International Journal for Modern Trends in Science and Technology, 9(06): 174-183, 2023

ISSN: 2455-3778 online
DOI: https://doi.org/10.46501/IJMTST0906026
Available online at: http://www.ijmtst.com/vol9issue06.html

Verification Of Asynchronous FIFO using System Verilog

S. Girish Gandhi | Md. Mukthadeer Ahamed | R. Raman Sravan Kumar | V. Sunil | P. Ambica Sravan Kumar

Department of Electronics and communications, Narayana engineering college, Nellore, Andhra Pradesh, India.

To Cite this Article
S. Girish Gandhi, Md. Mukthadeer Ahamed, R. Raman Sravan Kumar, V. Sunil and P. Ambica Sravan Kumar. Verification Of Asynchronous FIFO using System Verilog. International Journal for Modern Trends in Science and Technology 2023, 9(06), pp. 174-183. https://doi.org/10.46501/IJMTST0906026

Article Info
Received: 21 May 2023; Accepted: 12 June 2023; Published: 18 June 2023.

ABSTRACT

A FIFO is a "First In First Out" memory queue between any FIFO asynchronous domains with simultaneous write and read access to and from the FIFO, these accesses being on different clocks. The FIFO has input ports like data input (write), write clock, read clock, reset and output ports like FIFO full flag, data out (read) and FIFO empty flag. It also has control signals like write enable and read enable. The most important signals that control the FIFO operation are the write pointer and the read pointer. These pointers in the case of Synchronous FIFO operate in a single clock while in the case of Asynchronous FIFO operate in two clocks, write clock and read clock respectively. FIFO can be either Synchronous or Asynchronous. The basic difference between them is that the entire operation of Synchronous FIFO is entirely dependent on the clock whereas the write operation and read operation of Asynchronous FIFO are asynchronous to each other. In this project a Novel approach to designing an Asynchronous FIFO is used. Instead of taking a separate bit to identify whether the FIFO is full or empty, it is used to identify if the FIFO is full or empty. 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 project, the Asynchronous FIFO design is verified by using System Verilog. 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].

1.1 Synchronous FIFO

Synchronous FIFO are the ideal choice for high-performance systems due to high operating speed. As shown in Fig. 1 Synchronous FIFOs also offer many other advantages that improve system performance and reduce complexity. These include status flags: synchronous flags, half-full, programmable almost-empty and almost-full flags. Synchronous FIFOs are easier to use at high speeds because they use free-running clocks to time internal operations.

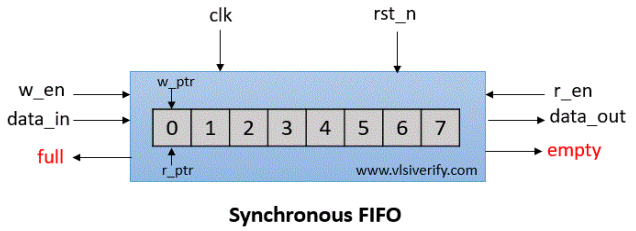

Fig 1: Synchronous FIFO

1.2 Asynchronous FIFO

Asynchronous FIFO refers to a FIFO where the data values are written to the FIFO at a different rate and data values are read from the same FIFO at a different rate, both at the same time. The reason for calling it Asynchronous FIFO as shown in Fig. 2, is that the read and write clocks are not Synchronized.

The basic need for an Asynchronous FIFO arises when we are dealing with systems with different data rates. For the rate of data flow being different, we will be needing Asynchronous FIFO to synchronize the data flow between the systems. The main work of an Asynchronous FIFO is to pass data from one clock domain to another clock domain.

