ECE-593

FUNDAMENTALS OF PRE-SILICON VALIDATION

IMPLEMENTATION AND VERIFICATION OF ASYNCHRONOUS FIFO USING BOTH CLASS-BASED AND UVM METHODOLOGIES

VERIFICATION PLAN (Till Milestone 2)

Section-1 & Team-1

Kumar Durga Manohar Karna (kkarna@pdx.edu)
Mohammed Abbas Shaik (mohammee@pdx.edu)
Nivedita Boyina (nivedita@pdx.edu)
Nikhitha Vadnala (vadnala@pdx.edu)

Contents:

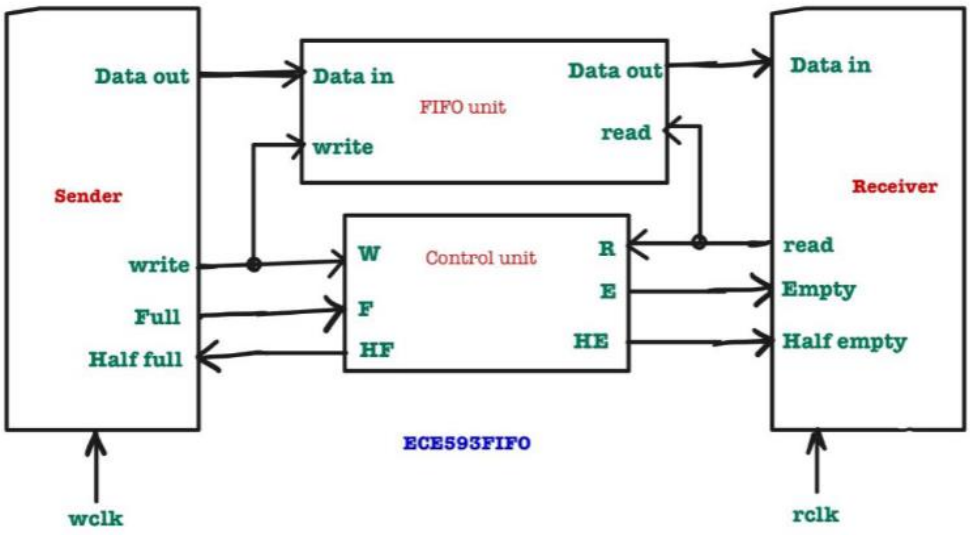
Introduction:

An Asynchronous FIFO (First in first out) is a specialized type of FIFO buffer that allows data to be read and written in different clock domains. Unlike a synchronous FIFO, where the read and write operations are controlled by the same clock signal, an Asynchronous FIFO will have separate read and write clock domains, enabling data transfer between two independently clocked systems.

The Asynchronous FIFO acts as a bridge between the sender and receiver modules, providing a reliable and efficient means of transferring data across different clock domains. It prevents data loss or data corruption during the transfer process, even when the write and read clocks are completely asynchronous to each other.

The design of an Asynchronous FIFO involves several challenges, such as metastability handling, gray code pointers, and synchronization techniques. Metastability can arise when data is transferred across asynchronous clock domains, which leads to undefined logic states. Proper metastability handling techniques, such as synchronizers, ensure reliable data transfer across clock domain boundaries. Grey cloud pointers with the property of changing only one bit at a time, which reduces the metastability. It also tracks the read and write positions in the memory. Additionally, the design incorporates synchronization techniques to properly coordinate the data transfer between the asynchronous clock domains.

Design Specifications:



Level Block Diagram of the Design System

To understand the FIFO design, we need 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.

