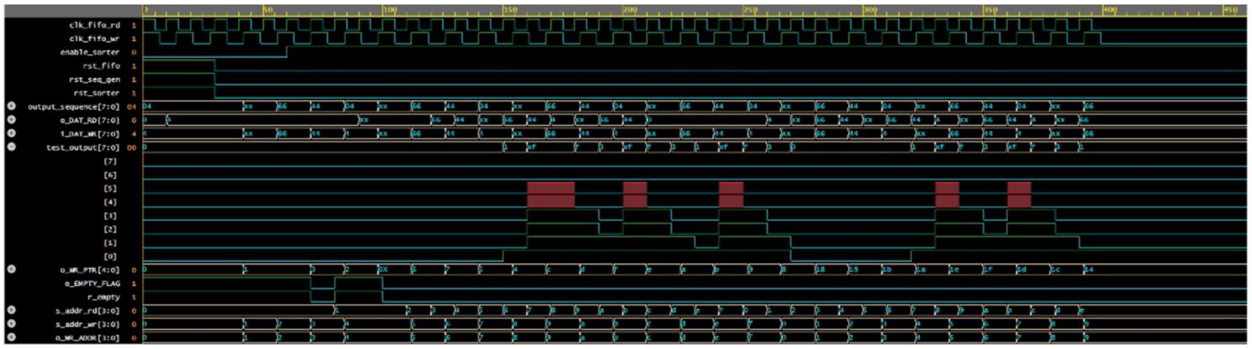

Fig 15 – Bypassing the gray code in a single cycle inside the WRITE controller. (with metastability inputs)

Conclusion

In this paper we dealt with Asynchronous FIFO Queues and showed that the system in question is metastability containing and also the importance of Gray coding in the calculation of the addresses for the pointers in the controllers.

It could be seen that if our controllers did not use Gray coding in the calculation of the memory addresses for the pointers, we would get a system that is completely destroyed given at least one metastable bit in the controllers.

In this way, we were able to show that indeed the system in question maintains inclusion and durability given metastability within it.

We remind you that all the code files we worked with were written in VHDL and can be viewed in the following link

Bibliography

- “Simulation and Synthesis Techniques for Asynchronous FIFO Design” by Clifford E. Cummings. Published by Sunburst Design, Inc.
- Github source code for this paper