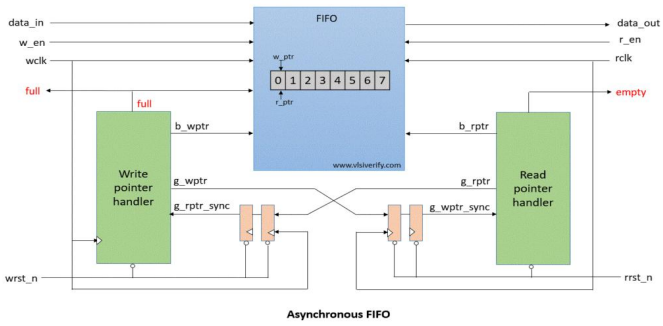

Fig 2: Asynchronous FIFO

2. LITERATURE REVIEW

2.1 Clock Domain Crossing

FIFO 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]. In digital electronic design a Clock Domain Crossing (CDC), or simply clock crossing, is the traversal of a signal in a synchronous digital circuit from one clock domain into another. If a signal does not assert long enough and is not registered, it may appear asynchronous on the incoming clock boundary. A synchronous system is composed of a single electronic oscillator that generates a clock signal, and its clock domain the memory elements directly clocked by that signal from that oscillator, and the combinational logic attached to the outputs of those memory elements. Because of speed-of-light delays, timing skew, etc., the size of a clock domain in such a synchronous system is inversely proportional to the frequency of the clock.

As shown in Fig. 3, A few modern CPUs have such a High-Speed clock, that designers are forced to create several different clock domains on a single CPU chip.

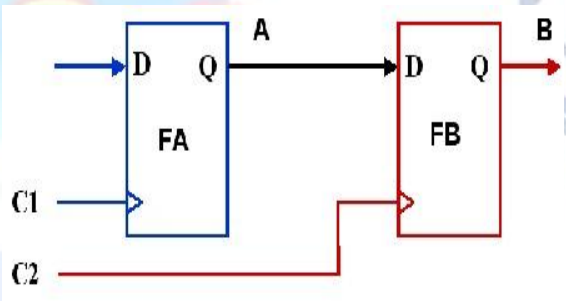

Fig 3: Clock Domain Crossing

A. Metastability

Metastability is one of the major defects. A flip-flop has metastability issues if the clock and data change very closely in time, causing the output to be at an unknown logic value for an unbounded period of time. While metastability cannot be eliminated, it is usually tolerated by adding a multi-flop synchronizer to control asynchronous boundaries and using those synchronizers to block the destination of an asynchronous boundary when its source is changing. FIFOs, 2-phase and 4-phase

handshakes are typical structures used for this type of synchronization.

Glitches on asynchronous boundaries can also cause defects, since a glitch on an asynchronous crossing can trigger the capture of an incorrect signal transition. Data coherency issues occur in a design when multiple synchronizers settle to their new values in different cycles and subsequently interact in downstream logic. The list goes on. While the concepts and methodologies for verification of such issues have been extensively researched in the past ten years, practical solutions have been offered primarily at the IP-level. Little work has been attempted to tackle clock domain crossing (CDC) verification signoff of large system-on-chip (SoC) designs.

B. Data Loss

Whenever a new source data is generated, the destination domain may not capture it in the very first cycle of the destination clock because of metastability. As long as each transition on the source signal is captured in the destination domain, data is not lost. In order to ensure this, the source data should remain stable for some minimum time, so that the setup and hold time requirements are met with respect to at least one active edge of destination clock.

If the active clock edges of C1 and C2 arrive close together, the first clock edge of C2, which comes after the transition on source data A, is not able to capture it. The second edge of clock C2 finally captures the data. However, if there is sufficient time between the transition on data A and the active edge of clock C2, the data is captured in the destination domain in the first cycle of C2. Hence, there may not be a cycle - by - cycle correspondence between the source and destination domain data. Whatever the case, it is important that each transition on the source data should get captured in the destination domain. For example: Assume that the source clock C1 is twice as fast as the destination clock C2 and there is no phase difference between the two clocks. Further assume that the input data sequence "A" generated on the positive edge of clock C1 is "00110011". The data B captured on the positive edge of clock C2 will be "0101". Here, since all the transitions on signal A are captured by B, the data is not lost. However, if the input sequence is "00101111", then the output in the destination domain will be "0011". Here the third data value in the input sequence which is "1". In order to prevent data loss, the data should be held constant in the source domain long enough to be properly captured in the destination domain.

