Pre-interview exercise: FIFO with Parity Check

Ariel Podlubne

February 21, 2017

1 Introduction

The present report describes the work done to design and test a FIFO with a Parity Check. The sections bellow describe all their parts and how they were designed.

2 Design and Test

The methodology for the design of the FIFO with the Parity Check as well as for its test environment consisted in a layer design. This means that all the small internal modules were designed and tested first. Later, each one was incorporated into its corresponding "outer" layer or TOP module. A description of each one and its simulation are shown below.

2.1 FIFO and Parity Check

This two modules were designed separately and each one tested with its own testbench. Afterwards, they were both instantiated into a TOP module called *FifoParityCheckerUB_TOP*.

2.1.1 FIFO

The FIFO has different input and output ports, which can be divided in *data*, *control* and *status*.

- Data
  - push_data: In port for the data to be pushed.
  - pop_data: Out port for the data to be poped.
- Control
  - push_valid_i: In port for the *sender* to push data.
  - pop_grant_i: In port for the *receiver* to pop data.
- Status

- push_grant_o: Out port for the *sender* to advise whether the FIFO is *full* or not.
- pop_valid_o: Out port for the *receiver* to advise whether there is data available to be poped or not.

The FIFO contains an internal array where the incoming and outgoing data is stored. There is a pointer to keep track of the pushed data and another one for the poped one. They are used to write or read data to the correct position as well as to know the status of the FIFO (full or empty).

The design can be seen as three concurrent blocks and they all depend on each other. The first one is to take care in the event of a *reset*, setting the pointers mentioned before and the status of the FIFO to zero (one for the empty register). This block also handles the pushed data, storing it in its location depending on the write pointer but only if the FIFO is not full. The second block handles the output data port, writing it with the data the read pointer points to. It also sets the operation state of the FIFO, depending on the control items (push_valid_i and pop_grant_i) with the full and empty registers. This is done like this because the user might want to push data, but the FIFO cannot take it if it is full. The same happens with the poped data. The VHDL code is: `operation <= not full_reg and push_valid_i & not empty_reg and pop_grant_i;`

The third block updates the registers accordingly to **operation** and also checks if the FIFO has been completely filled or emptied.

The aim of its testbench is to mimic the input signals (push_data, push_valid_i and pop_grant_i) shown in the exercise to prove the correct operation of the FIFO. The result of the simulation is shown in Figure 1.

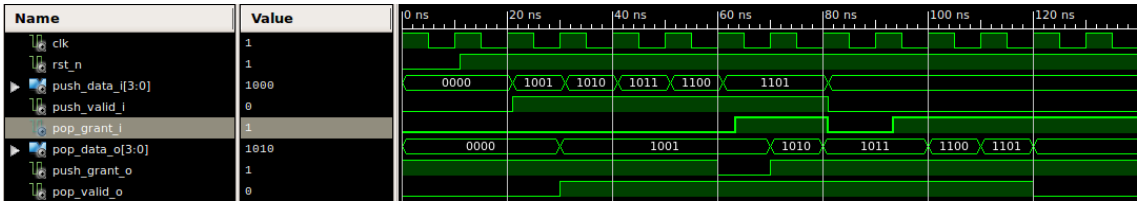

Figure 1: FIFO's simulation.

2.1.2 Parity Check

Data that it is *pushed* by the sender includes a parity bit. Therefore, the task of this block is to calculate the parity of the raw data (without the parity bit) and check if they match. This module can be divided in three *concurrent* blocks. The first one is to retrieve the parity bit, which can be in the MSB or LSB. This will depend on constant values defined on "my_pkg.vhd". The second one, calculates the parity of the raw data (without the included parity bit) by XOR-ing all its bits. If the result is 1, the calculated parity will be *odd* and *even* if 0. The third one compares if the included parity and the calculated one match. If they do, *valid_o* will be 1 and 0 if they do not. As this is a concurrent design, there is an enable condition that has to be true, which happens when the receiver is ready to read new data (grant_i) and the FIFO has data available (valid_i). Its output (valid_o) will be 0 if this condition is not met.

The testbench checks the correct operation of the logic, taking into account that the parity in the data will be in the LSB and will be ODD. This means that the LSB will be 1 if the parity is ODD or 0 if it is EVEN. This parameters can be easily changed in "my_pkg.vhd".

