

1 IP Task

1.1 Introduction

The following hardware diagram (figure 1) is a simple FIFO implementation with a parity check controller that will check for any fault hazard that may occur to bits during their flow inside the electronics.

It has been built using a top entity called *top* that contains two components: *my_FIFO* and *parity_checker*. All the constant are stored in the *constant_pkg* file.

In addition to the *clk* and the asynchronous *rst_n* signals, this block has three main inputs: *data_i*, *valid_i*, *grant_i*, each of them with a specific function:

- **data_i**: it is simply the input data stream. This data vector can change its size accordingly with a parameter set inside .txt file.
- **valid_i**: this signal is sent by the source and states the validity of the incoming data.
- **grant_i**: this signal works as a read enable signal. If the client is ready to read a signal, it informs FIFO that it can share data.

The block has also three outputs: *data_o*, *valid_o*, *grant_o*:

- **data_o**: it is the output data stream. This data vector changes its size accordingly with a parameter set inside .txt file.
- **valid_o**: this signal is sent by the source and states the validity of the incoming data that can be controlled both by the FIFO and the Parity Checker.
- **grant_o**: this signal works as a write enable signal. If the source is ready to write a signal inside the FIFO, it informs that it is waiting to write.

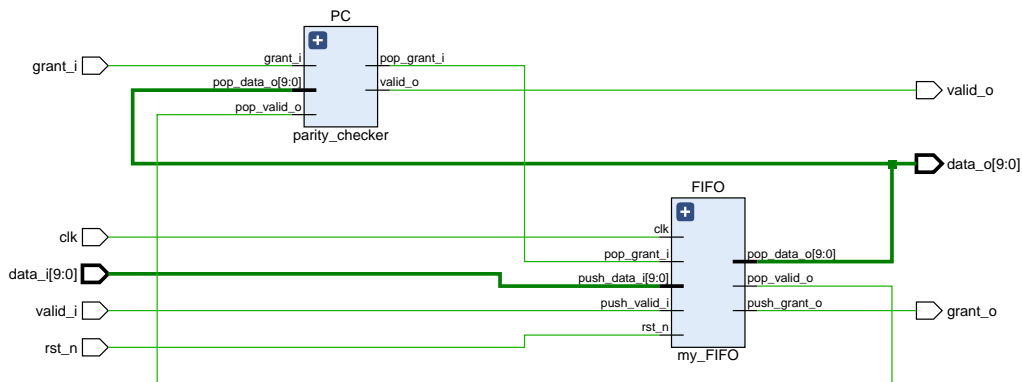


Figure 1: Block Diagram with FIFO and Parity Check modules

1.2 FIFO

The unit is parameterized via two **generic** constants: `DATA_WIDTH`, which defines the data bus size, and `FIFO_DEPTH`, which establishes the maximum number of words the buffer can store.

- **Write Port:** Managed by `push_data_i` and `push_valid_i`. The output signal `push_grant_o` acts as a *not full* indicator, informing the source that the FIFO is ready to accept new data.
- **Read Port:** Managed by `pop_data_o` and `pop_valid_o`. The `pop_grant_i` (Read Enable) signal is driven by the receiver to request data consume to the output.

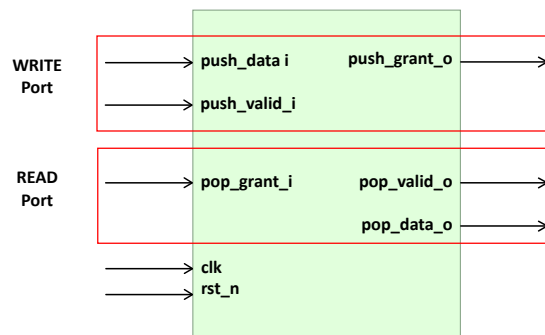


Figure 2: FIFO port structure

The FIFO is a circular buffer that implements a SRAM memory logic. Internal operation is divided into three main logical sections, that are three different processes: **fifo_indexes**, **FSM_write** and **FSM_read**. Even if a finite state machine is not the most efficient way to create a FIFO, the exercise required using a Moore FSM for its implementation.