2.2. Problem In Multi Clock Domain

It is problematic to synchronize multiple changing signals from one clock domain into a new clock domain and assuring that all the signals are synchronized to the same clock cycle in the new clock domain.[3]

Multiple clock domain designs 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.

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]

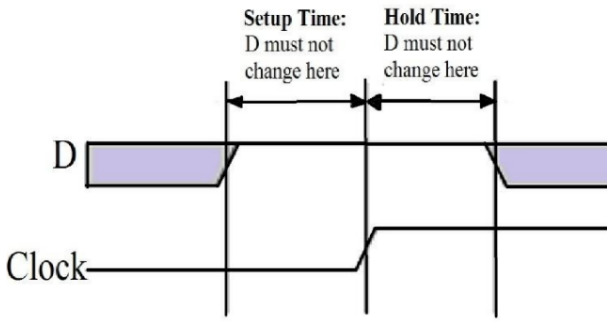

Fig 4: Setup and Hold Time Violations

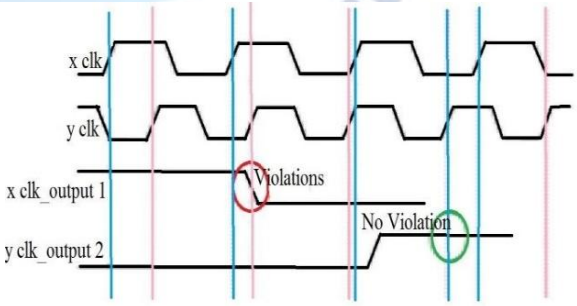

Fig 5: Setup and Hold Time Violations

A. Single Clock Domain

A clock domain is a section of the design that is driven by one or more clocks that are coupled to one another. A clock with a frequency of 10MHz is handled as a single

clock domain design, as a half clock is powered by a 10MHz clock. As Shown in below figure 6,

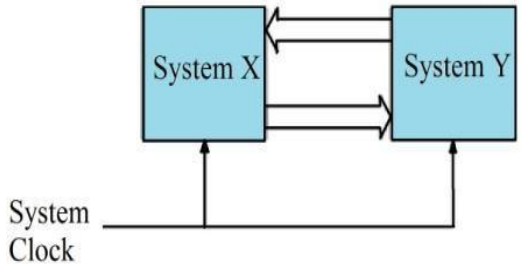

Fig 6: Single Clock Domain

B. Multiple Clock Domain

In the practical ASIC and SOC designs the multiple clocks are used and the designs are called as multiple clock domain designs. These kinds of designs need to be described using the efficient design architectures and Verilog RTL. This chapter focuses in the key design techniques which are used to describe the multiple clock domain designs while passing data from one of the Clock Domain to other. The chapter key highlights are the detail description for the synchronizers, data path, and control path synchronization logic using the efficient Verilog RTL. This chapter also discusses on the key design challenges in the multiple clock domain designs and even this chapter focuses on the design guidelines to describe the efficient clock domain designs.

