Verification of Asynchronous FIFO using System Verilog

Amit Kumar
School of Engineering and Technology,
ITM University, Gurgaon, India

Shankar
School of Engineering and Technology,
ITM University, Gurgaon, India

Neeraj Sharma
School of Engineering and Technology,
ITM University, Gurgaon, India

ABSTRACT

As the designs gets complex, the probability of occurrence of bugs increases. This necessitated the introduction of the verification phase for verifying the functionality of the IC and to detect the bugs at an early stage. In this paper, the Asynchronous FIFO design is verified using SystemVerilog. The design uses a grey code counter to address the memory and for the pointer.

Keywords

Asynchronous FIFO, Setup time, Hold time, Metastability, Verification

1. INTRODUCTION

FIFO (First In First Out) is a buffer that stores data in a way that data stored first comes out of the buffer first. Asynchronous FIFO are most widely used in the System on chip (SOC) designs for data buffering and flow control [7]. As the System on chip involves multiple IPs operating at different speeds. Generally, Asynchronous FIFO is used when the write operation is faster than the read operation. Therefore, they need to be synchronized. Otherwise, it may lead to the lead to the metastability conditions. This will affect the operation of the chip. To overcome this problem Asynchronous FIFOs are used.

The Asynchronous FIFO is a First-In-First-Out memory queue with control logic that performs management of the read and write pointers, generation of status flags, and optional handshake signals for interfacing. .

FIFO architectures inherently have a challenge of synchronizing itself with the pointer logic of other clock domain and control the read and write operation of FIFO memory locations safely with the user logic. Data is written into the FIFO by write clock domain and data is read from the FIFO by read clock domain where the two clock domains are asynchronous to each other[5].

2. PROBLEM IN MULTICLOCK DOMAIN

It is problematic to synchronize multiple changing signals from one clock domain into a new clock domain and insuring that all the signals are synchronized to the same clock cycle in the new clock domain[3]. Multiple clock domain design are difficult to implement as compared to single clock designs. This is because there is single clock, in the single clock design that goes through the entire design. The problem faced in the multiple clock domain designs are Metastability, Setup & Hold time violations.

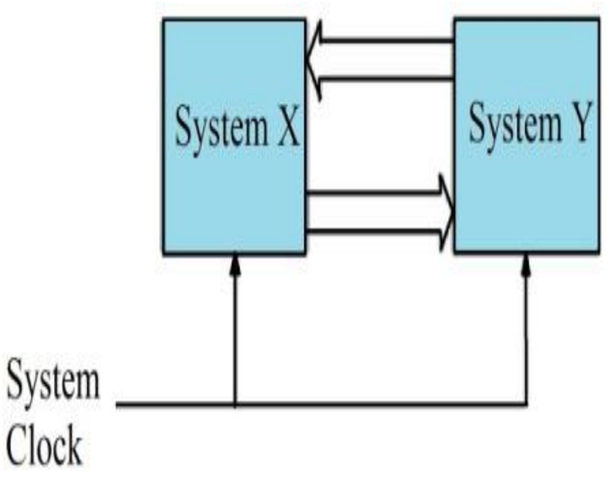


Figure 1: Single Clock Domain [1]

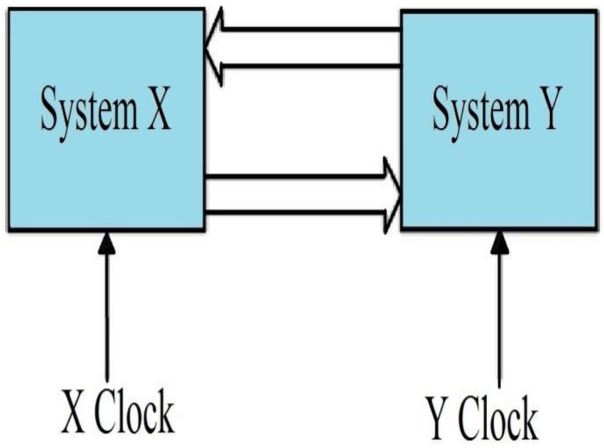


Figure 2: Multiple Clock Domain [1]

Setup time is the minimum amount of time required for which the data input should remain stable prior to the arrival of clock pulse so that the data are reliably sampled by the clock. Hold time is the minimum amount of time for which the data input should remain stable after the arrival of clock pulse so that the data is reliably sampled[1].