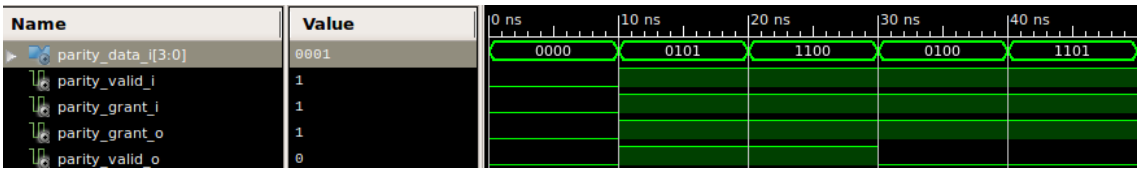

Figure 2: Parity Check's simulation.

Four different cases are shown in Figure 2.

- data_i=0101: The raw data is 010. Therefore, it has ODD parity which matches the included one so valid_o is 1.
- data_i=1100: The raw data is 110. Therefore, it has EVEN parity which matches the included one so valid_o is 1.
- data_i=0100: The raw data is 010. Therefore, it has ODD parity which does not match with the included one so valid_o is 0.
- data_i=1101: The raw data is 110. Therefore, it has EVEN parity which does not match the included one so valid_o is 0.

2.1.3 FifoParityCheckerUB_TOP

The modules described in section 2.1.1 and 2.1.2 are instantiated in this one. Its testbench also mimics the signals in the example, the same way as described in section 2.1.1. The only difference is that now valid_o will be different due to the ParityCheck module described in section 2.1.2. The simulation is shown in Figure 3.

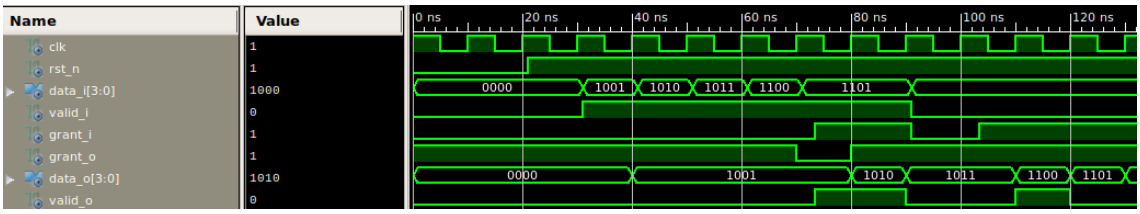

Figure 3: FIFO and Parity Check TOP module's simulation.

2.2 Test Environment

This module serves as a testbench for the one described in section 2.1.3. The FifoParity-CheckerUB_TOP module is instantiated in the TestEnvironment_TOP module as well as the three ones described next.

Table 1: Traffic Type

Traffic Type	Status	grant_i	Function
00	FULL	0	Data is pushed unti the FIFO is full and nothing is poped.
01	EMPTY	1	Data is constantly poped but not pushed.
10	100% BW	1	Data is pushed and poped simultaneously.
11	50% BW	Changes	Data is poped 50% of the time that it is pushed (50% BW).

2.2.1 Traffic Generator

This module receives the *test vectors* provided by the stimulus in the testbench. It separates the raw data and the traffic type (Table 1) to be sent to their respective modules. Depending on the traffic type and whether the FIFO is full or not (grant_o), *valid_i* changes its state depending on the current test vector. The structure of the test vector is: N bits of data and 2 bits for the traffic type. The array with the test vectors can be changed in "my_pkg.vhd".

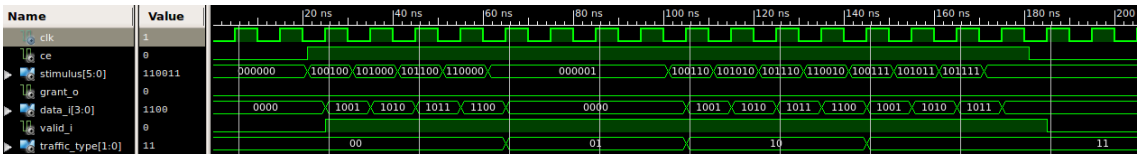

Figure 4: Traffic Generator module's simulation.

The result of the simulation is shown in Figure 4, where it can be seen how data_i and traffic_type change accordingly to stimulus.