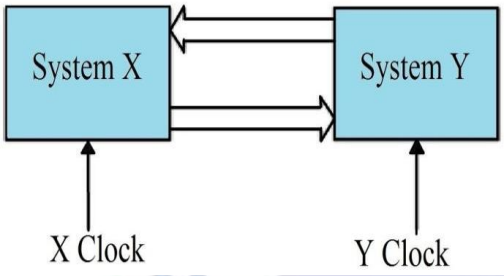

Fig 7: Multiple Clock Domain

2.3 Block Diagram

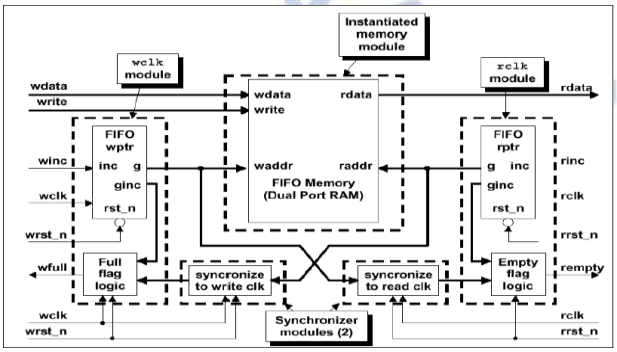

Fig 8: Block Diagram of Asynchronous FIFO

2.4 Flow Chart

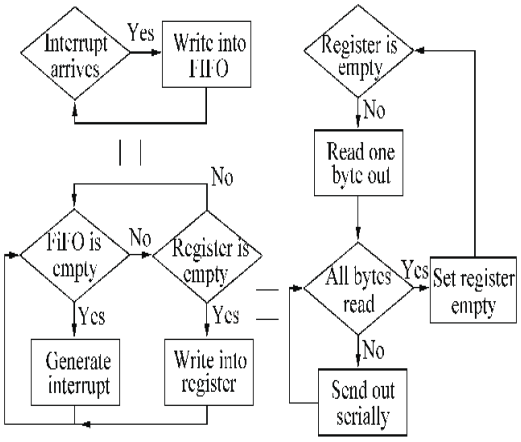

Fig 9: Flow Chart

3. FIFO POINTERS AND MODULES DESIGN

3.1 FIFO Pointers

FIFOs are commonly used in electronic circuits for buffering and flow control between hardware and software. In its hardware form, a FIFO primarily consists of a set of read and write pointers, storage and control logic. There are two FIFO pointers were are,

- Synchronous FIFO Pointers
- Asynchronous FIFO Pointers

A. Synchronous FIFO Pointers

For synchronous FIFO design (a FIFO where writes to, and reads from the FIFO buffer are conducted in the same clock domain), one implementation counts the number of writes to, and reads from the FIFO buffer to increment (on FIFO write but no read), decrement (on FIFO read but no write) or hold (no writes and reads, or simultaneous write and read operation) the current fill value of the FIFO buffer. The FIFO is full when the FIFO counter reaches a predetermined full value and the FIFO is empty when the FIFO counter is zero.

Unfortunately, for asynchronous FIFO design, the increment-decrement FIFO fill counter cannot be used, because two different and asynchronous clocks would be required to control the counter. To determine full and empty status for an asynchronous FIFO design, the write and read pointers will have to be compared.

B. Asynchronous FIFO Pointers

In order to understand FIFO design, one needs to understand how the FIFO pointers work. The write

pointer always points to the next word to be written; therefore, on reset, both pointers are set to zero, which also happens to be the next FIFO word location to be written. On a FIFO-write operation, the memory location that is pointed to by the write pointer is written, and then the write pointer is incremented to point to the next location to be written. Similarly, the read pointer always points to the current FIFO word to be read. Again on reset, both pointers are reset to zero, the FIFO is empty and the read pointer is pointing to invalid data (because the FIFO is empty and the empty flag is asserted).

As soon as the first data word is written to the FIFO, the write pointer increments, the empty flag is cleared, and the read pointer that is still addressing the contents of the first FIFO memory word, immediately drives that first valid word onto the FIFO data output port, to be read by the receiver logic. The fact that the read pointer is always pointing to the next FIFO word to be read means that the receiver logic does not have to use two clock periods to read the data word. If the receiver first had to increment the read pointer before reading a FIFO data word, the receiver would clock once to output the data word from the FIFO, and clock a second time to capture the data word into the receiver.

That would be needlessly inefficient. The FIFO is empty when the read and write pointers are both equal. This condition happens when both pointers are reset to zero during a reset operation, or when the read pointer catches up to the write pointer, having read the last word from the FIFO. A FIFO is full when the pointers are again equal, that is, when the write pointer has wrapped around and caught up to the read pointer.