Metastability occurs in the multi clock domain when the output from one clock domain changes at the rising edge of the second clock domain which may lead to wrong results.

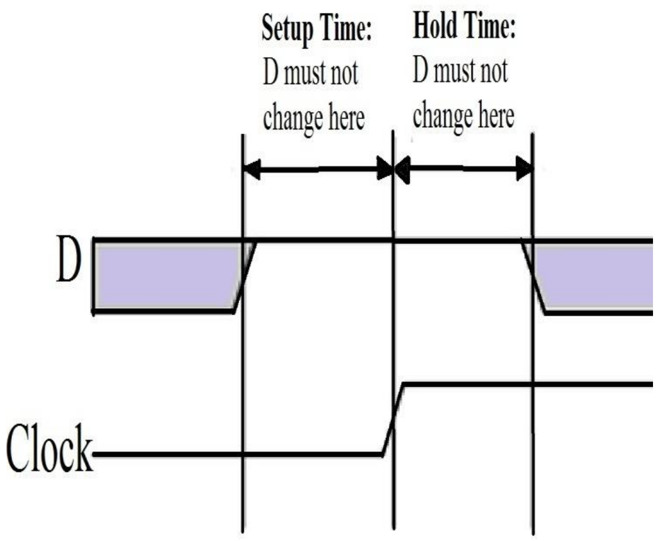


Figure 3: Setup and Hold Time Violations [1]

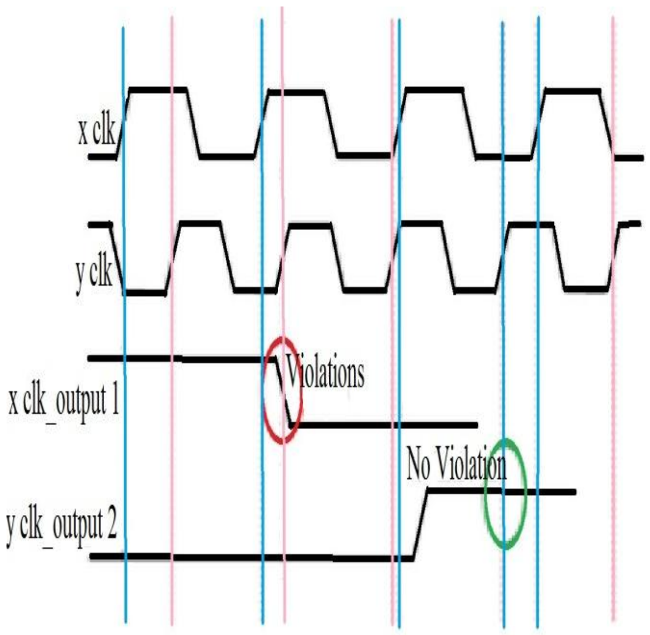


Figure 4: Setup and Hold Time Violations [1]

3. DESIGN

The asynchronous module consists of the following modules:

3.1 Fifo1

The module instantiates all the other modules used in the complete asynchronous fifo design. Therefore, it acts as a wrapper module to include all instances of the other module and the interface port of each module[2].

3.2 Fifomem

This module is instantiated synchronous dual port RAM that is accessed by both the write and read clock domains[2].