2.2.2 Grant_in Generator

As the name indicates, this module generates the state of grant_i. It takes a two bit vector with the traffic type as input and changes its output (*grant_i*) accordingly. This module can be seen as a 2-to-4 decoder with a "special feature" for the "11" input, which is a frequency divider (by 2). The four different possibilities that traffic_type represents are described in table 1.

The simulation shown on Figure 5 matches with table 1.

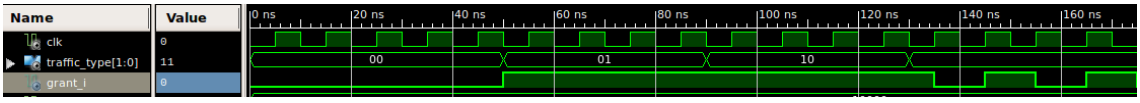

Figure 5: Grant_In Generator module's simulation.

2.2.3 Checker

This module checks the incoming and outcoming data. It counts poped data with correct parity (pass) and incorrect parity (drop), which depends on the output of the Parity Check module described in Section 2.1.2. It also checks if there is data loss. The "Checker" module

is then divided into two separate *concurrent* blocks shown in Figure 6. It is also important to note that it contains a chip enable input ("ce") that has to be set to 1 when the test starts (test vectors are being looped) and set to 0 when finished.

Pass and drop are chosen to be 8-bit output ports. Therefore, the length of the test (number of test vectors) has equal or less than 255, otherwise these two ports might overflow. In case there is need for larger tests, these ports can have more bits, but the module will have to be re-synthesized. These two counts are done concurrently in the "Pass and Drop" block of the module. They are obtained by checking whether valid_o is 1 (pass) or 0 (drop). There is a condition for this, which is that grant_i has to be 1. This means that the receiver is ready to pop new data and data_o is updated.

As the design is based on a FIFO, the N-poped value will be the N-pushed one. Therefore, the "loss block" checks if the N-poped data matches the N-pushed one. All data that it is pushed into the FIFO is stored into the RAM. A 4-bit counter keeps track of how many pushes have been and its output is used as an address pointer. Another 4-bit counter keep track of how many pops have been and its output is used as a read pointer. Whenever data is poped, the corresponding RAM position is read to obtain the N-pushed value and it is compared with data_o which corresponds to the N-poped value. They are compared in a simple comparator that sets its output to 1 if both its inputs are equal. This signal acts as the chip enable for the third 4-bit counter which is where the number of lost data is counted.