The FIFO is either empty or full when the pointers are equal, but One design technique used to distinguish between full and empty is to add an extra bit to each pointer. When the write pointer increments past the final FIFO address, the write pointer will increment the unused MSB while setting the rest of the bits back to zero. If the MSBs of the two pointers are the same, it means that both pointers have wrapped the same number of times.

3.2. Asynchronous FIFO Pointers using Gray Code Counter

One Gray code counter style uses a single set of flip-flops as the Gray-code register with accompanying Gray-to binary conversion, binary increment, and binary-to-Gray- conversion. A second Gray code counter style, the one described in this paper, uses two sets of registers, one a binary counter and a second to capture a binary to-Gray converted value.

The intent of this Gray code counter is to utilize the binary carry structure, simplify the Gray-to-binary conversion; reduce combinational logic, and increase the upper frequency limit of the Gray code counter.

The binary counter conditionally increments the binary value, which is passed to both the inputs of the binary counter as the next-binary-count value, and is also passed to the simple binary-to-Gray conversion logic, consisting of one 2-input XOR gate per bit position. Fig. 10, shows the block diagram for an n-bit Gray-Code counter. This implementation requires twice the number of flip-flops, but reduces the combinatorial logic and can operate at a higher frequency.

In FPGA designs, availability of extra flip-flops is rarely a problem since FPGAs typically contain far more flip-flops than any design will ever use. In FPGA designs, reducing the amount of combinational logic frequently translates into significant improvements in speed

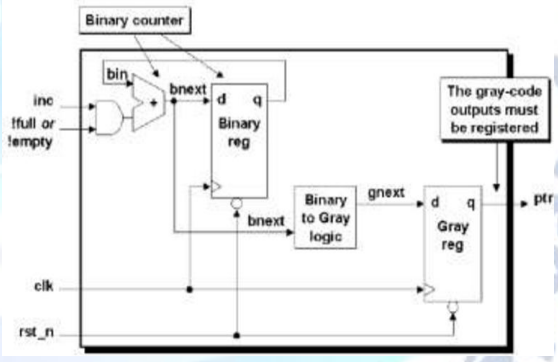

Fig 10: Gray Code Counter

3.3 Asynchronous FIFO Design

To Design a Asynchronous FIFO there are some of the modules which we are going to discuss,

- fifo1
- fifomem
- sync_r2w
- sync_w2r
- rptr_empty
- wptr_full

A. FIFO Top Level Model – fifo1

The top -level FIFO module (fifo1) is a parameterized FIFO design with all sub-blocks instantiated using the recommended practice of doing named port connections. Another common coding practice is to give the top-level module instantiations the same name as the module name. This is done to facilitate debug, since referencing module names in a hierarchical path will be straight forward if the instance names match the module names. As shown in below Figure 11,

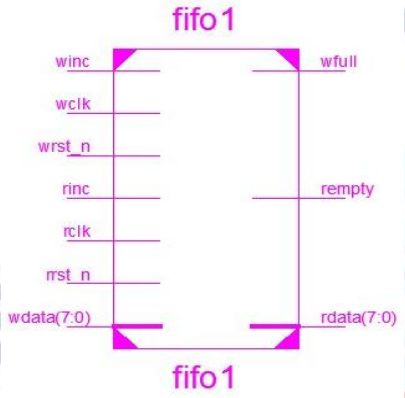

Fig 11: Schematic diagram of fifo1

B. FIFO Memory Buffer - fifomem

The FIFO memory buffer (fifomem) is typically an instantiated ASIC or FPGA dual-port, synchronous memory device. The memory buffer could also be synthesized to ASIC or FPGA registers using the RTL code in this module.

About an instantiated vendor RAM versus a Verilog-declared RAM, the Synopsys Design Ware team did internal analysis and found that for sizes up to 256 bits, there is no lost area or performance using the Verilog-declared RAM compared to an instantiated vendor RAM.

