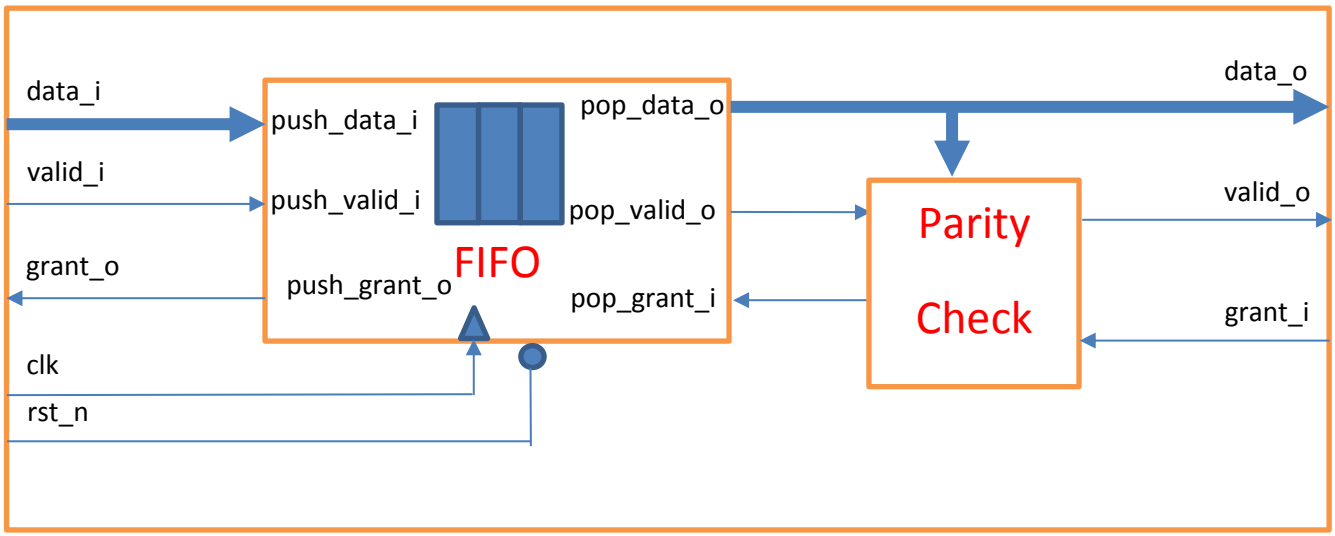
Design and verification of a FIFO with parity checker:

The focus of this task is the design of a small logic block composed by a Parity-checker block and a synchronous FIFO. First the incoming data (***data_i***) flow though the FIFO, then the output (***data_o***) is checked in the parity checker to avoid that corrupted data is propagated to the output channel. To do so, the parity checker inhibits (set it to 0) the ***valid_out*** in case of bit disparity. The following figure describes the block diagram of the proposed task.



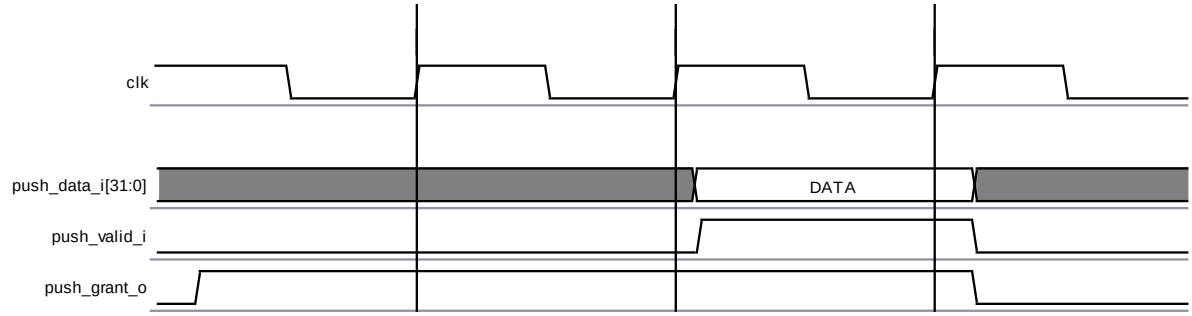
Design:

The FIFO must be designed in order to have a configurable DATA_WIDTH and FIFO_DEPTH (Assume a depth value >= 2, configurable, while the flow control will be based on a simple request-grant protocol.

FIFO Push protocol:

On the rising edge of the clock, if ***push_valid*** and ***push_grant_o*** are "1", the FIFO stores the ***push_data_i***.