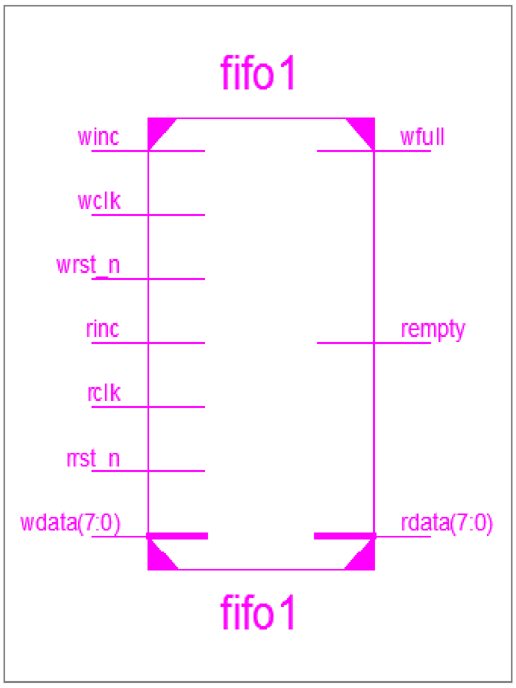


Figure 5: Fifo1 Top level design

3.3 Sync_r2w

This module consists of only flip flops to synchronize the read pointer into the write clock domain. This synchronized read pointer is used by the wptr_full module to generate the full condition when the fifomem is full and to overcome the metastability problem[3].

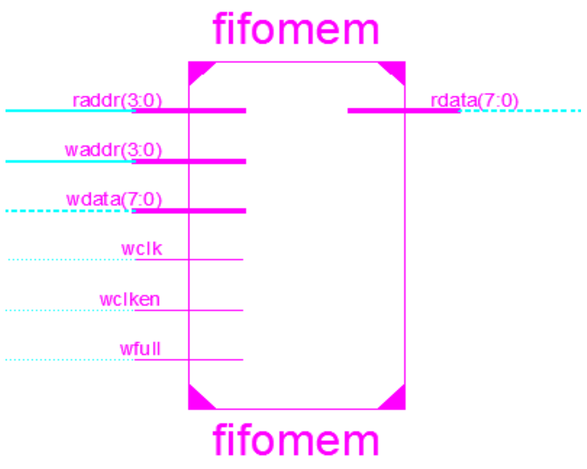


Figure 6: Fifomem

3.4 Sync_w2r

The sync_w2r module consists of two flip flops to overcome the metastabilty condition and to synchronize write pointer into the read clock domain which is used by the rptr module to generate the empty condition when there is no data in the fifomem block[2].