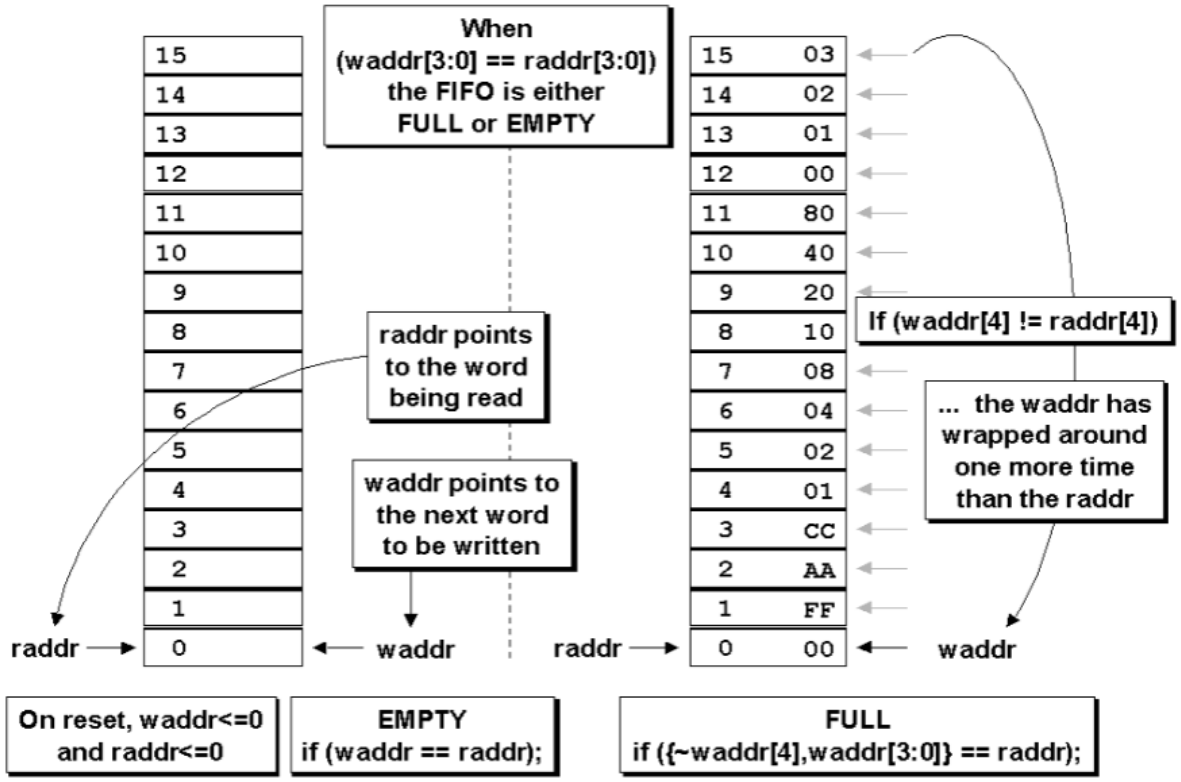
Similarly, the read pointer always points to the current FIFO word to be read. On reset, both pointers are reset to zero, the FIFO is empty, and the read pointer is pointing to invalid data (because the FIFO is empty, and the empty flag is asserted). As soon as the first data word is written to the FIFO, 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 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.

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. This is a problem. The FIFO is either empty or full when the pointers are equal, but which?

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 as shown in Figure 1 (the FIFO has wrapped and toggled the pointer MSB). The same is done with the read pointer. If the MSBs of the two pointers are different, it means that the write pointer has wrapped one more time than the read pointer. If the MSBs of the two pointers are the same, it means that both pointers have wrapped the same number of times.

Using n-bit pointers where (n-1) is the number of address bits required to access the entire FIFO memory buffer, the FIFO is empty when both pointers, including the MSBs are equal. And the FIFO is full when both pointers, except the MSBs are equal.



FIFO depth Calculations:

$f_A > f_B$ with idle cycles in both write and read.

Writing frequency = f_A = 120MHz.

Reading Frequency = f_B = 50MHz.

Burst Length = No. of data items to be transferred = 1024.

No. of idle cycles between two successive writes is = 4.

No. of idle cycles between two successive reads is = 2.

The no. of idle cycles between two successive writes is 4 clock cycle. It means that, after writing one data, module A is waiting for four clock cycle, to initiate the next write. So, it can be understood that for every five clock cycles, one data is written.

The no. of idle cycles between two successive reads is 2 clock cycles. It means that, after reading one data, module B is waiting for 2 clock cycles, to initiate the next read. So, it can be understood that for every three clock cycles, one data is read.

Time required to write one data item = $5 * 1/120$ MHz =41.67 ns = 42 ns

Time required to write all the data in the burst = 1024 * 42 ns. = 43008 ns.

Time required to read one data item = 3 ∗ (1/50) MHz =60 ns.

So, for every 60 ns, module B is going to read one data in the burst.

So, in a period of 43008 ns, 1024 no. of data items can be written.

The no. of data items can be read in a period of 43008 ns = 43008 ns / 60 ns= 717

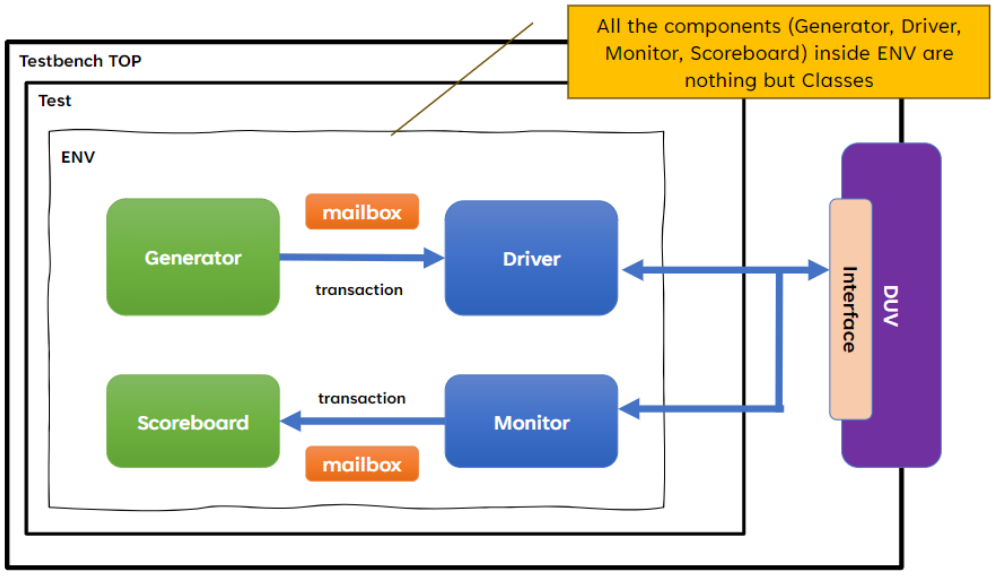
The remaining no. of bytes to be stored in the FIFO = 1024 – 717 = 307.