The **fifo_indexes** process serves as the "mechanical" core of the module. It handles Circular Indexing, with the `write_index` and `read_index` pointers that track current memory locations. The system implements automatic roll-over when a pointer reaches $FIFO_DEPTH - 1$, and resets it to 0 on the next clock cycle. The `fifo_count` signal tracks the exact number of words currently stored. This value is critical for determining **FULL** and **EMPTY** states consistently, especially during simultaneous read and write operations.

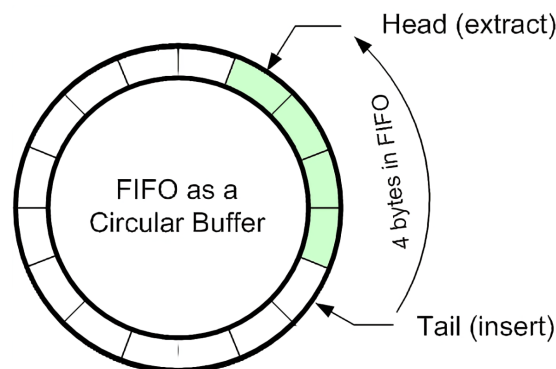


Figure 3: Circular FIFO representation

The write FSM (**FSM_write**) controls memory access authorization:

- **IDLE/WRITE:** The FIFO accepts incoming data as long as space is available.
- **FULL:** Once `fifo_count` reaches the maximum depth, the `push_grant_o` signal is de-asserted, preventing further writes and protecting against *overflow*.

The read FSM (**FSM_read**) controls the output dataflow:

- **IDLE:** The default state when the system is waiting for a read request. If the receiver asserts `pop_grant_i`, the FSM transitions to the **READ** state to begin data delivery.
- **READ:** In this state, the FIFO provides data to the output bus.
 - If `pop_grant_i` is de-asserted, the FSM returns to **IDLE**.
 - If the FIFO contains only one last word (`fifo_count` = 1) and a read occurs, the FSM transitions to **EMPTY**.
 - Otherwise, it remains in **READ** as long as data is available and requested.
- **EMPTY:** This state is reached when the buffer has no more data. The signal `pop_valid_o` is de-asserted to inform the receiver that the output is invalid. The FSM remains here until `push_valid_i` indicates that new data is being written into the FIFO, allowing a transition back to **READ** or **IDLE**.

Decoupling the pointer logic from the state machines allows for more efficient implementation of the FIFO. The use of local variables to determine operational status (`is_writing`, `is_reading`) ensures that the internal counter remains synchronized even when the FIFO is accessed for both reading and writing within the same clock cycle.

1.3 Parity Checker

The `parity_checker` module acts as a combinational filter positioned between the FIFO and the final receiver. Its primary role is to validate the integrity of the data stored in the FIFO by performing a real-time parity check. If a data packet is found to be corrupted, the module automatically discards it, preventing the transmission of invalid information to the output port.

The module calculates parity using XOR chain (figure 4). This logic operates based on the selected configuration: for Even Parity, a valid packet must result in a total XOR sum of `error = '0'`, while for Odd Parity, the expected result for a valid packet is `error = '1'`.

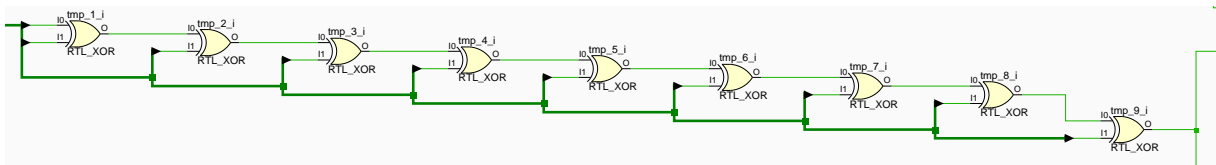


Figure 4: XOR chain in RTL schematic

An important aspect of this design is the use of the commutative property of the XOR operation ($A \oplus B = B \oplus A$). Since the XOR reduction aggregates all bits to produce the final parity sum, the result remains mathematically identical regardless of the bit sequence. This ensures that the module correctly detects errors whether the parity bit is located at the MSB or LSB, providing full compatibility with the required IP parameters without needing structural changes to the logic.