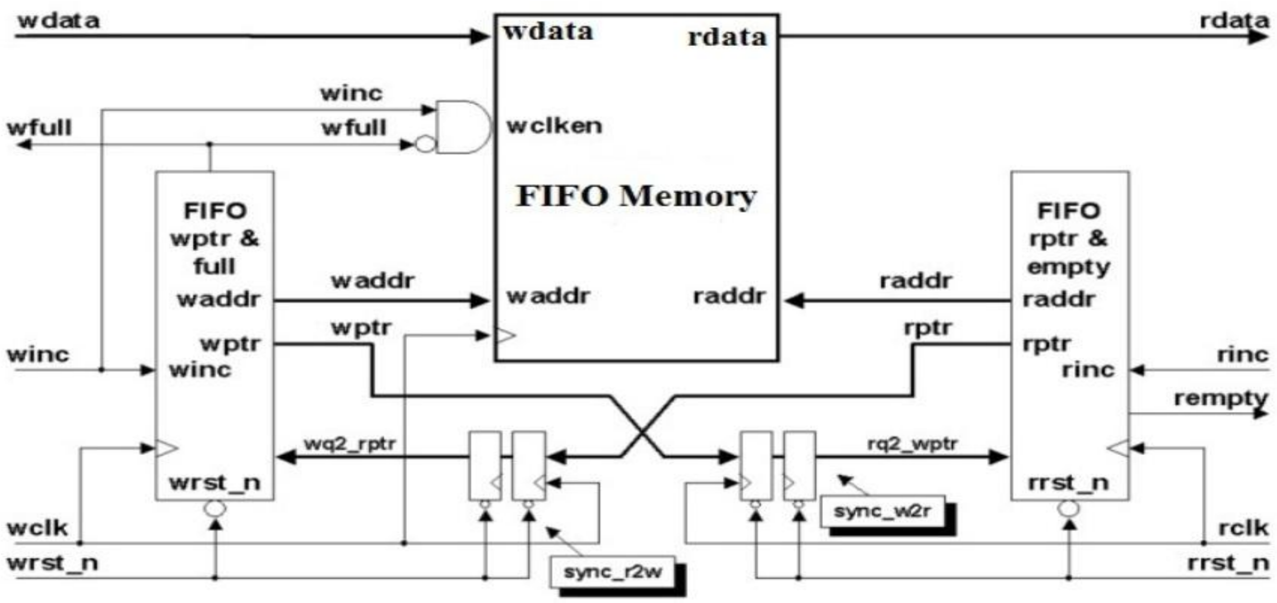


Figure 7: Asynchronous FIFO block diagram [2]

3.5 Rptr_empty

This module contains read pointer and empty flag. It is synchronized to the read clock domain. The module uses grey code counter to generate pointer and raddr[2].

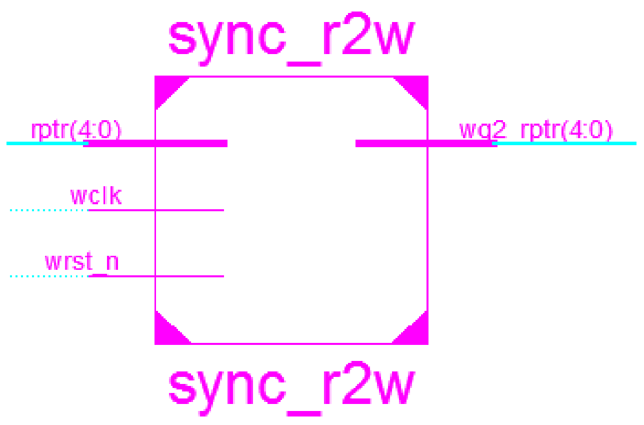


Figure 8: Sync_r2w

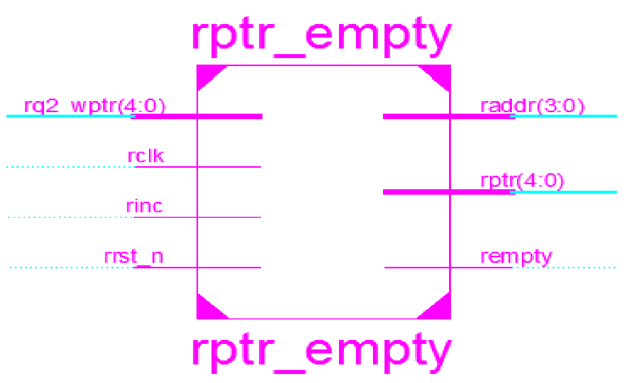


Figure 9: Rptr_empty

3.6 Wptr_full

This module contains the write pointer and full flag and it is synchronized to the write clock domain. The module uses grey code counter to generate the pointer and waddr[2].

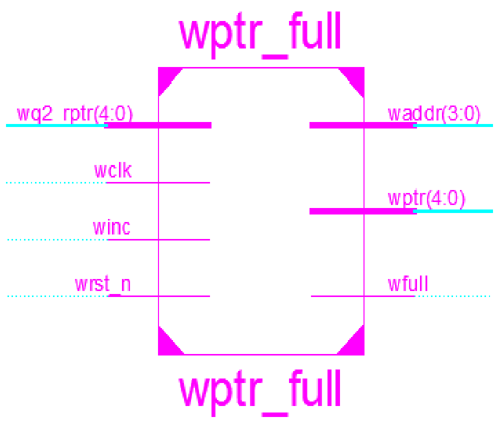


Figure 10: Wptr_full

Write and Read clock domain have different reset signals. These reset signals are intended to be asynchronously set and synchronously removed using the techniques described in Mills and Cummings[4].

4. VERIFICATION

The verification of the Asynchronous FIFO design is carried out to check that if the design is working as per the specification. The following modules are generated to check the functionality of the asynchronous fifo design.