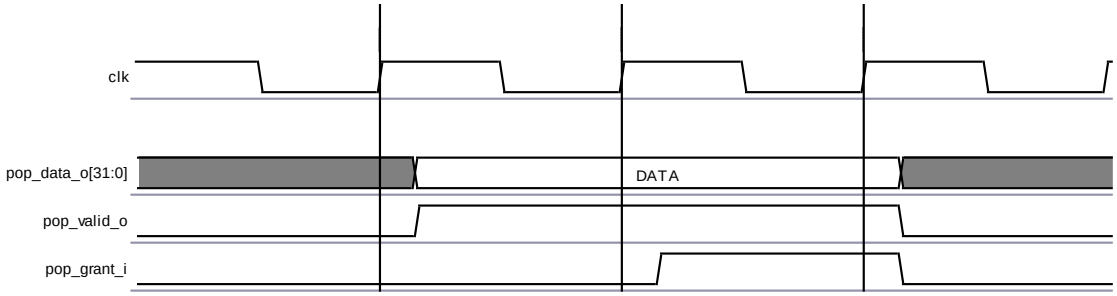
push_grant_o is 0 when the FIFO is FULL and 1 if not FULL. Grant must not depend on ***push_valid_i***, but is a function of the FIFO state. FIFO must be able to accept one transaction per cycle.



FIFO Pop protocol:

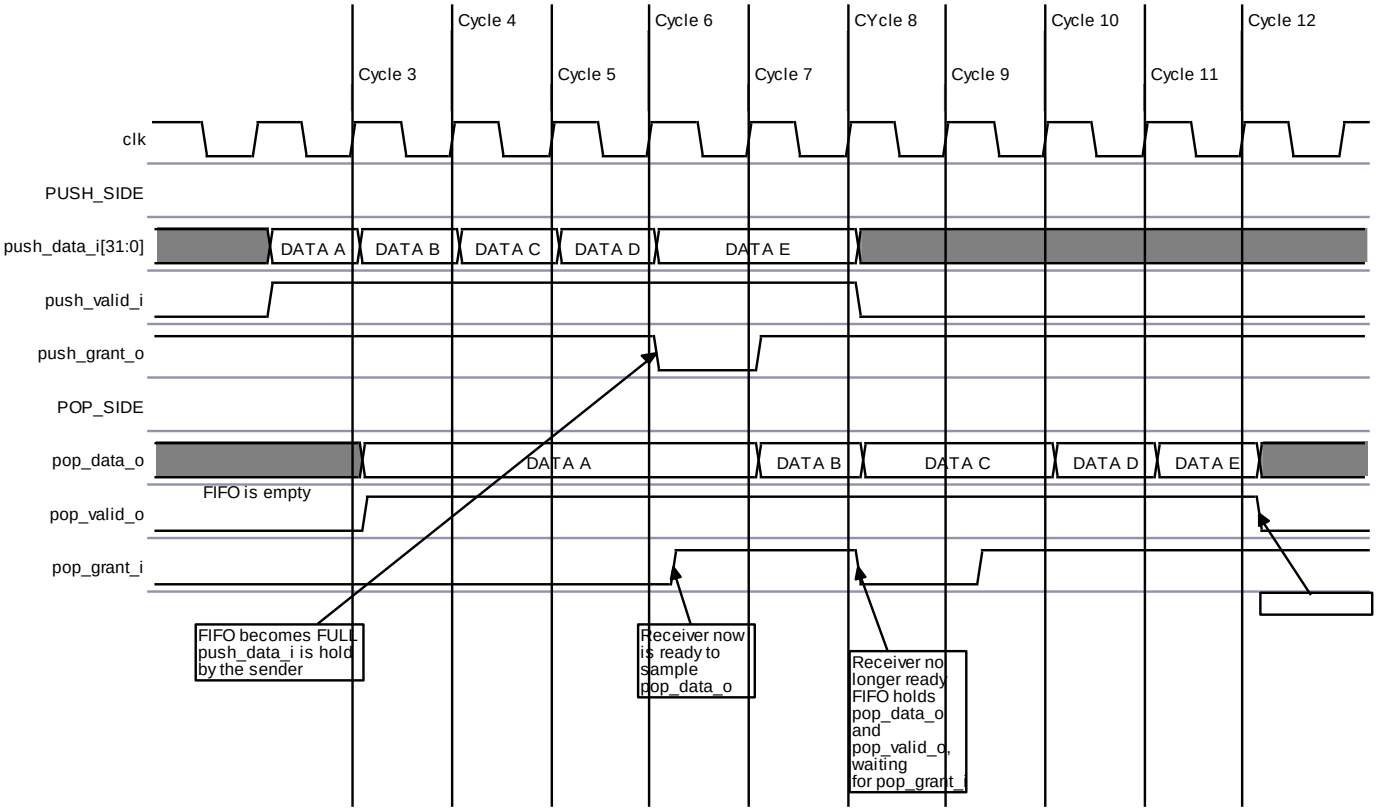
When there is some data to deliver, the FIFO uses the ***pop_valid_o*** signal to inform the receiver (consumer) that a valid data is available. If the receiver is ready, it asserts the ***pop_grant_i*** signal. When ***pop_grant_i*** and ***pop_valid_o*** are both 1, the FIFO considers dispatched that data, and moved to the next data (if any).