Although the current implementation performs a full-vector reduction, the logic is designed to be transparent to the parity bit position. Whether the parity bit is located at the Most Significant Bit (MSB) or the Least Significant Bit, the cumulative XOR result remains mathematically identical.

The handshake signals (**valid/grant**) are used to implement a "drop" rule for corrupted data. If an error is detected (**error** = '1', Odd Parity), the **valid_o** signal is forced to '0'. This ensures that the receiver never samples corrupted data, hiding the packet. In case of a parity error, the module asserts the **pop_grant_i** signal to the FIFO regardless of the external **grant_i** status. This forced acknowledgment "consumes" the corrupted word from the FIFO, clearing the buffer for the next valid packet.

Being a purely combinational block, the **parity_checker** module introduces no clock cycle latency. The parity evaluation and the handshake occur within the same cycle, the data appears at the FIFO output.

2 Verification

The verification of the system is conducted through a testbench designed to validate the integrity of the data flow and the effectiveness of the parity-based filtering mechanism. The simulation environment integrates three main functional components that interact with the DUT: a Grant Generator, a Traffic Generator, and a real-time Checker.

The **Grant Generator** simulates the behavior of an external receiver with varying readiness levels. It initially operates with a 50% duty cycle to stress-test the FIFO's ability to handle backpressure and data accumulation, subsequently switching to a continuous-assertion mode to verify maximum throughput. This component is essential to ensure that the DUT module correctly stops or resumes data transmission based on downstream availability.

The **Traffic Generator** manages the input stimuli by injecting a sequence of data words into the FIFO. The methodology involves distinct phases: an initial reset period, a burst phase to reach the full capacity of the buffer, and a random traffic phase. This sequence is designed to trigger all possible states of the internal Finite State Machines, particularly focusing on the transition between IDLE, WRITE, and FULL states. By modulating the *valid* signal and the input data, the generator creates the necessary conditions to observe how the system reacts to both saturated and empty buffer scenarios.

The **Checker** serves as the primary monitor for performance metrics and functional correctness. Instead of simple point-to-point checks, it adopts a transaction-based counting methodology. By synchronizing with the clock, it monitors the successful handshakes at both the input and output interfaces. A packet is marked as successfully sent only when both the input valid and output grant signals are high, while a packet is recorded as received only when the output valid and input grant signals coincide.