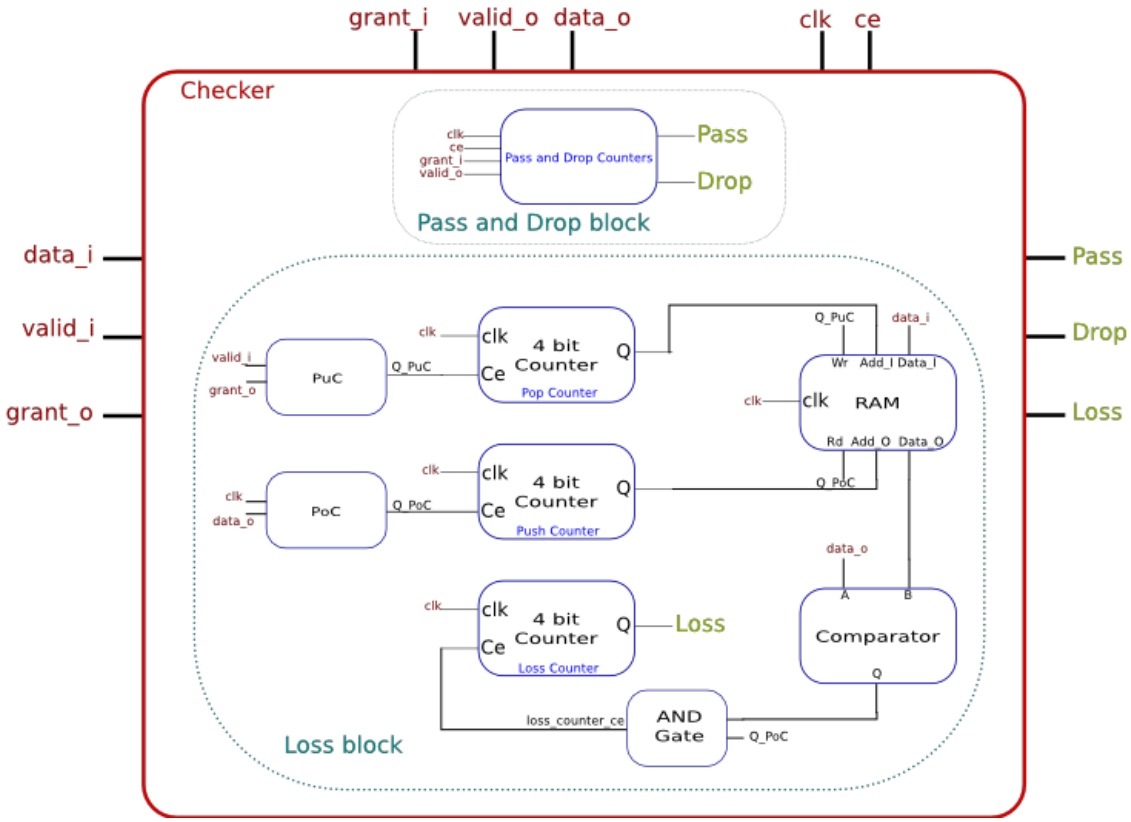

Figure 6: Checker module's block diagram.

The testbench was done so as to mimic the example of the exercise. As there is no lost

data there, a data_o value was changed to test the "loss block" operation. Therefore, 3 passed, 2 dropped and 1 lost values are expected and the simulation displayed in figure 7 shows the correct operation of this module.

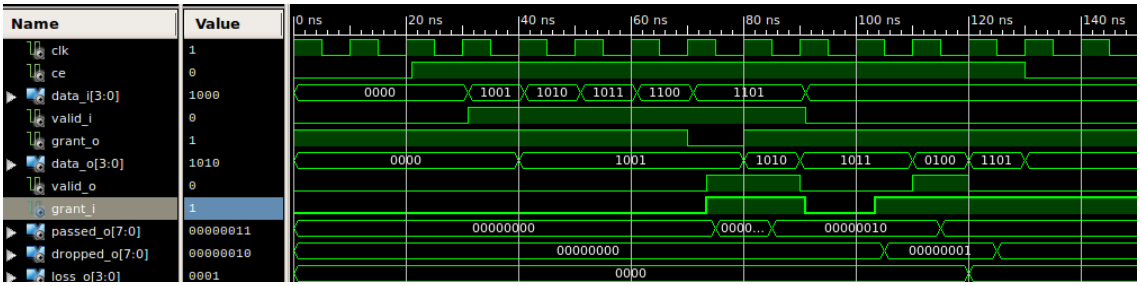

Figure 7: Checker module's simulation.

2.2.4 TestEnvironment_TOP

All the modules described before are instantiated and connected accordingly as requested in the exercise. The testbench was designed to test the different type of traffic requested:

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).

An array with 16 test vectors was used in the testbench for this module. They have the same structure as described in Section 2.2.1. There are 4 of them for each type of traffic as the FIFO was tested with a depth of 2 (or 4 slots). The result obtained in the simulation is displays on Figure 8.

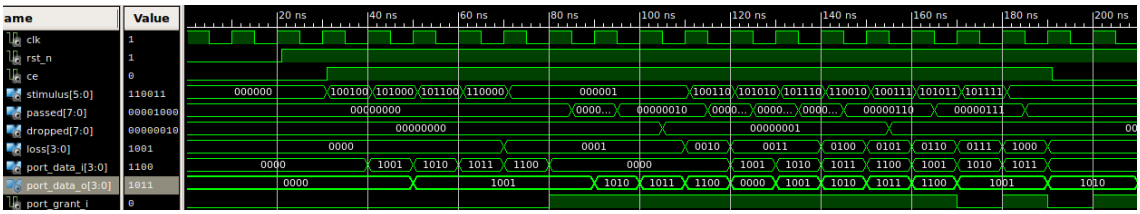

Figure 8: TestEnvironment module's simulation.

3 Conclusion

Each section and subsection showed the implemented design and their corresponding testbench. A layer-design allowed to code and test all the modules needed in an efficient manner.

By testing simpler modules individually to guarantee their correct operation made it easier to instantiate them in their TOP design. This also allowed testing the TOP designs to be easier.