If a vendor RAM is instantiated, it is highly recommended that the instantiation be done using named port connections, As shown in below Figure 12,

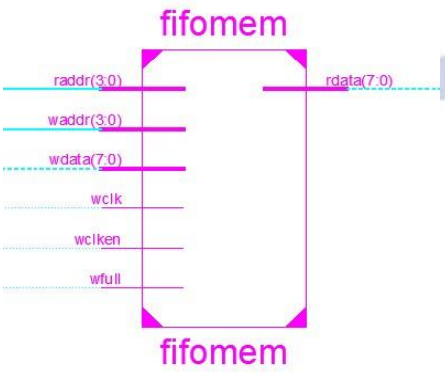

Fig 12: Schematic Diagram of fifomem

C. Read Domain To Write Domain Synchronizer – Sync_r2w

This is a simple synchronizer module (sync_w2r), used to pass an n-bit pointer from the write clock domain to the read clock domain, through a pair of registers that are clocked by the FIFO read clock. Notice the simplicity of the always block that concatenates the two registers together for reset and shifting. All module outputs are registered for simplified synthesis using time budgeting. All outputs of this module are entirely synchronous to the rclk and all asynchronous inputs to this module are from the wclk domain with all signals named using an "w" prefix, making it easy to set a false path on all "w*" signals for simplified static timing analysis. As shown in below Figure 13,

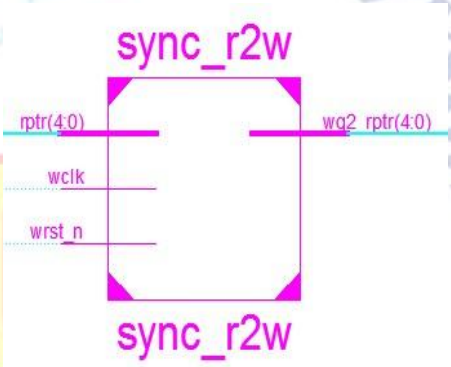

Fig 13: Schemati Diagram of Sync_r2w

D. Write Domain to Read Domain Synchronizer – Sync_w2r

This is a simple synchronizer module (sync_w2r), used to pass an n-bit pointer from the write clock domain to the read clock domain, through a pair of registers that are clocked by the FIFO read clock. Notice the simplicity of the always block that concatenates the two registers together for reset and shifting. All module outputs are registered for simplified synthesis using time budgeting. All outputs of this module are entirely synchronous to the rclk and all asynchronous inputs to this module are from the wclk domain with all signals named using an "w" prefix, making it easy to set a false path on all "w*" signals for simplified static timing analysis. As shown in below Figure 14,

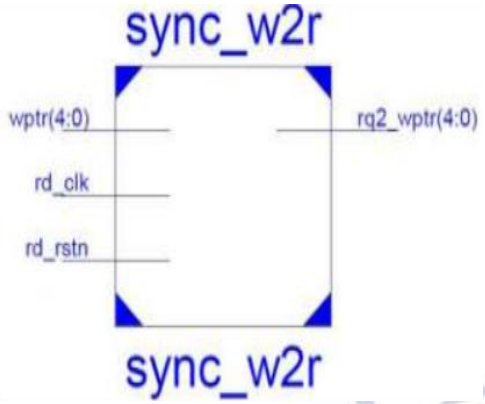

Fig 14: Schematic Diagram of sync_w2r

E. Read Pointer And Empty Generator Logic – rptr_empty

This module encloses all of the FIFO logic that is generated within the read clock domain (except synchronizers). The read pointer is a dual n-bit Gray code counter. The n-bit pointer (rptr) is passed to the write clock domain through the sync_r2w module. The (n-1)-bit pointer (raddr) is used to address the FIFO buffer.