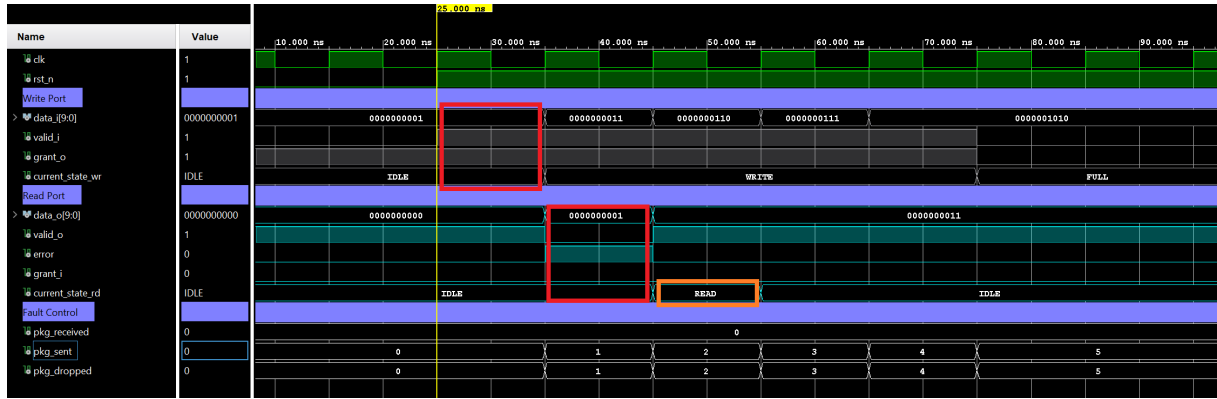


Figure 5: First faulted packet

In Figure 5, it is possible to see that after 25 ns, when `rst_n` is brought high, the first valid packet arrives (first red square).

In the next clock cycle (second red square), this data is visible at the output, while the combinational logic of the *parity_checker* detects an error: '000000001' has an odd number of '1's, but the MSB states it is an even number, which is incorrect (we can also see the opposite case, with the entire word composed of zeros and the LSB stating it is an odd number, which is again incorrect). The system reacts by outputting the data with `valid_o` = 0, even if `grant_i` is not asserted high, so the client will not accept the data and will be notified it is not valid.

In the third clock cycle, it is possible to see that the FIFO Finite State Machine was brought to the READ state in order to discard the previous data, returning then to IDLE since `grant_i` = '0'.

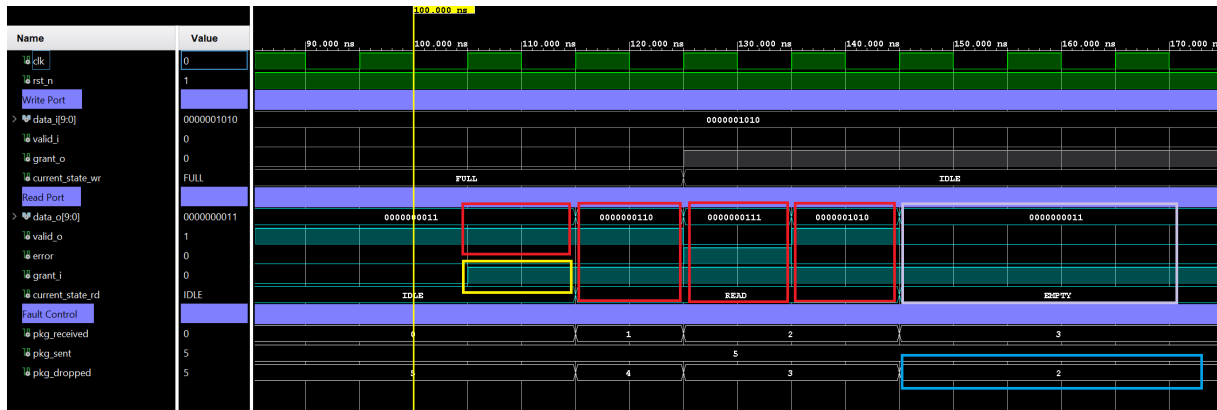


Figure 6: FULL and EMPTY state with number of dropped packets

The FIFO is then filled until it becomes full. The numbers with which it has been filled are visible in Figure 5: '000000011', '000000110', '000000111', '0000001010'. Then `grant_i` is asserted high (yellow box) and immediately the first data '000000011' is output as valid (first red box). The second data is valid. The third data is an error, and so the error signal is brought high and the valid signal is brought low. The final data (last red box) is valid. Now the FIFO is empty and enters the EMPTY state. At the end of these transactions, 2 packets have been dropped since the beginning; indeed, the counter in the light blue box

indicates exactly this condition.

The results of the simulation are derived from the correlation between these counters. The total number of dropped packages is calculated as the difference between the packets sent and those successfully received by the client. This calculation confirms how the drop works: since the Parity Checker is designed to consume corrupted data without asserting the output validity, any discrepancy between the sent and received counts represents a detected parity error that has been correctly filtered.