# **Designing** Asynchronous FIFO, functioning of its waveforms **and calculating its depth**

**A PROJECT**

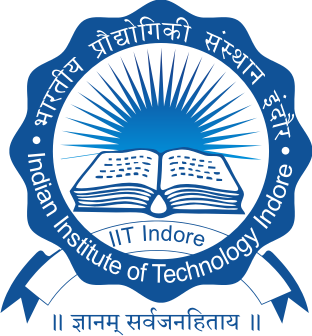
***of***

## Master of Technology

## VLSI Design and Nanoelectronics (VDN)

by

**Atharv Alok Limaye (Roll no. 2302102019)**



**DISCIPLINE OF ELECTRICAL ENGINEERING**

**INDIAN INSTITUTE OF TECHNOLOGY INDORE**

**January 2024**

**Contents**

|  |  |
| --- | --- |
| Sno. | Topic |
| 1 | Introduction |
| 2 | Working Principle |
| 3 | Architecture |
| 4 | Asynchronous FIFO Pointers |
| 5 | Synchronizers & Binary Gray Counter |
| 6 | Full & Empty Logic Block |
| 7 | Applications |

**Introduction**

A First-In-First-Out (FIFO) memory is a type of data buffer that stores data and releases it in the same order as it was received. It is extensively used in digital systems where data needs to be transmitted between two units operating at different clock domains. An Asynchronous FIFO is a particular type of FIFO where the write and read operations occur in different clock domains. The independence of clock domains helps in avoiding the complexities of synchronizing different clocks, which is especially useful in applications like data buffering, rate-matching, and clock-domain crossing in communication systems.

**Working Principle**

An asynchronous FIFO operates by writing data into memory cells using a write clock (WCLK) and reading data using a read clock (RCLK). The two clocks are independent, which means they can have different frequencies and phases. This independence requires mechanisms for safe data transfer and pointer management to ensure data integrity. The basic elements of an asynchronous FIFO include:

1. Write Pointer (w\_ptr): Points to the memory location where the next data word will be written.

2. Read Pointer (r\_ptr): Points to the memory location from where the next data word will be read.

3. Memory Array: Stores data between write and read operations.

4. Control Logic: Manages the synchronization between read and write pointers to avoid overflows and underflows.

**Key Challenges**

The key challenges in the design of an asynchronous FIFO are:

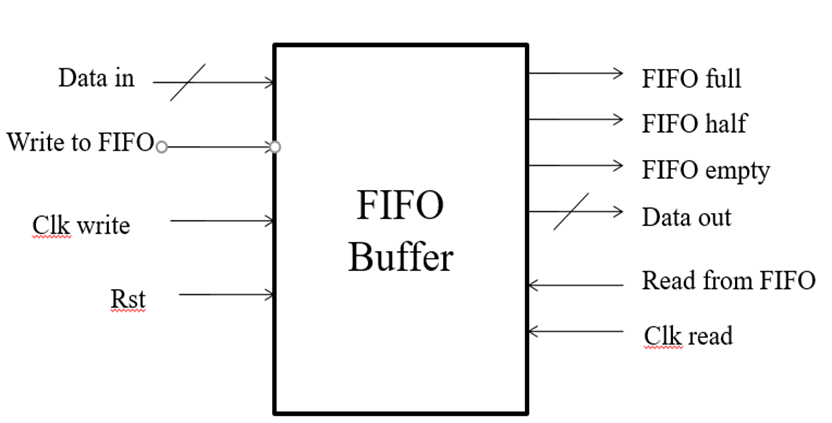
- Metastability: Since the write and read operations occur in different clock domains, the design needs to account for the risk of metastability when signals are sampled asynchronously.

- Synchronization: Proper synchronization between the read and write pointers is crucial to ensure that data is read correctly and to avoid reading or writing from the same location simultaneously.

- Full and Empty Conditions: Determining when the FIFO is full (write pointer catches up with the read pointer) or empty (read pointer catches up with the write pointer) is necessary to prevent overflows and underflows.

**Metastability** is a phenomenon that occurs when a signal is sampled in a state that falls between a stable logic '0' and a stable logic '1'. This is especially critical when dealing with asynchronous systems, where data is transferred between different clock domains.

* Root Cause: In an asynchronous FIFO, the read and write operations are clocked independently. When one clock domain samples a signal from another clock domain (e.g., read clock sampling the write pointer), there is a chance that the data might be in transition exactly at the time of sampling. This can cause the sampled signal to become metastable.
* Impact: A metastable state can lead to unpredictable behavior of digital circuits, as the sampled value may take some time to settle to a definite '0' or '1'. This can cause issues in control logic, like generating incorrect full/empty signals, potentially leading to data corruption.
* Mitigation: To minimize the risk of metastability, multi-stage synchronizers are used. A common practice is to use two or more flip-flops in series to resynchronize the asynchronous signals into the target clock domain. The additional flip-flops provide time for the metastable state to resolve, reducing the probability of errors.

[](https://user-images.githubusercontent.com/72481400/114535379-9c257180-9c6d-11eb-972d-fcfaf2aca1eb.png)

**Synchronization**

Synchronization between the read pointer and write pointer is a critical challenge because of the independent clock domains involved. Accurate synchronization ensures that the read and write pointers are properly managed to avoid reading from or writing to the wrong memory locations.