4.1 Interface

The interface consists of bundle of wires i.e. multiple signals used to connect the Testbench to the DUT. The modports used in the interface block are used to group the signals for an individual block and to specify the directions of the signals.

The interface block used in the verification of asynchronous FIFO consists of two interfaces one synchronized to the write clock domain and other synchronized to the read clock domain.

4.2 Testcase
The testcase module will instantiate the environment module and calls the methods in the environment.

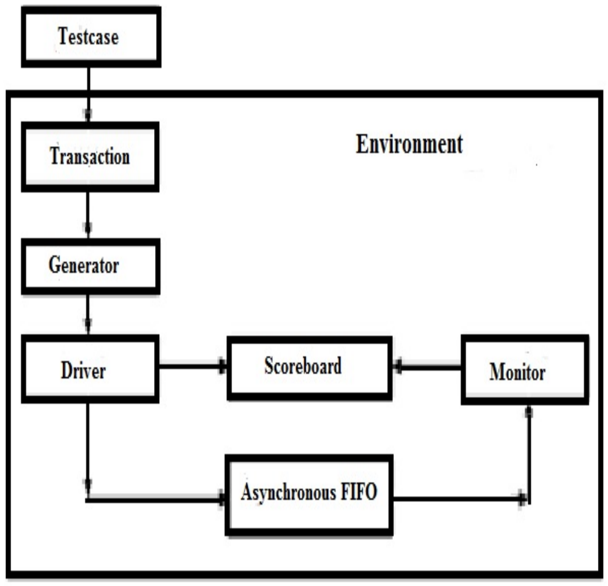


Figure 11: Testbench [6]

4.3 Transaction
This block randomizes the data values “wdata” to be given to the DUT and also assigns values to all the control bits that controls the read and write operation.

4.4 Generator
The generator block creates a mailbox mbx. The mbx mailbox is used to send the generated transaction to the driver block. The generator put the transaction tr into the mailbox mbx which is later retrieved by the driver block.

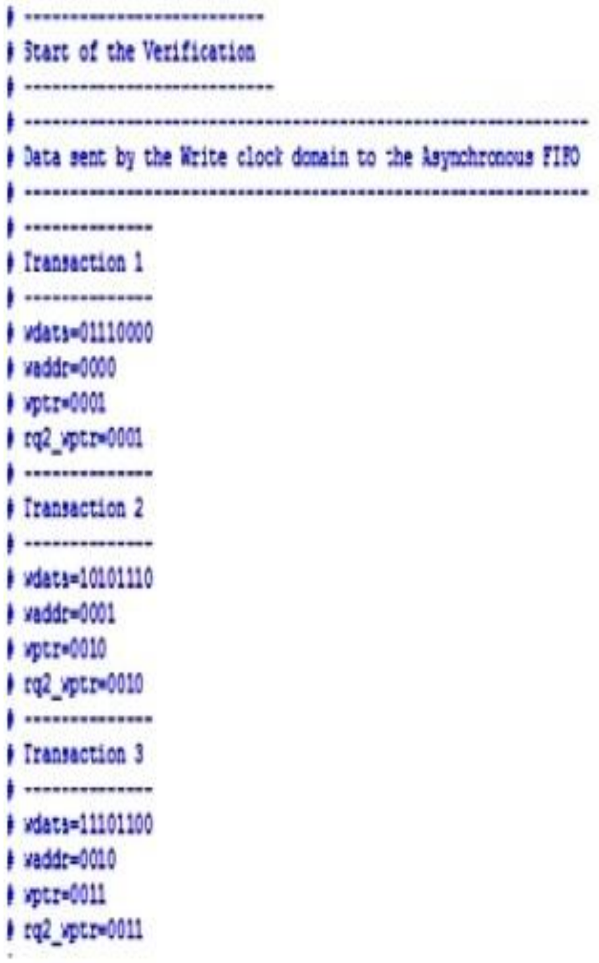
4.5 Driver
The driver block receives the transactions from the mailbox mbx and assigns the values in the transaction to the individual signals of the DUT through virtual interfaces. The driver also sends the transaction to scoreboard using drv2sb mailbox.