When FIFO is not empty, the ***pop_valid_out*** is set to 1, as longer there is some data stored inside the FIFO. If FIFO becomes empty, ***pop_valid_out*** is set to 0. ***Pop_valid_out*** is function of the state of the FIFO, and does not depend on the ***pop_grant_i***.



FIFO Push – Pop Waveforms example:

Consider for instance, a FIFO with a depth of 4 slots, and 32 bits of data-width.



At cycle 1-2, FIFO is empty, ***pop_valid_o*** is 0

At cycle 3, ***push_data_i*** (DATA_A) is valid, and since the FIFO is not FULL (***push_grant_o*** == 1). On the rising edge of Cycle 3, the transaction is stored in the FIFO. FIFO is empty, and then ***pop_valid_o*** is 0.

At Cycle 4: a ***pop_data_o*** is valid, but ***pop_grant_i*** is 0, then FIFO holds DATA_A on its output. On push side, DATA_B is valid, and on the rising edge of cycle 4 the push_data_in is pushed inside (FIFO not full again).

At Cycle 5: a ***pop_data_o*** is valid, but ***pop_grant_i*** is 0, again the FIFO holds DATA_A. On push side, DATA_C is valid, and is pushed inside (FIFO not Full again) on the rising edge of the clock.

At Cycle 6: a ***pop_data_o*** is valid, but ***pop_grant_i*** is 0, again the FIFO holds DATA_A. On push side, DATA_D is valid, and is pushed inside (FIFO not Full again).

At Cycle 7: a ***pop_data_o*** is valid, ***pop_grant_i*** is 1, the FIFO dispatches DATA_A and increments its internal pointer. On push side, DATA_E is valid, but FIFO is FULL. The sender (producer) must hold ***push_data_i*** on its outputs, to avoid data loss.

At Cycle 8: a ***pop_data_o*** is valid, ***pop_grant_i*** is 1, the FIFO dispatches DATA_B and increments its internal pointer. FIFO is no longer FULL and on push side, DATA_E is valid and is pushed inside (FIFO not Full again).

At Cycle 9: there is nothing to push. Receiver is not ready (***pop_grant_i***) therefore then FIFO holds DATA_C on its output.

At Cycle 10: there is nothing to push. : ***pop_data_o*** is valid, ***pop_grant_i*** is 1, the FIFO dispatches DATA_C and increments its internal pointer.

At Cycle 11: there is nothing to push. : ***pop_data_o*** is valid, ***pop_grant_i*** is 1, the FIFO dispatches DATA_D and increments its internal pointer.

At Cycle 11: there is nothing to push. : ***pop_data_o*** is valid, ***pop_grant_i*** is 1, the FIFO dispatches DATA_E and increments its internal pointer.

At Cycle 13: there is nothing to push. FIFO is empty.

Parity Checker:

Parity checker, checks the correctness of ***pop_data_o***. In case of disparity, it must drop the data, which means that must grant even the receiver is not ready. Corrupted data must not be sampled in the receiver (***valid_o*** will be always 0 in case of corrupted data).

Task Guidelines

The design will be coded used HDL languages (Verilog or VHDL). Main parameters can be stored in a text file (e.g. an include file with the IP configuration) or passed as module parameters. The FIFO parameters are:

- Depth of the FIFO
- DATAWIDTH of the FIFO defined as WIDTH+1 where 1 is the parity bit.

The parity checker parameters are:

- Parity: EVEN or ODD
- Parity BIT: MSB or LSB

The desired structure of the project is to a hierarchical block (top) which instantiates the FIFO and the parity checker. The IP must be designed in order to be readable (comments, naming conventions ….) and it is required a clear separation between the data-path and the controls logic that compose the design. Comments are welcome!! The design is fully synchronous (rising edge of the clock) and the reset must be asynchronous, active low.
The FIFO must be capable to operate at full speed (1 transaction per clock cycle, both on the in and out channel).