So, the FIFO which must be in this scenario must be capable of storing 307 data items.

So, the minimum depth of the FIFO should be 307.

Testbench Architecture in System Verilog :

For Class Based Verification



Generator:

Generator class is responsible for generating the stimulus by randomizing the transaction class and sending the randomized class to driver via mailbox. The transaction class handle is declared, randomized and an event is added to indicate the end of the generation of packets. We used "triggering" here to indicate end of the generation.

Driver:

A Driver will receive the stimulus from the generator and drive them to the DUT by assigning the transaction class values to interface signals. Adding a local variable to track the number of packets driven and increment the variable in the drive task.

Monitor:

A Monitor is used for sampling the interface signals and will convert the signal level activity to a transaction level. Once sampled, the transaction will be sent to the scoreboard using the mailbox. In the monitor class, we instantiate the virtual interface which allows to pass an argument to a method, then the extended class method handle will get assigned to the base class handle.

Interface:

An interface is a bundle of signals or nets through which a testbench communicates with a design. Allows the number of signals to be grouped together and represented as a single port. A virtual interface also allows you to pass an interface as an argument to a task, function, or method. This way, you can reuse the same code for different interfaces without changing the code.

Mailbox:

Mailbox is a communication mechanism in System Verilog that allows messages to be exchanged between process. Here, we have used 2 mailboxes. One is Generator to Driver and the other one is for Monitor to Scoreboard transactions. We use mailbox because it stores temporarily in system's memory object.

Transaction:

Fields required to generate the stimulus are declared in the transaction class. Here, we are giving input stimulus in randomized way, thereby making use of the "rand" keyword.

Scoreboard:

Scoreboard receives the sampled packet of the signals from the monitor via the mailbox and then compares each transaction with the expected result, an error will be reported if the comparison results in a mismatch.

Environment:

Environment class contains the Mailbox, Generator, Driver, Scoreboard, Monitor and Transaction classes. The communication between Generator and Driver are instantiated (mailboxes) and pass the interface handle. A task is added to call all the above methods.

Test:

The test is responsible for the creation of environment and also setting the number of transactions to be generated and initializing the stimulus driving process.

Testbench:

The testbench is the top module which connects the test module and the DUT module by making use of the interface. A clock is also generated in this module itself.

Test Plan:

Designed the functionality of an asynchronous FIFO across various clock domains.

Handling of metastability and synchronization between the write and read clock domains.

Verified the proper reset and initialization of the FIFO and its internal signals.

Performed various write and read operations with different burst sizes and idle cycles, as per the design specifications.

The write operation is performed successfully ensuring that FIFO accepts the data whenever the write signal is enabled, and the data is stored.

The read operation is performed successfully ensuring that FIFO sends valid data whenever the read signal is enabled, and the data is retrieved.

FIFO full condition is verified by ensuring data is written into FIFO till maximum depth is reached. FIFO empty condition is verified by making sure that the data read is done from the FIFO until it becomes empty.

With a conventional testbench with class-based verification we have checked the very basic functionality of the design.

Created different classes in the testbench environment and implemented the class-based verification for the DUT. Generating various randomized stimuli in the generator and verifying it for number of transactions.

Contribution:

Kumar Durga Manohar Karna - Gone through the design specifications and have implemented the generator logic in class read to write module and top module. Used mailbox as a medium to communicate with scoreboard. Created a class-based environment to instantiate generator, driver, monitor, scoreboard along with mailboxes to enable the communication between generator-driver and monitor-scoreboard.

Mohammed Abbas Shaik - Have one through the design specifications and have implemented the driver logic using class with read and write operations. Created a virtual interface to communicate with DUV and a function to communicate with mailbox. Have created an interface for the testbench and have implemented clocking for driver and monitor.

Nivedita Boyina - Have gone through the design specifications and have implemented the monitor logic with read and write operations including half full, half empty, full and empty conditions. Used mailbox as a medium to communicate with scoreboard. Have created the test environment.

Nikhitha Vadnala - Have gone through the design specifications and created a class for scoreboard which includes the read and write operations with half full, half empty, full and empty conditions and displays the output. Have created an interface for the testbench and have implemented clocking for driver and monitor.

Resources:

1.http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf

2.http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO2.pdf

3.https://hardwaregeeksblog.files.wordpress.com/2016/12/fifodepthcalculationmadeeasy2.pdf

4.https://ieeexplore.ieee.org/abstract/document/7237325

GitHub link:

https://github.com/BNiVeDiTa29/ECE593s24_team_01_Async_FIFO