4.6 Monitor
This is the receiver section that receives the data from the receiver side of the Asynchronous FIFO. The transaction is also sent to the scoreboard using mon2sb mailbox.

4.7 Scoreboard
The scoreboard receives the transactions from the driver through mailbox “drv2sb” and another transaction from the mailbox “mon2sb”. The two transactions are compared with each other. Since in case of Asynchronous FIFO the data sent by the write clock domain system to the DUT should be same as that of the data received by the read clock domain system of the DUT. Therefore, if the two transactions received by the scoreboard are the same, then the DUT is working correctly.

4.8 Environment
The environment block instantiates all the modules and mailboxes. It consists of the following modules:
Build: It instantiates the mailboxes and other testbench modules i.e. driver, monitor, scoreboard.
Reset: It is used to initialize all the signals at the time of initialization and set them to their initial values.
Start: This method is used to run all the task and functions in all the modules.
Wait for end: This method is used to wait for the completion of the last transaction.
Run: This task run all the methods in the environment module in the specified order.
Report: Its main function is to detect the errors in the design and report the errors.

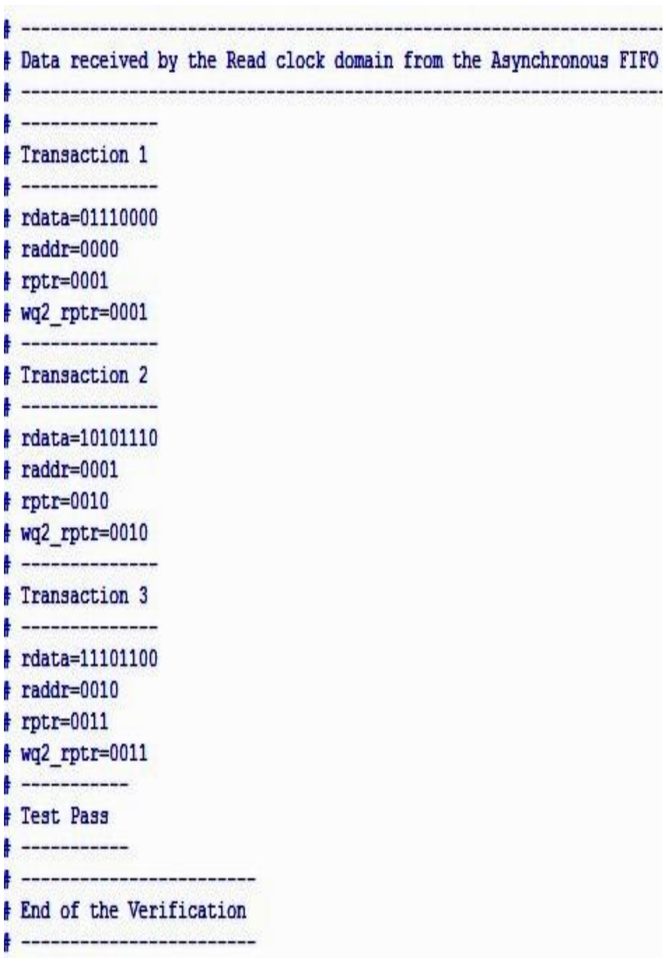
5. SIMULTION RESULTS



```
# -------------------------
# Start of the Verification
# --------------------------
# -----------------------------------------------------------
# Data sent by the Write clock domain to the Asynchronous FIFO
# -----------------------------------------------------------
# -------------
# Transaction 1
# -------------
# wdata=01110000
# waddr=0000
# wptr=0001
# rq2_wptr=0001
# -------------
# Transaction 2
# -------------
# wdata=10101110
# waddr=0001
# wptr=0010
# rq2_wptr=0010
# -------------
# Transaction 3
# -------------
# wdata=11101100
# waddr=0010
# wptr=0011
# rq2_wptr=0011
```