The sequential elements of the FIFO must be a SRAM behave model, instantiated in the FIFO block. So pushing and popping requests are synchronous and they take 1 cycle to be completed. The Flow control signals (pop_valid_o, push_grant_o) must be generated by a sequential logic (MOORE Type)

Output of the IP task:

- HDL code
- Block diagram
- Brief description of all parts
- Compile script

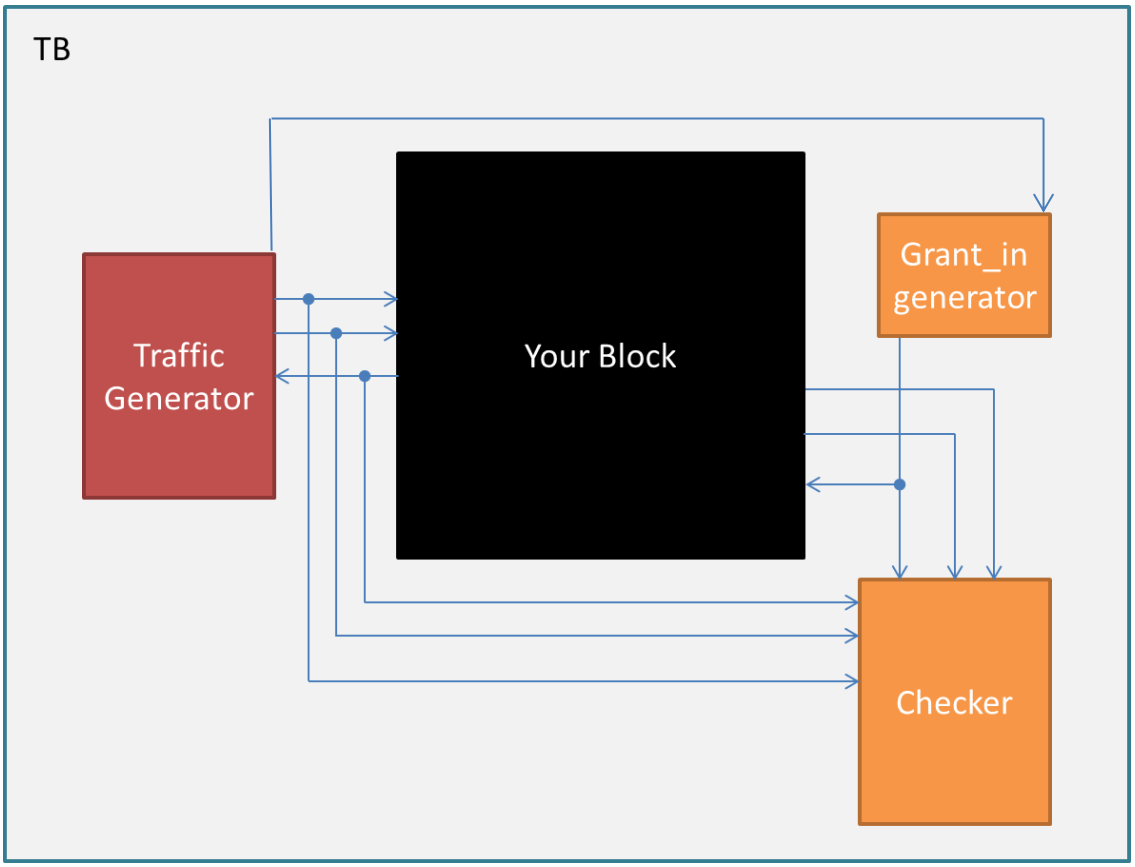
Verification

It is required to build the verification environment (testbench) to validate the implemented block. First the testbench must supply the test vectors to the FIFO following the above described flow control. Then random errors must be applied to the generated vectors (no more than one fault per transaction), and finally, a checker must monitor the output of the FIFO to ensure the correctness of the design.

Type of traffic to Inject:

1) Full the FIFO
2) Empty the FIFO
3) Random traffic, at max BW (this means that ***pop_grant_i*** is always 1)
4) Random traffic, with random ***pop_grant_i*** (50% low, 50% high)

The testbench should check correctness of popped data, not only parity bit. It must detect any problem like data losses, and must count passed and dropped transactions.



The Grant_in generator must be designed to test all the possible scenarios listed at the beginning of this section.

Output of Verification task:

- HDL code of the design and the verification environment
- Brief description of all parts, methodology and results
- Simulation script