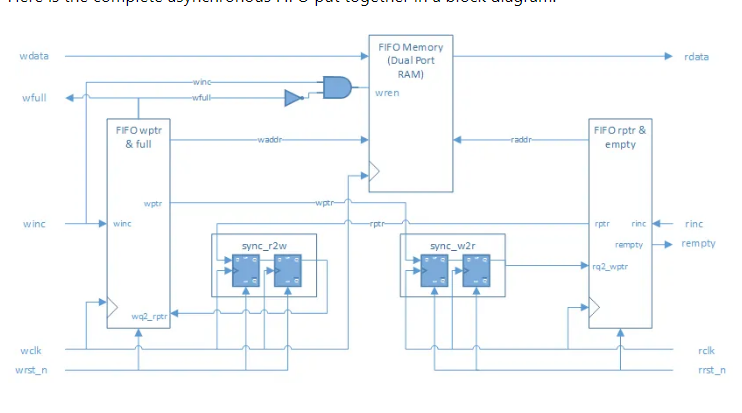
Gray Code Usage: To address synchronization challenges, Gray code is often used for pointers. Gray code is a binary numeral system where only one bit changes between successive values. This minimizes the chance of multiple bits transitioning simultaneously, which can cause incorrect pointer values due to partial sampling.

Crossing Clock Domains:

Write Pointer in Read Domain: The write pointer needs to be synchronized into the read clock domain to determine how much data has been written into the FIFO and whether the FIFO is full. The read clock domain uses this synchronized write pointer to adjust its read operations accordingly.

Read Pointer in Write Domain: Similarly, the read pointer is synchronized into the write clock domain to track how much data has been read out and whether the FIFO is empty. The write clock domain uses this information to decide when it can write new data.

Timing Constraints: The synchronization process introduces delays because the multi-stage synchronizers take time to resolve metastability. This delay can affect how quickly the system recognizes changes in the pointers, potentially impacting the speed of data transfer. Therefore, balancing the depth of the synchronization stages against timing requirements is crucial in the design.

[](https://user-images.githubusercontent.com/72481400/111077783-6c713580-8518-11eb-83e7-8f8824ece83f.png)

**Full and Empty Conditions**

Accurately determining when the FIFO is full or empty is crucial to prevent overflows and underflows, which can cause data loss or data corruption.

**FIFO Full Condition**:

* The FIFO is considered full when the write pointer has advanced such that it is exactly one position behind the read pointer (in a circular memory arrangement). Writing more data in this state would cause overwriting of unread data.
* Challenge: Detecting this condition requires a comparison between the synchronized read pointer (in the write clock domain) and the write pointer. However, due to synchronization delays, there can be a lag in recognizing the full condition, potentially allowing one extra write.
* Mitigation: A common solution is to add a margin by designing a slightly larger FIFO buffer than required, allowing for a safe window to avoid overflow.

**FIFO Empty Condition**:

* The FIFO is empty when the write pointer and the read pointer are at the same position, indicating that all written data has been read.
* Challenge: Similar to the full condition, detecting the empty condition requires a comparison between the synchronized write pointer (in the read clock domain) and the read pointer. The synchronization delay can result in a lag before recognizing the empty state, potentially causing a read operation when no new data is available.
* Mitigation: To avoid issues during the empty condition, it is common to add a slight delay before generating the empty signal, ensuring that the pointers have stabilized after synchronization.

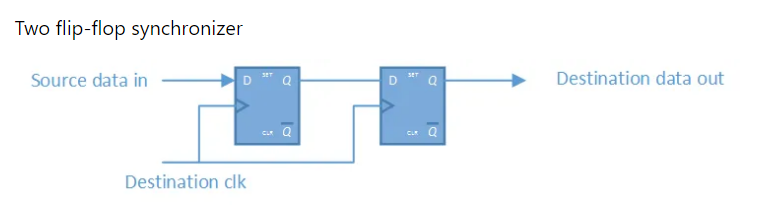
Handling Pointer Wrap-Around: Since the FIFO is typically implemented as a circular buffer, the pointers will wrap around once they reach the end of the memory array. The control logic must account for this to accurately detect full or empty conditions, adding complexity to the design.

**Pointer Synchronization**

In an asynchronous FIFO, the Gray code is often used for pointer synchronization because it ensures that only one bit changes at a time, minimizing the chance of errors when crossing clock domains. Here’s how synchronization is achieved:

Write Pointer Synchronization: The write pointer is sampled into the read clock domain to determine the relative positions of the read and write pointers for generating the "FIFO Full" signal.

Read Pointer Synchronization: Similarly, the read pointer is sampled into the write clock domain to track the relative positions for generating the "FIFO Empty" signal.

[](https://user-images.githubusercontent.com/72481400/111077754-49468600-8518-11eb-9bfd-87d57d6dcd14.png)

This two-way synchronization ensures that read and write operations do not overlap improperly, avoiding issues like overwriting unread data or reading unwritten data.

**Full and Empty Conditions**

- FIFO Full Condition: The FIFO is considered full when the write pointer is one position behind the read pointer in a circular buffer. This means that writing another element would overwrite data that has not yet been read.

- FIFO Empty Condition: The FIFO is empty when the write pointer and the read pointer are at the same position, indicating that there are no new data elements to read.

The control logic uses these conditions to assert "Full" or "Empty" flags, which can halt write operations when the FIFO is full or read operations when the FIFO is empty.

**Applications**

Asynchronous FIFOs are widely used in various fields such as:

* Crossing Clock Domains: They help to bridge communication between modules operating at different clock frequencies, such as in microprocessor and DSP interfaces.
* Data Rate Matching: Used in systems where data is produced and consumed at different rates, like in video processing and network buffering.
* SoC Design: Facilitates data transfer between different IP blocks operating in different clock domains, allowing for modular design in complex systems-on-chip.