Figure 12: Data sent to the write clock domain

Each time the data “wdata” is sent to the asynchronous fifo the address “waddr” gets updated. At the same time the write pointer “wptr” and the read pointer synchronized to the write clock domain gets updated.



```
# ------------------------------------------------------------------
# Data received by the Read clock domain from the Asynchronous FIFO
# ------------------------------------------------------------------
# --------------
# Transaction 1
# --------------
# rdata=01110000
# raddr=0000
# rptr=0001
# wq2_rptr=0001
# --------------
# Transaction 2
# --------------
# rdata=10101110
# raddr=0001
# rptr=0010
# wq2_rptr=0010
# --------------
# Transaction 3
# --------------
# rdata=11101100
# raddr=0010
# rptr=0011
# wq2_rptr=0011
# -----------
# Test Pass
# -----------
# ------------------------
# End of the Verification
# ------------------------
```

Figure 13: Data received at the read clock domain

Each time the data “rdata” is read from the asynchronous fifo the address “raddr” gets updated. At the same time the read pointer “rptr” and the write pointer synchronized to the read clock domain gets updated.

6. CONCLUSION

Since the data sent by the write clock domain to the asynchronous fifo is same as the data received at the read clock domain from the asynchronous fifo. Therefore, the asynchronous fifo is functionally correct. As shown in Figure. 12, during “Transaction 1” when the wdata is sent by the write clock domain, the 8-bit wdata is stored at the memory location pointed to by the waddr. The wptr and rq2_wptr gets incremented to point to the next empty memory location in the fifo..

During next transaction “Transaction 2”, the next word wdata is stored at the next memory location pointed to by the waddr. and the pointers wptr and rq2_wptr gets incremented.

So, in this way data from the write clock domain are stored at the consecutive memory location present in the asynchronous fifo until the memory becomes full. In case the memory is full the full flag is generated to prevent the overflow condition.

As shown in Figure 13, the data stored at the memory location in asynchronous fifo is read by the read clock domain through 8-bit rdata bus. Since the design has the fifo implementation. Therefore, the data is read in the same way as it is written. Hence, the rdata at the first memory location pointed by the raddr is read first provided the memory is not empty. Otherwise, empty flag will be high. When the first data word is read by the read clock domain, the pointers rptr and wq2_rptr gets incremented to point to the next memory in the asynchronous fifo to be read. So, on completing the read operation of “Transaction 1”, the “Transaction 2” is read by the read clock domain in the same way.

Therefore, the read operation is performed on the consecutive memory locations of the asynchronous fifo by the read clock domain until the asynchronous fifo becomes empty.

This asynchronous fifo design can be used in the future to overcome the timing issues which occurs in the multiclock domain systems.

7. REFERENCES

[1] Mohit Arora, “*The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits*,” Springer, 2011, ch 3, sec 3.3, pp 54-55

[2] Clifford E. Cummings, “Simulation and Synthesis Techniques for Asynchronous FIFO Design,” *SNUG 2000 Users Group Conference, San Jose, CA, 2002) User Papers*, March 2002.

[3] 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

[4] Clifford E. Cummings and Don Mills, “Synchronous Resets? Asynchronous Resets? I am So Confused! How Will I Ever Know Which to Use?” *SNUG 2002 (Synopsys Users Group Conference, San Jose, CA, 2002) User Papers*, March 2002.

[5] Dadhania Prashant C. “Designing Asynchronous FIFO,” *Journal Of Information, Knowledge and Reseaarch In Electronics and communication Engineering*, Vol.2, Issue.2, November 2013

[6]Chris Spears, “System Verilog for Design, “*A Guide to Using System Verilog for Hardware Design and Modeling*,” Springer Second edition.

[7] Mu-Tien Chang, Po-Tsang Huang, and Wei Hwang,”A Robust Ultra-Low Power Asynchronous FIFO Memory with Self-Adaptive Power Control,*” SOC Conference, 2008 IEEE International*, pp.175-178, Sept., 2008