The FIFO empty output is registered and is asserted on the next rising rclk edge when the next rptr value equals the synchronized wptr value. All module outputs are registered for simplified synthesis using time budgeting. This module is entirely synchronous to the rclk for simplified static timing analysis. As shown in below Figure 15,

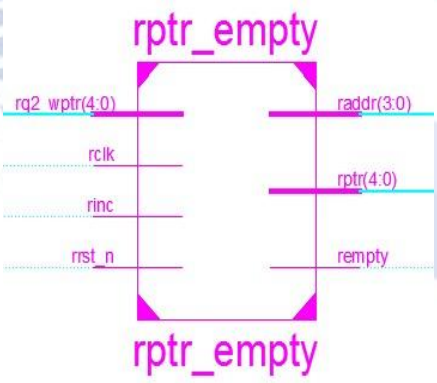

Fig 15: Schematic Diagram of rptr_empty

F. Writer Pointer and Full Genration Logic – wptr_full

This module encloses all of the FIFO logic that is generated within the write clock domain (except synchronizers). The write pointer is a dual n-bit Gray code counter. The n-bit pointer (wptr) [2]is passed to the read clock domain through the sync_w2r module.

The (n-1)-bit pointer (waddr) is used to address the FIFO buffer. The FIFO full output is registered and is asserted on the next rising wclk edge when the next modified wgnext value equals the synchronized and modified wrptr2 value (except MSBs). All module outputs are registered for simplified synthesis using time budgeting.

This module is entirely synchronous to the wclk for simplified static timing analysis. As shown in below Figure 16,

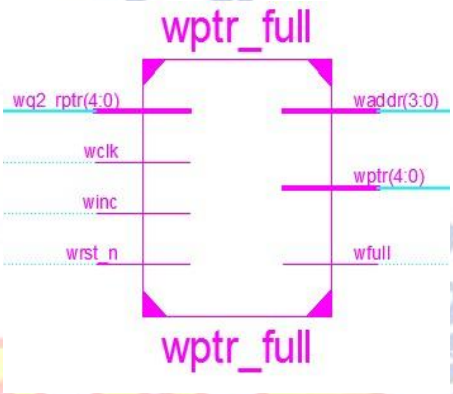

Fig 16: Schematic Diagram of wptr_full

3.4 EDA Playground

EDA Playground gives engineers immediate hands-on exposure to simulating and synthesizing SystemVerilog, Verilog, VHDL, C++/SystemC, and other HDLs as shown in Fig 17. All you need is a web browser.

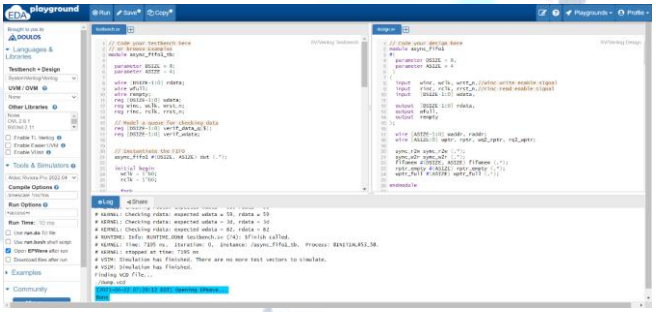

Fig 17: EDA Playground

- With a simple click, run your code and see console output in real time.
- View waves for your simulation using EP Wave browser-based wave viewer.
- Save your code snippets ("Playgrounds").
- Share your code and simulation results with a web link. Perfect for web forum discussions or emails.

Great for asking questions or sharing your knowledge.

- Quickly try something out – Try out a language feature with a small example. – Try out a library that you're thinking of using.
- For Quick prototyping – try out syntax or a library/language feature.
- When asking questions on Stack Overflow or other online forums, attach a link to the code and simulation results.
- During technical interviews to test candidates' System Verilog/Verilog coding and debug skills.

Trying out different verification frameworks: UVM, SV Unit, Plain Verilog, or Python.

A. Aldec Riveria Pro 2022

Riviera-PRO™ addresses verification needs of engineers crafting tomorrow's cutting-edge FPGA and SoC devices. As shown in Fig.18 Riviera-PRO enables the ultimate testbench productivity, reusability, and automation by combining the high-performance simulation engine, advanced debugging capabilities at different levels of abstraction, and support for the latest Language and Verification Library Standards.

- Extensive simulation optimization algorithms to achieve the highest performance in VHDL, Verilog/SystemVerilog, SystemC, and mixed-language simulations.

The industry-leading capacity and simulation performance enable high regression throughput for developing the most complex systems.

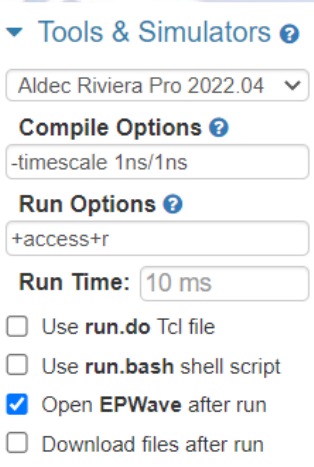

Fig 18: Simulating By using Tools

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 as shown in below Figure 1,

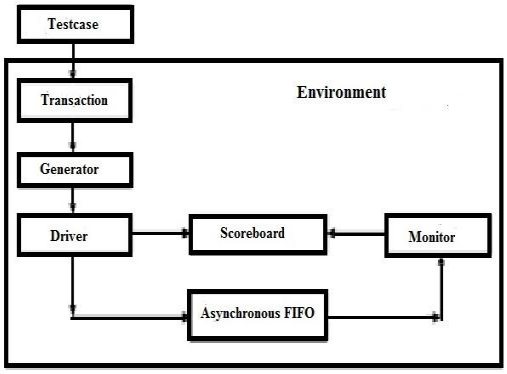

Fig 19: Verification of Testbench

A. Interface

The interface consists of bundle of wires i.e. multiple signals used to connect the Testbench to the DUT. The mod ports 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.

B. Testcase

The Testcase module will instantiate the environment module and calls the methods in the environment

C. 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

D. 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.

E. 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.

F. Monitor

This is the receiver section that receives the data from the receiver side of the Asynchronous FIFO. This Monitor, it gets recorded by themselves the transaction and we can able to see the purpose of the Transaction of the Asynchronous FIFO and The transaction is also sent to the scoreboard using mon2sb mailbox.

G. 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.

H. Environment

The environment blocks 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. RESULT

The Simulation results are Monitored from the EDA Playground's EP Waveform. As we can see that all the FIFO designs succesfully designed the output waveform. At last, after the simulating we are going to note down the waveforms are shown in below Fig 19. It Describes as the log data, where it gets self estimated output in binary or hexadecimal of rdata and wdata.With the help of the binary numbers the verfication of this Asynchronous FIFO is going to be done and are going to verify the data with the two necessary actions of Gray Code Counter as ,

- Data Send to Write Clock Domain(Fig 20) and
- Data Recived at Read Clock Domain.(Fig 21)

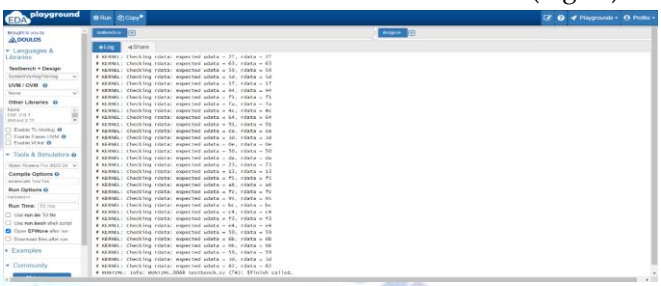

Fig 20: Log Data After Simulating

Fig 21: Data Sent to Write Clock Domain

Fig 22: Data Received at Read Clock

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 Fig 20, 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, As shown in Fig 21 "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 Fig 21, 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 Multi Clock domain systems.

Conflict of interest statement

Authors declare that they do not have any conflict of interest.

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